Old Dominion University
ODU Digital Commons

Computer Science Theses & Dissertations | Computer Science

Summer 1992

High Performance Issues on Parallel Architectures

Peter J. Looges
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/computerscience_etds

 Part of the Computer Sciences Commons

Recommended Citation
Looges, Peter J.. "High Performance Issues on Parallel Architectures" (1992). Doctor of Philosophy (PhD), dissertation, Computer Science, Old Dominion University, DOI: 10.25777/n4gg-qe54
https://digitalcommons.odu.edu/computerscience_etds/114

This Dissertation is brought to you for free and open access by the Computer Science at ODU Digital Commons. It has been accepted for inclusion in Computer Science Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.

High Performance Issues on Parallel Architectures

Peter J. Looges

B.S. May 1985, Rensselaer Polytechnic Institute
M.S. May 1991, Old Dominion University

A Dissertation submitted to the Faculty of
Old Dominion University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

Old Dominion University
August 1992

Approved by:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

In an effort to reduce communication latency in mesh-type architectures, these architectures have been augmented by various types of global and reconfigurable bus structures. The static bus structures provide excellent performance in many areas of computation especially structured numerical computations, but they lack the flexibility required of many large numerical and non-numerical applications. Reconfigurable bus systems have the dynamic adaptability to handle a much wider range of applications. While reconfigurable meshes can often yield constant time results for many problems, the cost of this performance is paid in the number of processors required. While in actuality the majority of these processors are employed as switching elements for the bus system and often do little actual computation.

In an effort to reduce the processor cost while maintaining performance and communication flexibility, we present a new hybrid parallel array architecture with the goal of optimizing the best features of arrays with global buses and arrays with reconfigurable bus systems. The result is an architecture of n processing elements and a bus interconnection network which requires very basic circuitry to construct and control.

This architecture allows prefix computations, such as prefix sum, prefix maximum(minimum) to be accomplished in $O(\log n)$ time. These functions then form the building blocks for complex procedures, which more fully exploit the communication flexibility of the architecture. Application of the architecture to graph theory produces optimal algorithms for graph properties such as spanning forest bipartiteness, fundamental cycles, bridges and biconnected components. Other optimal algorithms for the more complex least common ancestor and the connected component problems are also presented. By design, all algorithms maintain optimality for very large sparse graphs. We further examine the architecture's ability to handle basic image processing tasks as well as its potential to simulate other parallel architectures and theoretic models.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

©Copyright by Peter J. Looges 1992
All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgments

No dissertation is prepared without considerable advice, help, firm guidance, and support. I would especially like to thank my advisor, Stephen Olariu, for his many hours of guidance, hard questions, and encouragement. His high personal standards are an excellent example to all.

I am indebted to the members of my Ph.D. committee, Dr. C. E. Grosch, Dr. L. Wilson, Dr. J. L. Schwing, Dr. P. Bogacki and Dr. S. Olariu for their careful reading of my dissertation and their insightful comments. I would like to express my appreciation to Dr. Kurt Maly for the atmosphere conducive to research and scholarship which he maintains in the department and for his financial support of my graduate studies.

I cannot begin to thank my wife, Heather, for her support during the last couple of years. Additionally, she must be credited with convincing me that it was possible to work and study, without that initial impetus none of this may have happened.

This dissertation is dedicated to my parents, who often saw ability where I thought there was none. As we get older we realize how smart our parents often were and we desire to make them proud. I thank them for the support and eternal patience over the years, all of which contributed more than they will ever realize to the success of this endeavor.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

State of the Art

1.1 Why Parallel Architectures?

Many problems require considerable computational power not only because they are inherently computationally intense, but also because the solution is needed in *real time*. Problems of this type include:

- Artificial Vision,
- Data Base Searches,
- Finite Element Analysis,
- Computational Fluid Dynamics,
- Simulation,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Optical Ray Trace,
- Signal Processing[30, 64].

As computing power continues to increase the limits of the physical circuitry are quickly being reached. One of these primary limits is the speed of light. Since signals cannot propagate faster than the speed of light, about one foot in a nanosecond, the goal has become one of packing circuitry closer together to reduce communication delay. This circuit packing also seems to be reaching its limits under current technologies. The limit is that if circuits are packed too closely they begin to interact unpredictably and unreliably [4, 30, 72].

The resolution to these limits is becoming recognized as parallel architectures. These are architectures of many complete processors all operating concurrently to solve a problem. While an architecture of n processors can be theoretically n times as powerful as a single processor this is often not the case. Problems, however, cannot always be distributed evenly over the n processors nor is processor intercommunication free, both are primary contributors to inefficiency.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.2 SIMD vs. MIMD

Computer architectures may be grouped into four main classes which were first defined by Flynn in 1966 [20]. These are:

SISD Single Instruction Single Data;

MISD Multiple Instruction Single Data;

SIMD Single Instruction Multiple Data;

MIMD Multiple Instruction Multiple Data.

The first two categories will not be addressed here. SISD is standard sequential processing and MISD has only been utilized in special purpose applications. SIMD and MIMD tend to be more general purpose in nature and therefore of interest in the examination of general purpose high performance architectures. Actual implementations of the SIMD type system include DAP, MPP, GAPP, Illiac IV, Connection Machine(early versions), CLIP4, Adaptive Array Processor, GRID, and GF11. Actual examples of the MIMD system include NCUBE, iPSC, Connection Machine(later versions), PASM, WARP, and transputer based machines [30].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

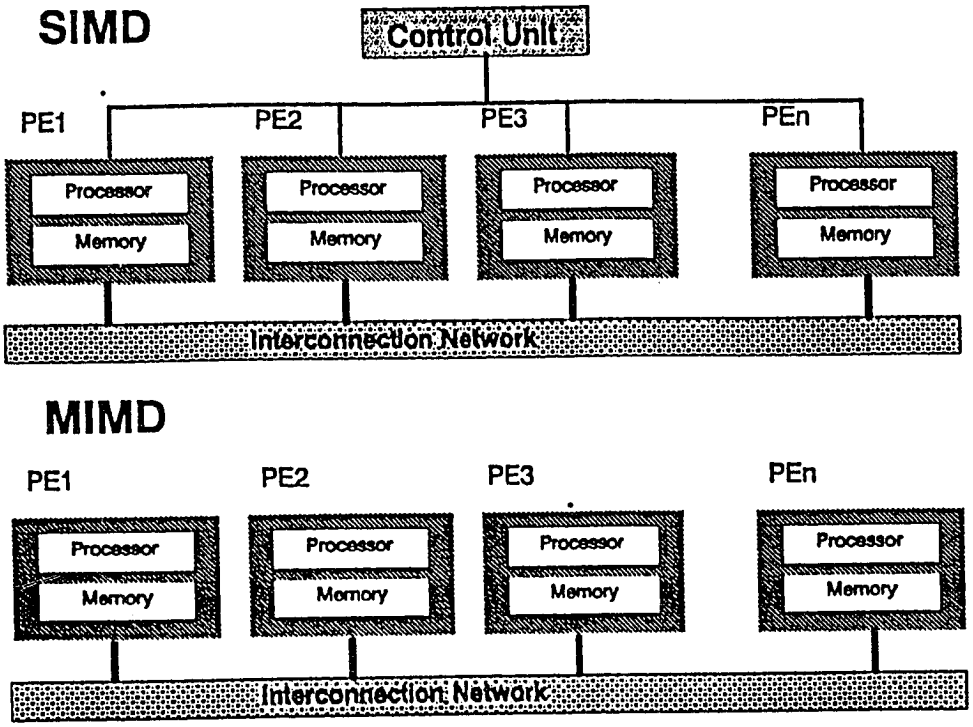

Figure 1.1: The basic SIMD and MIMD architectures

The difference between the two remaining categories is shown in the block diagram of figure 1.1. The primary difference is that in SIMD architectures the processors act synchronously under the control of a single control unit on the data with in their local memories. Thus all processors execute the same instruction or do nothing in a wait state as directed by so predetermined parameter. In MIMD architectures each processor acts independently, communicating with other processors when necessary. Each architecture has specific advantages over the other. Some of these advantages are listed below:

- Advantages of the SIMD Architecture

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Ease of programming - only a single program is required and all processors are synchronous;
- Reduces interprocessor communication delay - all processors are always synchronized;
- Less memory is required - only a single copy of the program need be maintained;
- Minimal instruction decoder cost - only one decoder is required.

• **Advantages of the MIMD Architecture**

- More Flexible - no constraints on the operations which can be performed concurrently;
- Conditional statements more efficient - each processor executes as if it were the only processor;
- No SIMD control unit cost;
- Variable time instructions more efficient - SIMD waits for slowest processor at each instruction, MIMD waits for slowest processor in a block of instructions (see figure 1.2) [30, 56, 71, 67].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

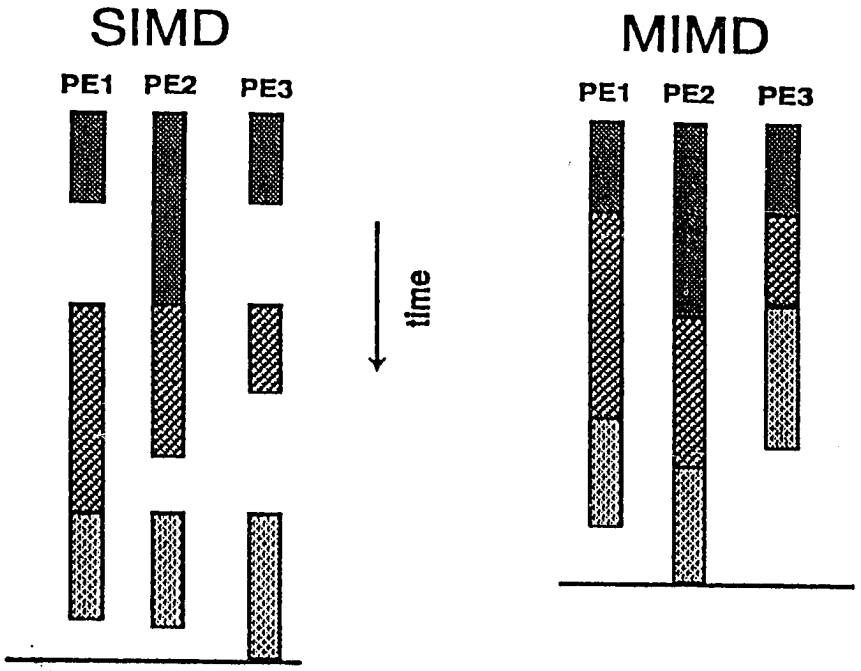

Figure 1.2: The relative efficiency of variable time instruction on SIMD and MIMD

At this point it is appropriate to state that for the purposes of the model presented later in this work, an SIMD architecture will be assumed at all times.

1.3 Introduction to Parallel Architectures

A parallel architecture is a set of processing elements connected via some type of interconnection network [69]. Many different interconnection topologies have been proposed or implemented in recent years. These include the n-cube [10, 51, 79], the perfect shuffle [74], the mesh of trees [35, 36, 50], the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pyramid [48] the multi-dimensional mesh array [5, 6], the array with global buses[1] [1, 11, 33, 43, 46, 75], and parallel arrays with reconfigurable bus systems [23, 39, 47, 80].

The most basic type of processing element interconnection scheme is that of a linear array. In this scheme, a processing element is connected to at most 2 other processing elements. The main advantage of this scheme is its simplicity and regularity, which in turn reduces cost. Large linear arrays can easily be constructed from smaller linear arrays by coalescing together a series of smaller arrays. Linear arrays have been shown to be useful in low level image processing and some numerical computation (see [32, 34, 60] for further details).

While the linear network has definite advantages from the cost and engineering standpoints, there are some very significant drawbacks. Primary among these is the difficulty of long distance communication. Standard linear arrays are characterized by **large communication latency** in addition to a **small communication bandwidth** [17]. This implies that communication time between two processing elements could be of the order of the

[1]The array with global buses is also known as the array with multiple broadcast.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

size of the array. For communication intensive algorithms this would be an excessive and intolerable cost. The more complex array topologies solve the communication cost problem, but at the expense of complexity and higher engineering costs.

We examine two of the general mesh interconnection schemes, the mesh with global buses [11] and the parallel array with reconfigurable bus systems [39]. Both of these architectures are designed to improve communication ability of the standard mesh of processing elements. The mesh with global buses has a global bus system which interconnects each row and column of processing elements (see figure 1.3). This topology improves long distance communication, but still has limitations in that only one processing element can send information over a particular row or column at a time. The parallel array with reconfigurable bus system consists of an element to element bus system with connections that are controlled by each processing element in the mesh (see figure 1.4). This control of interconnections allows the buses to be split and reset dynamically throughout computation. Thus multiple processing elements in the same column or row can use the same bus as long as it is split and only one processing element is writing to a particular

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

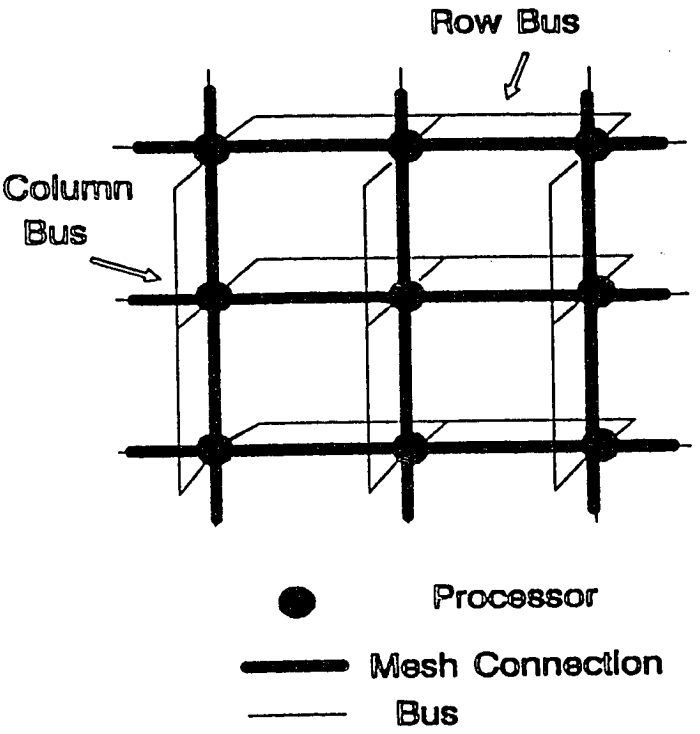

Figure 1.3: The Mesh with Global Buses

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

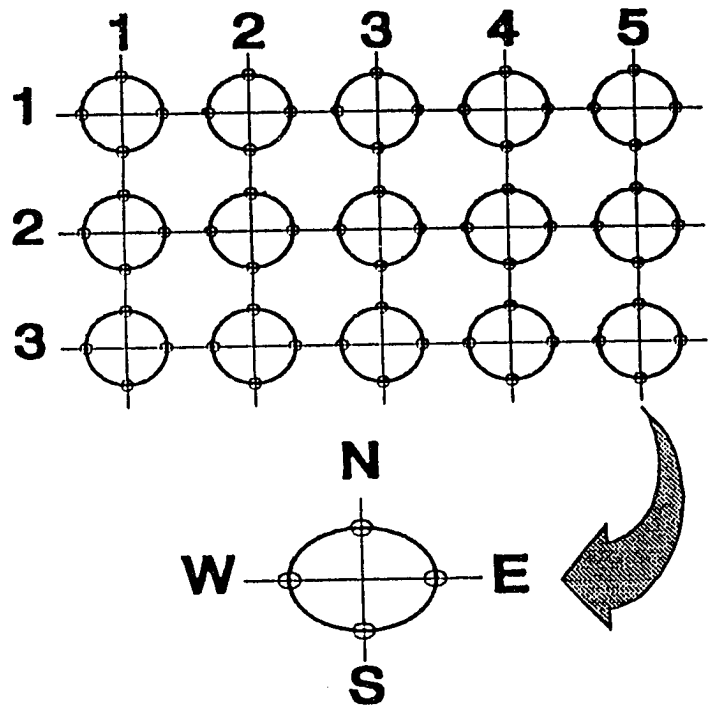

Figure 1.4: The Parallel Array with Reconfigurable Global Bus System (PARBS)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sub-bus at a time.

These architectures obtain excellent results, but often at a high processor cost. For example, constant time sorting of n elements is achieved on the parallel array with reconfigurable bus system at the cost of n^2 processing elements [40]. The mesh with global buses accomplishes sorting of n^2 elements in $O(n)$ time on n^2 processing elements . The tradeoff of time versus processing elements is not always as direct. In this case, however, it can be shown that on a parallel array with reconfigurable bus system of n processors, n elements can be sorted in $O(n^{\frac{1}{2}})$ time since the reconfigurable bus system tends to respond like the global bus system for the dense data case.

Since in the sparse data case, many processing elements of the parallel array with reconfigurable bus system are simply acting as switches and not actually accomplishing viable computation. In the dense data case, the parallel array with reconfigurable bus system tends to respond more like the simpler mesh with global buses and thus little is gained from the added complexity. In an effort to optimize the best features of both architectures we present a standard mesh of processing elements with *reconfigurable global buses* added to reduce the cost of communication between processing elements.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.4 Overview of the RISC Architecture

One of the latest innovations in individual processor design is that of the Reduced Instruction Set Computer architecture most often referred to as the RISC architecture. While not all RISC systems are identical, they do tend to share a set of important key characteristics

- A limited instruction set;
- A large number of general-purpose registers;
- An emphasis on optimizing the instruction pipeline[72].

This architecture offers two main advantages (1) improved performance and (2) optimized use of VLSI. Performance is improved through the simplified instruction set which reduces or eliminates the need for microcode and allows more efficient and effective instruction pipelining. A large number of registers allows a significant reduction in memory accesses which provides a corresponding increase in speed.

The optimized use of VLSI relates to the ability to implement the entire processor on one chip. The simplicity of the RISC architecture also makes it

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

easier to implement and optimize the limited resource of chip surface area. Additionally, since only a simple control unit is required and an extensive ROM is not required to store microcode additional space is saved [72].

The RISC architecture is a departure from conventional processor design which tends more towards increasingly complex implementations that are designed to directly handle the constructs of High Level Languages (HLL). The first generation of the RISC work has been experimental in nature, but a number of commercial RISC systems have been produced since the first commercial effort produced the Pyramid [7, 9, 49, 59, 68, 72, 82].

Current technology has RISC architectures on their third generation and in a wide variety of applications. The Motorola 88110 [16, 55], the NVAX [78] and the LR33020 GraphX Processor [77] are examples of the latest applications of RISC architecture theory. These applications utilize integration that places on the order of 500,000 transistors on one chip at a reasonable cost.

The key goal behind RISC architectures has been one of optimizing the performance of operations that the processor spends the most time executing from the view point of common HLLs. Thus a large number of registers are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

provided to significantly reduce operand fetch time and to even store some global variables in the processor. It shall be the assumption that a RISC-type processing element can be constructed to accomplish the computations required later in this work. The potential does exist to contain multiple RISC processors on an single chip as well as the intercommunication network. Similar VLSI technology should be able to produce the bus interconnections in a compact manner.

1.5 Historical Background

The type of interconnection scheme which will be presented here is not entirely new. It was originally employed by Bell Telephone as a non-blocking call switching system in the 1930's. This scheme was known as a crossbar network which interconnected inputs and outputs in a manner that provided a maximum possibility of the call being completed, refer to figure 1.5.

As the number of subscribers for phone service increased, the crossbar interconnection network became infeasible, given 1930's technology. This demand lead to a study of alternate, and smaller, non-blocking switching networks. One of the more in depth examinations of this problem appeared in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

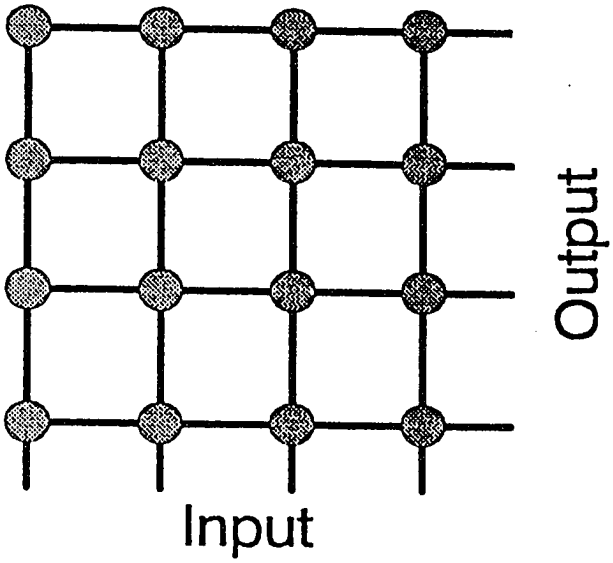

Figure 1.5: The basic Cross Bar Network

the *Bell Systems Technical Journal* by C. Clos [14]. In this study it is shown that as the number of switching stages increases the number of switches required to accomplish non-blocking switching is decreased. This relationship was expressed by Clos as the following :

$$C(s) = 2\sum_{k=2}^{\frac{s+1}{2}} N^{\frac{2k}{s+1}}(2N^{\frac{2}{s+1}} - 1)^{\frac{s+3}{2}-k} + N^{\frac{1}{s+1}}(2N^{\frac{2}{s+1}} - 1)^{\frac{s-1}{2}} \qquad (1.1)$$

where s is the number of stages and N is the number of inputs/outputs. Thus we have :

$$C(1) = N^2$$

$$C(3) = 6N^{\frac{3}{2}} - 3N$$

$$C(5) = 16N^{\frac{4}{3}} - 14N + 3N^{\frac{2}{3}}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$C(7) = 36N^{\frac{5}{4}} - 46N + 20N^{\frac{3}{4}} - 3N^{\frac{1}{2}}$$

$$C(9) = 76N^{\frac{6}{5}} - 130N + 86N^{\frac{4}{5}} - 26N^{\frac{3}{5}} + 3N^{\frac{2}{5}}$$

These equations demonstrate how the system complexity increases with the number of stages. It should also be obvious that for computational applications a multistage communication system (especially of this type) is not practical. Just the cost of routing information in a highly dynamic application could be excessive. The systems discussed above were designed to set up a non-blocked telephone communication, not to handle high speed inter-processor communication where the time to set up the link could far exceed the duration of the communication. In most cases the opposite is true for telephone communications. Thus we shall employ the single stage communications system.

1.6 Previous Work

As noted above, the architecture to be presented here is not entirely new. In addition to the applications to telephone systems, the crossbar communication network has received limited examination as a processor intercommunication system [12, 21, 22]. This prior work focuses on the potential of actual

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

construction of the system vice the specific algorithmic power offered by the use of a crossbar type communication system. The results presented in these papers are especially encouraging, since they establish a firm foundation for a crossbar based multiprocessor architecture.

The model to be presented here considers the crossbar network as a system of reconfigurable global buses that provide a powerful enhancement to a linear array of processing elements. We depart from the models noted above, in that we do not consider message type point to point communication. In the model to be presented, a processing element will place a value at a precise location in the communication network, normally based on it's own ID. Other processing elements will read values from locations in the network. Under this model the sender is not concerned with the receiver(s) and the receiver(s) are not concerned with the sender. This allows the communication system to become a participating portion of many computations. It also removes the need for a message routine system and related costs.

In light of these subtle, but fundamental differences, the crossbar network shall be treated and referred to as a system of *Reconfigurable Global Buses* for the remainder of this work.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

Fundamentals of Arrays with Reconfigurable Global Buses

2.1 Introduction to the Architecture

2.1.1 Motivation

As noted earlier the primary negative aspect of standard linear and 2-dimensional arrays is the loss of performance caused by *long distance* communication. In some applications, these communication problems can be overcome by properly modeling the problem on the architecture. Many highly dynamic problems, however, rely on communication that is strictly input dependent. List ranking is an example of this type of application.

In order to overcome the communication slowdown while keeping the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

number of processing elements linear in the size of the input, we provide a system of controllably interconnected buses. The buses are interconnected by *smart interconnection points.* These interconnection points are *smarter* than a standard interconnection, in that they will interconnect two buses only when directed by the processing element which controls the vertical bus on which the smart interconnection point lies. While this does require the smart interconnection point to contain some logic, it is considerably less complex than a complete processing element. This is considered to be a complexity improvement over the parallel array with reconfigurable bus systems which often uses full processing elements as bus control points. The cost of this complexity reduction appears to be an increase in time complexity from $O(1)$ to $O(\log n)$. The benefit is a reduction in processor requirements on many applications from $O(n^2)$ to $O(n)$. Chin and Lin [12] clearly note that while the communication network requires $O(n^2)$ space, the difference between the space required of a processing element and a switch point is considerable. A processing element requires over a 100,000 transistors, while a switchpoint only requires $O(BW^2)$ transistors where BW is the width of the communication bus. Thus for a 32bit wide bus the switchpoint requires just over 1024 transistors, resulting in a considerable savings in space and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

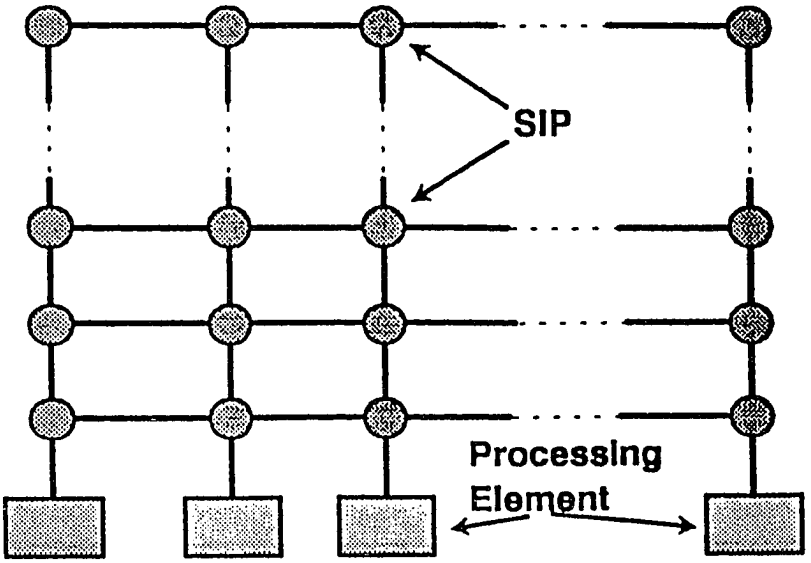

Figure 2.1: The Linear Array with Reconfigurable Global Buses(LARGB)

related costs. Additionally, we have obtained a very flexible architecture, which will require considerable investigation to fully explore its potential.

We shall now more formally define this architecture, and for the sake of clarity, we shall consider a one dimensional array of processing elements instead of the considerably more complex two dimensional model. We begin with a linear array of n processing elements . Each processing element is connected to a vertical bus , thus PE_i connects to VB_i. Each vertical bus has n horizontal buses connected to it by n smart interconnection points . Figure 2.1 is an example of an n processor version of this architecture. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

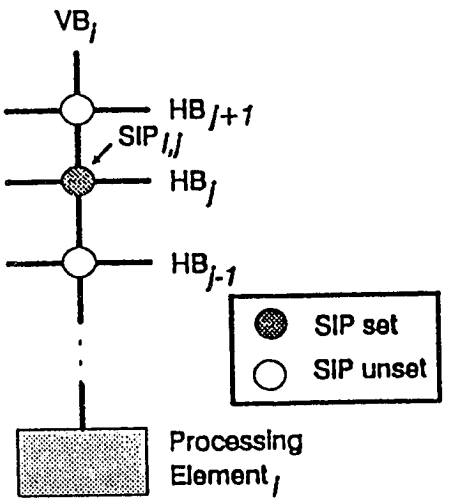

Figure 2.2: A single vertical bus of the LARGB

smart interconnection point at the intersection of VB_i and HB_j $\{(1 \leq i \leq n), (1 \leq j \leq n)\}$, referred to as $SIP_{i,j}$, can only set or break the connection between VB_i and HB_j. This occurs if processing element$_i$ sets the value j on the control bus (see figure 2.2).

2.1.2 Rules of Operation

It is appropriate to now state the rules of operation and restrictions of this hybrid architecture for the sake of consistency and to maintain viability as an implementable system. These basic operating parameters are as follows:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. A processing element can only direct one smart interconnection point on its vertical bus to set the connection to an horizontal bus at a time.

2. Any number of processing elements can connect to a single horizontal bus.

3. Any number of processing elements can read from a single horizontal bus.

4. Only one processing element can be allowed to place a value on a particular horizontal bus at any given time [1].

5. A processing element can accomplish any basic mathematical operation, (+,-,*,/), or logical operation (<, >, =,not,and,or).

We will use these few basic parameters to develop the elementary algorithms which will become the building blocks for applications of this architecture.

[1]This must be prevented at the algorithm level. There is no explicit mechanism to prevent multiple writes at the hardware level.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

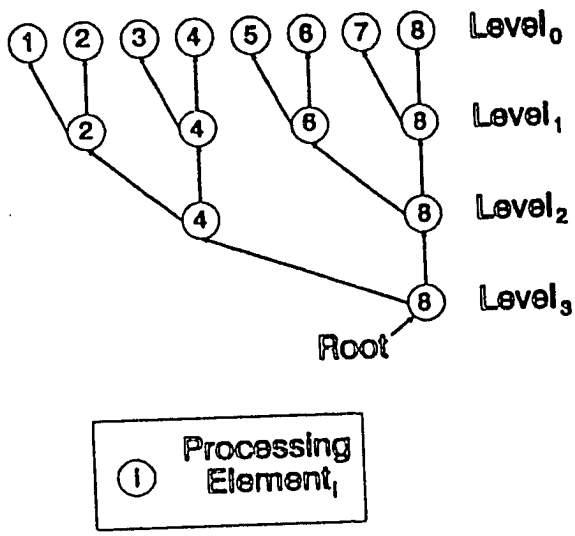

Figure 2.3: Binary Tree Model 1

2.2 Basic Algorithms

Since most prefix computations are easily modeled by a binary tree, we offer two different methods of modeling a binary tree on the linear array with reconfigurable global busses.

Binary Tree Model 1

Given n processing elements, we model a binary tree of depth $\lceil \log n \rceil$ as follows (see figure 2.3):

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- The root is PE_n;
- At all levels above 0, $node_i$ has two children (if they exist);
- The values for the children of $node_i$ at level j are stored as:
  - rightchild is in PE_i;
  - leftchild is in PE_k, where $k = i - 2^{j-1}$.
- $node_i$ communicates with $node_k$ via HB_k (when $i \neq k$);

Since the rightchild of a node exists in the *same* processing element as its parent this model can only be used where the child information is no longer required, once the parent information is found. This is true in a number of problems, including prefix computations.

Binary Tree Model 2

Given n processing elements, we model a tree of depth $\lceil \log n \rceil - 1$ as follows (see figure 2.4)[4]:

- The root is PE_1;
- Level of $node_i = \lfloor \log i \rfloor$;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

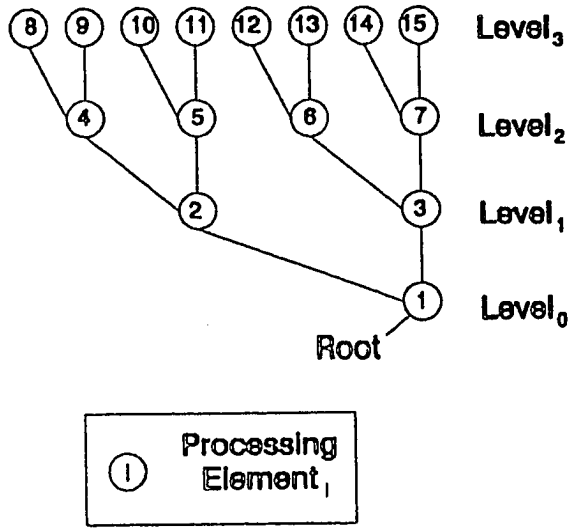

Figure 2.4: Binary Tree Model 2

- The children of node$_i$ are held as follows:
  - leftchild is in PE_k, where $k = i * 2^{level(node_i)-1}$
  - rightchild in PE_{k+1}
- PE_i communicates with PE_k via HB_k $(i \neq k)$.

With these models defined we shall proceed with the prefix computation algorithms for the linear array with reconfigurable global buses. We will use binary tree model 1 in the prefix computation algorithms and binary tree model 2 in the more complex parenthesis matching algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2.1 Prefix Sum

The prefix sum of an array of data, with one value held in each processing element , is one in which once accomplished, the result is such that each processing element now holds the sum of all previous values. More formally, the prefix sum at PE_i when PE_i initially holds x_i is:

$$Psum_i = \sum_{j=1}^{i} x_i \quad (1 \leq i \leq n)$$

The example below demonstrates:

The input of:

3,4,2,8,1,3,2,4,5,10,1,0,2,3,5,7

Upon completion of the prefix sum we have:

3,7,9,17,18,21,23,27,32,42,43,43,45,48,53,60

It should be noted that the last value in the result is also the total sum for the sequence. This is important because the complexity of finding the sum

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of a sequence on a linear array of processors is $O(\log n)$ and the complexity for the prefix sum is also $O(\log n)$. Thus the more powerful prefix sum, can be obtained with no additional complexity. Function prefix_sum provides the details:

Function prefix_sum;
{**Input:** A sequence of values of length n;
Output: The prefix sum of the input sequence in the array Psum.}
begin
for $i \leftarrow 1$ **to** $\log n$ **do**

for $j \leftarrow 1$ **to** n, **step** 2^i, **do in parallel**

$PE_j \leftarrow PE_j + PE_{j-2^{i-1}}$
{ note: this is all that is necessary to compute the sum of the sequence, PE_n will now hold the sum. Additionally each PE_i hold the correct prefix sum values. }

{Let LC_i be the left child of PE_i at the current level, and define RC_i similarly}
for $j \leftarrow \log n$ **down to** 0 **do**

for $i \leftarrow \log n$ **down to** 1, **step** 2^j **do in parallel**

if (PE_i **not** marked) **then begin** {Initially no PE is marked}

PE_i accomplish $RC_i \leftarrow LC_i + RC_i +$ $Value\_Received_i$;
$Value\_Received_i \leftarrow LC_i +$ Value_Received at $Parent_i$;
mark PE_i;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

end;

end.

Theorem 1. *Function prefix_sum correctly computes the prefix sum of n elements in O(log n) time on a linear array with reconfigurable global buses.*

Proof. *Correctness:* The outer "for" loop combines successive sub-totals to form the final total. Thus at the root of the binary tree (see Binary Tree Model 1) is the total and proceeding recursively down the line of left children, for each sub tree rooted at this child the sub-total held at this node is the prefix sum for this node. All that remains is to distribute the information back to all other nodes, so that they each are updated to the correct prefix sum. Since each right child need only add to itself the value of its sibling and the value its parent received on the previous step this is accomplished in the second for loop. The left child Need only add to itself the value that its parent added to it self. It is *trickle down* effect which updates all of the leaves by the time the function halts.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Time Complexity: Both loops which are iterated are of length $\log n$. With O(1) time elements with in the loops, the function halts in $O(\log n)$ time. Thus function prefix_sum correctly computes the prefix sum of the input in $O(\log n)$ time. □

2.2.2 Prefix Maximum

The prefix maximum is a similar operation to the prefix sum, in that the value at a particular processing element is dependent on all nodes which come before it. The example below demonstrates:

The input of:

3,4,2,8,1,3,2,4,5,10,1,0,2,3,5,7

Upon completion of the prefix maximum we have:

3,4,4,8,8,8,8,8,8,10,10,10,10,10,10,10

Just as in prefix_sum the last value is the global result for the entire sequence. Function prefix_max provides the details.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Function prefix_max;
{Input: A sequence of values of length n;
Output: The prefix max of the input sequence in the array Pmax.}
begin
for i ←1 to log n do

    for j ←1 to n, step 2^i, do in parallel

        PE_j ←max(PE_j, PE_{j-2^{i-1}})
        { note: The global maximum of the sequence is now
          held in PE_n. }

for j ←log n down to 0 do

    for i ←n down to 1, step 2^j do in parallel

        if (PE_i not marked) then begin {Initially no PE
          is marked}
            LC_i ←max(LC_i, LC_{i-2^j}); {if LC_{i-2^j} does
              not exist the result is LC_i}
            RC_i ←max(LC_i, RC_i);
            mark PE_i;
            end;

end.
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem 2. *Function prefix_max correctly computes the prefix max of n elements in $O(\log n)$ time on a linear array with reconfigurable global buses.*

Proof. *Correctness:* Just as in prefix sum the root holds the maximum at the end of the first loop, and each left child holds the prefix maximum for its sub-tree.

All that need occur now is for the maxima information to trickle back down to the individual leaves. The second "for" loop accomplishes this task directly. The right child of a given node must contain the maximum between itself and the left child at any level j (refer to figure 2.3). The value of the left child needs to import the prefix maximum from everything which lies further left. This is accomplished by using the value from its parent's left child. This value is directly accessible due to the tree structure, in that each processing element can compute the PEID of a sibling directly. Thus the left child is updated, and then the left child, proceeding down the tree level by level.

Time Complexity: Both iterated "for" loops are of length $\log n$ and contain strictly constant time operations, thus the Function prefix_max halts in $O(\log n)$ time. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The method for finding the prefix minimum is perfectly similar to that for prefix maximum and for this reason will not be detailed here. For later reference the function is given the obvious name: prefix_min.

2.2.3 The Multiple Source Problem

Next we examine a slightly more complex, and communication intensive problem, known as *multiple source*. In the multiple source problem a subset of the processing elements in the linear array are marked. The method of marking is not vital to the procedure, we shall consider each processing element to have a boolean flag known as marked which it can set. The goal of multiple source is for every processing element to know the identity, of the first processing element to its right which is marked. Identity is defined as the processing element identification number or PEID. If the processing element, itself is marked, it maintains its own PEID. The example below demonstrates:

For a set of processors where '*' denotes a marked processor:

0,0,*,0,0,0,0,*,*,0,0,0,*,0,0,*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The result is:

3,3,3,8,8,8,8,8,9,13,13,13,13,16,16,16

The procedure multiple_source, below, details the process.

Procedure multiple_source;
{**Input:** A subset of marked processing elements;
Output: Each processing element knows the value of the first marked processing element to its right. This is stored in the array marked_right where the value of marked_right$_i$ is the first marked element to the right of PE$_i$. If PE$_i$ is marked, marked_right$_i$ is set to i. }
begin
marked_right ←0;
for i ←1 **to** n **do in parallel**

if (PE$_i$ **is marked**) marked_right$_i$ ←i;

i ←1;
repeat

for j ←1 **to** n **do in parallel**

begin
if (marked_right$_j$ ≠ 0 **and** marked_right$_{j-i}$ = 0)
marked_right$_{j-i}$ ←j;
i ←i * 2;
end;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

until (i = $\lceil \log n \rceil$);

return(marked_right);

end.

Theorem 3. *Procedure multiple_source correctly computes the multiple source problem for a subset of marked processing elements from an original set on n processing elements in O($\log n$) time.*

Proof. To show correctness, we assume that some entry in *marked_right* is the wrong value. This can occur in only 2 cases, (1) the value is never altered from the original initialization to zero and (2) a marked processing element which is not closest to the right manages to make the entry in marked_right.

Case 1. Since at each step of marking the number of marked processors can potentially double and $\lceil \log n \rceil$ steps occur all entries will be marked even if only one processing element is initially marked. If multiple processors are marked, then the actual time required to mark all values is $\lceil \log k \rceil$ where k = size of largest unmarked space between two marked processing elements . Therefore all entries in marked_right will receive a value.

Case 2. To address the potential of an entry in marked_right being set by the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

wrong processing element , consider 3 processing elements PE_i, PE_j and PE_k, where $i < j < k$, and let PE_j and PE_k be marked. Assume marked_right$_i$ becomes set to k, but for this to occur marked_right$_i$ would have had to have been zero when a processing element which had the value of k checked. In order for this to happen, a processing element further right than PE_j would have had to check PE_i first, but since all values are decremented by the same value at each step PE_j **must** reach PE_i before PE_k.

The possibility of a PE_j being modified by PE_i, where $i < j$, is zero since the values are always decremented, so in this procedure a processing element never "looks" right.

Thus we have contradicted the assumption that marked_right$_i$ could receive the wrong value, and correctness is shown.

To address time complexity: The one iterated loop (repeat - until) is accomplished exactly $\lceil \log n \rceil$ times. All other steps are accomplished in parallel and thus take O(1) time. The communication method for PE_j to check the value of PE_i is direct. In that first each PE_k, where $1 \leq k \leq n$, puts marked_right$_k$ onto HB_k via VB_k. Now for PE_j to check the value of marked_right$_i$ it need only read the value which is on HB_i, a constant time operation.

Thus the procedure correctly computes the multiple source problem in $O(\log n)$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

time. □

2.2.4 List Ranking

List ranking is the problem of letting each processing element know where it lies in a linked list, when the given input is such that each processing element only knows which processing element is next in the list. A processing element whose next value is zero is considered to be the end of the list. The example below demonstrates:

The input of:

4,7,5,13,0,10,16,15,8,14,9,2,12,3,6,11

Upon completion of the list_rank we have:

15,11,1,14,0,4,10,6,7,3,8,12,13,2,5,9

Many algorithms are improved considerably if each processing element knows where it lies in the structure. This is the purpose of the list_rank

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

procedure shown below.

Procedure list_rank;

{**Input:** Each processing element has the value of the next processing element in the list, for example $next_i$ is the processing element which follows PE_i;

Output: Each processing element knows how far it lies from the end of the list. This is stored in the array *distance* where the value of $distance_i$ is how many processing elements lie between PE_i and the end of the list.}

begin

for $i \leftarrow 1$ **to** n **do in parallel**

begin

if ($next_i \neq 0$) $distance_i \leftarrow -1$

otherwise $distance_i \leftarrow 0$;

$nextnext_i \leftarrow next_i$

end;

for $i \leftarrow 1$ **to** $\lceil \log n \rceil$ **do**

for $j \leftarrow 1$ **to** n **do in parallel**

begin

if ($distance_{nextnext_j} \neq -1$) $distance_j \leftarrow 2^{i-1} + distance_{nextnext_j}$;

$nextnext_j \leftarrow nextnext_{nextnext_j}$;

end;

return (distance);

end.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem 4. *Procedure list_rank correctly computes the list rank for each node in $O(\log n)$ time.*

Proof. This is the standard pointer doubling technique where a pointer is doubled until it finds a PE which "knows" how far it is until the end of the list. Additionally, only processing elements which have not found this value continue to search, this prevents processing elements from trying to read a bus that does not exist.

Formally, let PE_i find a node which has found the end of the list at step k, thus it has found a value in distance which is not -1. Since each processing element doubles the distance it looks at each step, any processing element which found a value in the previous step will be closer to the end of the list than the processing element which is currently reading the value. Therefore, since the processing element has the value of distance which informs it how far the processing element it is currently examining is from the end of the list, and it has counted how many steps (i) it took to find this processing element, the processing element can directly compute its own distance to the end of the list.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To further address time complexity. Since at each step the pointer is doubled, the farthest processing element from the end must find a processing element which knows it's distance to the end in $O(\log n)$ time.
Thus procedure list_rank correctly computes the distance of each processing element from the end of the linked-list structure. □

2.3 Parenthesis Matching

In the preceding section, basic algorithms were provided as building blocks for more complex algorithms. The first of these algorithms to be examined is parenthesis matching. The P_match procedure is given as input a sequence of n parentheses. After processing the algorithm will return, (1) if the sequence is well formed, (2) if so, it will also return for each parenthesis in the sequence the location of its match in the sequence. The examples below demonstrate.

The input of:

(((()())))(()())

Upon completion P_match we have:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The Sequence is well formed

10,9,4,3,6,5,8,7,2,1,16,13,12,15,14,11

The input of:

(((()())(()())

Upon completion P_match we have: *The Sequence is NOT well formed*

Procedure P_match, below, provides the details of parenthesis matching on the linear array with reconfigurable global buses. This procedure uses binary tree model 2.

Procedure P_match;
{**Input:** Each processing element has either a '(' or a ')';
Output: The status of the sequence of parentheses, and if well formed the identification of pairs.}
begin
set all node values ←0;

{Let node$_j$ hold a 5-tuple of (a,b,c,d,e) where
'a' is the number of unmatched left parentheses below node$_j$
'b' is the number of unmatched right parentheses below node$_j$
'c' is the number of matches made by node$_j$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

'd' is the number of matches made at or below LC_j
'e' is the number of matches made at or below RC_j }

(1.)**for** all $node_j$ at $level_0$ **do in parallel**

if $node_j$ holds a left parenthesis **then** node.a $\leftarrow$1;
if $node_j$ holds a right parenthesis **then** node.b $\leftarrow$1;

(2.)**for** $i \leftarrow 1$ **to** $\lceil \log n \rceil$ **do begin**

for all $node_j$ at $level_i$ **do in parallel**

$node_j$.c $\leftarrow$min(LC.a,RC.b);
$node_j$.a $\leftarrow$LC.a + RC.a - $node_j$.c;
$node_j$.b $\leftarrow$LC.b + RC.b - $node_j$.c;
$node_j$.d $\leftarrow$LC.c + LC.d + LC.e;
$node_j$.e $\leftarrow$RC.c + RC.d + RC.e;

end;{for i}

(3.)**if not** (Root.a = root.b = 0) **then begin**

return('*Sequence NOT well formed*');
all stop

end;
else begin

return('*Sequence is well formed*');
Num_Pairs $\leftarrow$Root.c + Root.d + Root.e;
Root_Avail_Range $\leftarrow$[1 ... Num_Pairs]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

end;{else}

{Each node$_j$ shall also hold a 3-tuple of ranges. These will be the range.k, range.L and range.R which is the range kept, range sent to the leftchild and the range sent to the right child respectively. }

(4.)**for** i $\leftarrow\lceil \log n \rceil$ **down to** 1 **do begin**

for all nodes at level$_i$ that made a match **do in parallel**

min_avail$_j$ $\leftarrow$min(range received at node$_j$);
node$_j$.range.k $\leftarrow$[min_avail$_j$ ... node$_j$.c + min_avail -1];
node$_j$.range.L $\leftarrow$[node$_j$.c + min_avail$_j$ ... node$_j$.d + min_avail -1];
node$_j$.range.R $\leftarrow$[node$_j$.d + min_avail$_j$ ... node$_j$.e + min_avail -1];
{Note: consider [0 ... 0] to be a null range}

end;
{All that remains is to distribute the information maintained in the field range.k}
(5.)**if** node$_i$.range.k $\neq$ [0 ... 0] **then begin**

node$_i$ **send** node$_i$.range.k **to** node$_i$.LC **as** made.L;
node$_i$ **send REV**(node$_i$.range.k) **to** node$_i$.RC **as** made.R;

end;
(6.)**for** $i \leftarrow$1 **to** $2n-1$ **do in parallel**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

total$_i$ ←node$_i$.a + node$_i$.b + 2(node$_i$.c + node$_i$.d + node$_i$.e);
free_left$_i$ ←$\frac{total_i}{2}$ - (2node$_i$.d + node$_i$.c);
free_right$_i$ ←$\frac{total_i}{2}$ - (2node$_i$.e + node$_i$.c);

(7.)**for** j ←1 **to** $\lceil \log n \rceil$

for i ←1 **to** $2n-1$ **do in parallel**
{Now each node may receive a range $[f_1 \ldots f_k]$.x, where x = (L,R)
}

if node$_i$ is not a leaf **then begin**
if x = 'R' **then begin**
if free_left$_i \neq 0$ **then begin**
n ←min(free_left$_i$, k);
send $[f_1 \ldots f_n]$ to LC$_i$;
free_left ←free_left -
1;
end;{if free_left}
if free_right$_i \neq 0$ **And** $k - n > 0$ **then begin**
m ←min(free_right$_i$,
$k - n$);
send $[f_{n+1} \ldots f_{n+m}]$
to RC$_i$;
free_right ←free_right
- 1;
end;{if free_right}
else;{x = 'L'}
if free_right$_i \neq 0$ **then** be-
gin

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    n ←min(free_right_i,
                        k);
                    send [f_1 ... f_n] to RC_i;
                    free_right ←free_right
                        - 1;
                end;{if free_right}
            if free_left_i ≠ 0 And k−n >
                0 then begin
                    m ←min(free_left_i, k−
                        n);
                    send [f_{n+1} ... f_{n+m}]
                        to LC_i;
                    free_left ←free_left -
                        1;
                end;{if free_left}
        else {node_i is a leaf}
            stop{node_i}
    end;{if node_i is a leaf}

end.{Procedure P_match}
```

Before we prove the correctness and complexity of procedure P_Match, we make the following observation in the form of lemma 1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lemma 1. *A set of matched parentheses lie as close as possible to the center of the tree rooted at the node at which they were matched.*

Proof. Since binary tree is made up of many sub-trees which are also binary trees, we begin with the matches that occur one level above the leaves, these are adjacent parentheses and since their trees have only two leaves they lie to the center of the tree. Thus we have the basis to prove the general case. Assume this works for level $k-1$ in the tree, we need now show that it works correctly for level k. Some parentheses in the sub-trees rooted at level k will have been matched at lower levels, thus not all of the leaves can be assumed to be available. Assume that one set of parentheses were matched by some node at level k. It is straightforward to observe, however, that these 2 parentheses will originally lie as close to the center of the sub-tree rooted at the node where they were matched as possible. The only parentheses which could lie closer to the center of this sub-tree are those which are matched at levels below k.□

Theorem 5. *Procedure P_Match accomplishes the following in $O(\log n)$ time: (1.) Checks if a sequence of parentheses are well formed, and if they are (2)provides each matching set a unique identification number in the range [1*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

... $\frac{n}{2}$] where n is the number of parentheses input.

Proof First we examine the process of verifying that the sequence is well formed. This occurs in steps 1-3 of procedure P_Match. The loop at step (2.) accomplishes the actual task. Step (1.) is initialization and step (3.) just tests and reports results. Therefore we will examine step (2.) in detail. It proceeds level by level from the leaves to the root. At each step, potential matches are checked. It is obvious that the maximum possible number of matches are made since no more than the minimum of the available left parentheses from the left child and the available right parentheses from the right child can from matched sets at this point. The remainder of the step is straightforward addition of information to be used later. The test at the root remains, the root may hold no unmatched parentheses, since it has no where else to forward them to be matched. Additionally, the root cannot have zero unmatched parentheses when the sequence is not well formed, since the only way a parenthesis is not forwarded is if it is matched with another. Therefore, steps (1-3) correctly determine is a sequence of parentheses are well formed. To address the actual matching of the parentheses, we make use of lemma 1. Before actual pairs can be given identification numbers, each node which

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

actually made a match must receive a *unique* value to assign to that match. The end of step (3.) identified the total number of pairs of parentheses which were matched somewhere in the tree. Now all that need occur is for this information to be disseminated down the tree to each node which made a match. I needs to be shown that every node which made a match will receive a range equal to the number of matches it made along with those its children made. The root starts this with the total number of matches in the tree. Step (4.) further distributes the sub-ranges to the sub-trees in a straightforward manner.

Step (5.) Each node which is the root of a sub-tree now holds a unique range of values which it can assign to the parentheses it matched. These ranges are now broken down into individual ranges for each child. The range sent to the right child is reversed for the reasoning behind lemma 1. Step (6.) just initiates for each node the number of available nodes under each child that need to be matched, by nodes which lie above. Step (7.) continues to pass these values down to the leaves, working from the center out as shown in lemma 1. Thus each parenthesis receives a value that is only held by its match.

To address time complexity: Step (1.) requires O(1) time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step (2.) works level by level, thus $O(\log n)$ time.

Step (3.) requires O(1) time.

Step (4.) passes down the range information level by level, thus $O(\log n)$ time.

Step (5.) requires O(1) time.

Step (6.) requires O(1) time.

Step (7.) pipelines the passing of values down the tree, thus the farthest trip (from the root) requires $O(\log n)$ time.

Therefore the entire procedure directly requires $O(\log n)$ time.

Procedure P_Match correctly matches the parentheses of the input sequence in $O(\log n)$ time, provided the sequence is well formed, otherwise it returns that the sequence is not well formed in $O(\log n)$. □

Procedure P_check, below, offers an alternate method for checking whether or not a sequence of parentheses are well formed or not. It does not provide a mechanism for matching the set of parentheses.

Procedure P_check;
{**Input:** Each processing element has either a +1 or a -1 signifying a '(' or ')' respectively;
Output: The status of the sequence of parentheses, and if well formed the identification of pairs.}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

begin
Psum ←prefix_sum(input); {Prefix sum the input}
marked ←0;
if (Psum$_n \neq 0$) PE$_n$ set marked$_n$ ←1;
for i ←1 **to** $n-1$ **do in parallel**

    if (Psum$_i < 0$) PE$_i$ set marked$_n$ ←1;

j ←sum(marked); {The function sum simply uses the first loop of the prefix sum procedure}
if ($j > 0$) **then**

    return(*The sequence is NOT well formed.*);
    stop.

return(*The sequence is well formed.*);

end

Theorem 6. *Procedure P_Check correctly checks if a sequence of parentheses are well formed in O(*log n*) time.*

Proof. Procedure P_Check is based on the observation that the prefix sum of the (+1,-1) sequence corresponding to a sequence of ['(',')'] must have a final sum of 0, and at no point may it be negative. Directly, if the final

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sum is non-zero, at least one parenthesis is left unmatched by the end of the sequence. With in the sequence, the prefix sum may never become negative, since for this to occur more ')' will have occurred than '('. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

Graph Theory

3.1 Introduction

First we shall formally define a graph. A graph consists of a finite set V and an irreflexive binary relation on V. V is referred to as the vertex set of the graph. The binary relation is most often referred to as a collection, E, of *ordered* pairs. Adj(v) is referred to as the *adjacent set* of the vertex v, and the ordered pair $(v, w) \in E$ is called an edge. Directly,

$$(v, w) \in E \qquad \text{if and only if} \qquad w \in \text{Adj}(v).$$

The assumption of irreflexivity implies:

$$(v, v) \notin E \text{ for } (v \in V).$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Additionally the *Neighborhood* of v is defined as:

$$N(v) = \{v\} + \mathrm{Adj}(v).$$

It should be noted, however, that while loop edges will not be permitted, edge pointers, in the algorithmic sense, will be allowed to point to the vertex from which they originate. In this presentation, an edge will often be denoted without the parentheses and comma. Therefore:

$$vw \in E \qquad \text{and} \qquad (v, w) \in E$$

have the same meaning [24].

We intend to focus on very large sparse graphs. This type of graph may be found in numerical applications such as finite element analysis[66]. This type of graph is especially difficult to handle due primarily to its size, but also the scarcity makes efficient assignment of processing power to the problem more complex. The goal is to use the dynamic power of the linear array with reconfigurable global buses to evaluate the graph and then reconfigure resources to suit the problem with no loss in efficiency.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Graphs are computationally represented by a small group of standard structures, adjacency matrix, adjacency list and a set of edges. While all three of the structures contain the same information, it is stored and accordingly accessed in very different manners. The adjacency matrix has a non-zero entry at location i, j of the matrix if and only if the edge ij is in the graph. The entry may be just representing the existence of the edge or it may also have weight information. It is obvious that for very large sparse graphs, this storage method would be very inefficient since most entries would be simply zero. The adjacency list only maintains entries for the edges which do exist. The semi-structured nature of the adjacency list, however, makes it difficult to map to any static multiprocessor system. The edge set of a graph maintains all of the information of a graph in an manner that is clearly mappable to a multiprocessor system for computation. Thus, we assume that the graph to be processed is initially input as a set of $2m$ directed edges.

3.2 Connected Components

The connected component problem is answering the question, *Are V_i and V_j in the same component of the graph?* This implies the existence of a path

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

between the vertices and accordingly some type of relationship if the graph is resultant from real physical data. Additionally, solution to the connected component allows redistribution of computational resources to increase the efficiency of computation. This redistribution is especially important if $P << 2m$, where we want to store related vertices in the same or adjacent processors which helps to speed more complex computations.

The spanning tree/forest is closely related to the connected component problem, in that if a spanning forest is available for a graph, each tree in the forest represents a connected component. The spanning forest maintains more structure information than the connected component model. An edge is in the forest only if there is a corresponding edge in the graph. It is also shown earlier that the linear array with reconfigurable global buses can very efficiently model and process tree structure, therefore, we generate a spanning forest from the result of the connected component problem.

Shiloach and Vishkin offer an optimal parallel connected component algorithm for the parallel random access memory (PRAM) model of computation in [70]. This algorithm requires $O(\log n)$ time and $n + 2m$ processors. We take inspiration from this algorithm and generalize it for an LARGB with P

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

processors.

We begin with the $2m$ edges input $\frac{2m}{P}$ edges per processing element. First each processing element conducts a breadth first search on the edges it stores locally. The result being a spanning forest for those edges [3].

Before proceeding we will briefly review the functions used to complete the connected component processing. For further details of these functions the reader is referred to [70].

Conditional Hooking: if the parent of i is a root, j is adjacent to i in the original graph, and parent[j] < parent[i] then hook j to i.

Star Hooking: If i belongs to a star, j is adjacent to i in the original graph, and j is not in i's star then hook j to i.

Shortcutting: if i is not a root, parent[i]=parent[parent[i]].

These three functions are the basis of Shiloach and Vishkin's algorithm.

We employ the hooking idea to construct increasingly larger connected components or trees, but we do not bother to employ shortcutting. Once we have a spanning forest where each vertex knows the value of the root of its tree, the connected component query can be resolved in O(1) time. Thus we have the following algorithm:

Procedure Complete Spanning Forest
{input: The spanning forest from the BFS conducted by each PE on its input

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

edge set.
output: A spanning forest for the original graph where each vertex knows the value of the root of it's spanning tree in the forest.}
begin
repeat

{ Hooking}
if j is a root and root[i] $< j$ and $ij \in E$ **then begin**

hook j to i;
update root[descendant of j] = root[i]; **end**

{Check if root is hookable}
if i is root and not ($\exists\ j \in V$ such that $ij \in E$ and root[j] $\neq$ i) then shift root of the tree to a vertex with a link outside of the tree;

until(Check root step did not produce any changes)

end{Procedure Complete Spanning Forest}

Theorem 7. *Procedure Complete Spanning Forest correctly generates a spanning forest from the BFS input in* $O(\frac{m}{p}\log n)$ *time.*

Proof. Assuming that the BFS step found nothing but individual edges, thus the input is exactly that of the algorithm assumed by Shiloach and Vishkin. Any more structure obviously makes the task easier. Thus at each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

step a tree is joined to another tree at least doubling the size of the smaller tree. Cycles are not induced because the larger indexed root always hooks to the tree of the smaller indexed root. The process of shifting the root is straightforward since all nodes know the value of their root at all times. Reversing pointers on the LARGB to maintain a parent pointer tree is also a direct operation.

The time complexity is directly $O(\frac{m}{P} \log n)$ since each hooking at least doubles the size of the smaller tree. The $\frac{m}{P}$ factor is directly derived from the distribution of multiple edges to a single processing element. □

Corollary 7.1 *The entire Spanning Forest construction is accomplished in* $O(\frac{m}{P} + \frac{m}{P} \log n)$ *time.*

The $\frac{m}{P}$ factor is the cost of the initial BFS and the second is shown in Theorem 1. □

It should be noted that this would be the result predicted by Ahmdal's law [58] except for the BFS cost. Since the input set is expected to be derived from some actual structure or physical data set, the BFS step should save

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

iterations of the second procedure, thus reducing the constant factor in the total complexity. It is recognized that the BFS cost is not actually significant in light if the strict definition of O(f(n)). Thus the $O(\frac{m}{P}\log n)$ performance is as expected and is optimal.

3.3 Introduction to Least Common Ancestors

The LCA problem is: given a collection of rooted trees, answer a query of the type *what is the lowest common ancestor of the nodes i and j?*[2] This shall be denoted as LCA(i, j) There are many versions of the problem, We follow from [28] and examine five versions of the problem. In each version, the LARGB solution has equal or better performance than those presented in [28]. The versions are:

1. The collection of trees are static and the entire sequence of queries is specified in advance.

2. The collection of trees is static but the queries are given on-line.

3. The queries are given on-line. Interspersed with the queries are on-line commands *link*(i, j) where i and j are tree roots. The effect of*link*(i, j)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is to combine the trees rooted at i and j by making j a child of i.

4. The queries are on line. Interspersed with the queries are on-line commands *link*(i, j) such that i need not be a root.

5. The queries are on-line. Interspersed with the queries are commands of two types *link*(i, j) and *cut*(i). The command *cut*(i) has the effect of causing node i to delete its parent pointer and become a root.

Proposition 1.[28] *Let T be a complete binary tree with n leaves. Any pointer machine requires $\Omega(\log \log n)$ time to answer any LCA query in the worst case, independent of the representation of the tree.*

This proposition is of interest because the LARGB is representing the tree with parent pointers, just as the pointer machine would. Thus this is a viable lower bound. Additionally, the lower bound for the *link* and *cut* operations was left open in [28]. It is unknown if these bounds have been proven, they will be proven here for the linear array with reconfigurable global buses.

The trees will be initially input as nodes and pointers, each node will be held by a processing element, that processing element will only "know" its

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

parent. Nodes with no parent are by default roots.

3.4 The LCA Problem on Static Trees

First we examine the single query and show how the LARGB easily obtains the lower bound given in Proposition 1. To answer LCA(i, j) the two nodes i, j start a traversal to the root of their tree marking the path as they travel. The first node at which the two paths coincide is naturally the LCA, if both reach the root without finding the other's path, i and j are in separate trees and have no LCA. The traversal is accomplished by the pointer doubling technique which doubles the length of the step at each stage. Thus any node will reach the root in $O(\log \log n)$ stages. Each stage entails visiting an ancestor parent, and examining it to see if it was already visited by the other node. It is also straightforward to observe that even if the two traversals are accomplished in series, the running time will still be $O(\log \log n)$.

For the case of an extremely deep tree, where many parents have only a single child, it is observed that the above method yields a result in $O(\log n)$ time. The closer the tree comes to being a linked list the larger the constant factor for the time will be.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Even with a balanced binary tree the $O(\log \log n)$ running time could become too much if many queries need to be answered. If the tree structure is regular and the relationship between the assignment of processing elements to this structure is known, then the solution of $LCA(i, j)$ is direct. This cannot be expected to be the case. We assume the input to be the nodes in any order, with any node only knowing its parent.

In order to efficiently answer queries each node needs some method of knowing what nodes are its children. We could have each processing element store the PEID for each of its children, but this is redundant and wasteful. The most efficient method maintaining this information would be to allow each processing element to maintain a range, $[a \ldots b]$, where a is the number of the first of its children and b is its last child. Since the order is unknown the PEID will not suffice for the numbers. Thus we must find a method of assigning an additional value to each node so that each parent can hold the range $[a \ldots b]$.

To accomplish this we employ the *Euler Tour Technique* presented by Tarjan and Vishkin in [76] where they use the technique to compute the preorder and post order numbering of the vertices. We are interested in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

preorder numbering of the vertices. The *Euler Tour* requires O(n) processors and O($\log n$) time on an EREW pram. In section 4.3 it is shown that the LARGB can model the CREW pram in equal space and time, so accomplishing the *Euler Tour* is direct. We allow an extra 2m processors in this step to represent each tree link as 2 links, one up and one down the tree. We still employ only O(n) processors since in any tree $m = n - 1$.

Upon completion of the *Euler Tour* each node now has a value assigned. These values correspond to the preorder numbering of the tree. Thus, if node$_i$ has the value k all of its children have values greater than k, therefore $a = k + 1$ for node$_i$. To find the value of b each node need only find the value of the largest child. Due to the numbering, it is straightforward that this node must be a leaf, so each leaf need only pass its value to its parent and its parent only forwards the largest value. As it turns out, with little modification, the *Euler Tour* will deliver this value directly, all that need be done is for each up link that is traversed to leave the current number at the parent.

This method accounts for the LCA only within a tree, for multiple trees we assign each root a unique value. A prefix sum over the b value held at

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the root nodes gives each tree a unique value, and an increment factor for giving each node a unique value in the forest. Each root just broadcasts this value to each node where the increment is added to the preorder number it already holds.

Now it is almost a direct lookup to determine LCA(i, j). The method is as follows:

1. i is broadcast to all processing elements. The processing element which holds node i places its preorder number onto HB_1 to inform all processing elements.

2. the same occurs for j.

3. all processing elements for which $a \leq i \leq b$ and $a \leq j \leq b$ are true mark themselves.

4. all unmarked processing elements examine their parent, if their parent is marked, place their parent's node number on HB_1. This is the LCA(i, j).

5. if no processing elements are marked, then i and j are in different trees.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Thus once the preprocessing is accomplished, we have a very straightforward LCA solution method for many queries. Additionally with this method, it is not important if the queries are known in advance or not.

3.5 The LCA Problem with Dynamic Trees

The *link* and *cut* commands simply require some pointer reassignment and renumbering of nodes. The link operation will be considered in its general case, the linking of two roots is a special case and less difficult than the linking of a root to any other node in the tree. We assume multiple queries are to be solved also. *Links* could be accomplished an then simply reapply the preprocessing step given above, but this would be costly as the majority of the structure of the trees did not change. We need an efficient method of updating the preorder numbering of the nodes in which it changes. To do this we maintain a couple of pieces of information that were available in the preprocessing step. Each root maintains the PEID of the root to its left in the prefix sum computation, and each node maintains the PEID of its own root.

If a root is *linked* it is removed from the list of roots and all other roots

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

compensate for the loss. There are two cases for this compensation:

(i)The root is *linked* to a tree which lies to the left of its former position in the root list, this causes each root that is newly to its right to increment all of its node prenumbers by the size of the tree moved.

(ii)The root is *linked* to a tree which lies right of its former position, this requires that all trees which are newly to the left of the tree decrement all preorder numbers by the size of the tree.

This causes all other trees to be updated to maintain unique values throughout the forest. All that remains is to update the tree to which the *link* occurred. For the purpose of the numbering we assume that the *linked* node becomes the right most child of the parent to which it was *linked.* The b value for the parent of the *link* and all values greater than b in the tree must be incremented by the size of the tree *linked.* This may be accomplished in O(1) time by broadcast from the root. The nodes of the *linked* tree are all that remains to be updated. This is accomplished by adding its new parent's a value less the old root's a. With this complete the nodes can be informed of their new root by the old root through a simple broadcast. The old root sets its parent pointer to its new parent and becomes part of the tree.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This straightforward example demonstrates the following obvious result:

Theorem 8. *The linking operation has a lower bound of* $\Omega(1)$ *for a forest of trees that has been preprocessed.* □

The *cutting* operation is perfectly similar to the *linking* operation, and thus only the specific differences will be noted. The major difference is how a new tree is handled once it is cut out of an existing one. If the new root is added to the list of roots directly to the right of the its former root, none of the other trees in the list need to do any updating. Placing it somewhere else would require updating. All that remains is to update the tree which lost a subtree and the new tree itself all nodes in the old tree whose a is greater than the b value of the tree which was removed must decrement their a by the former section's $b-a$. The new tree renumbers its nodes by incrementing all of its nodes by the difference between its b and the a of next root in the list less 1. Again it is a constant time operation and we have the following result:

Theorem 9. *The cutting operation has a lower bound of* $\Omega(1)$ *for a forest*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of trees that has been preprocessed. □

3.6 Further Graph Algorithms

Once the connected components of a graph are known, each component may be examined independently of all others. This is especially important for very large sparse graphs since a natural partition for the problem is generated by the various components. This partitioning method does have the drawback that the component size and distribution cannot be know *a priori*. It is extreme flexibility in communication of the linear array with reconfigurable global buses that makes the efficient algorithms presented here possible. The combination of the least common ancestor and connected component algorithms form the basis of the algorithms to find additional graph properties.

3.6.1 The Bipartiteness of a Graph

A bipartite graph is defined as a graph in which the vertices of the graph may be divided into two sets, X, Y, where all edges of the graph exist only between X and Y refer to figure 3.1. No vertex in X is adjacent to any

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

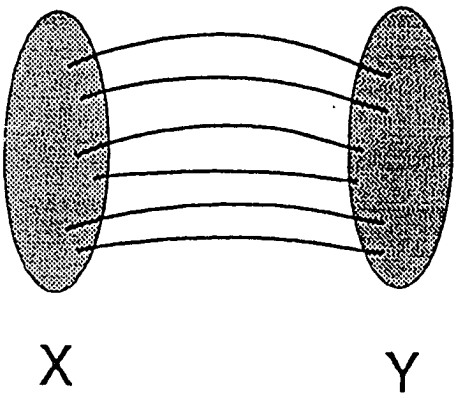

Figure 3.1: The Bipartite Partition of a Graph

other vertex in X and similarly for Y [24].

The spanning forest input form for the graph allows a straightforward solution to the bipartiteness question. For clarity the processing of one tree is shown, the remainder are perfectly similar. First, the graph is partitioned into the X and Y subsets by assigning alternating levels of the tree to set X and Y respectively. All that remains if for each vertex to verify that all of its neighbors lie in the opposite partition. The check in X is accomplished by assigning each vertex in X a single bit value of '1' and each vertex in Y a bit value of '0'. Each vertex in X now need only assemble the bits from its neighbors into an integer and check if the integer is greater than zero the graph is not bipartite. The process is perfectly similar for the Y partition.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem 10. *The bipartiteness of a connected component can be determined in* $O(\frac{M_i}{P_i})$ *time given that there are* M_i *edges lying in the connected component*$_i$*, but not in the spanning tree and* P_i *processing elements to accomplish the checks.* □

3.6.2 Bridges in a Connected Component

A *bridge* is an edge in a connected component of a graph that causes the component to become disconnected if removed. Again, the spanning tree for each connected component is of primary interest. The bridges with in a connected component are found as follows:

Procedure bridges;
{input: a connected component, $C_i(G)$, of a graph and its spanning tree, ST_i
output: all edges which are bridges in the input component}
begin $G'_i \leftarrow C_i(G)$ - ST_i; {Find edges not in spanning tree}
$SF'_i \leftarrow$Spanning forest[G'_i];
if SF'_i is single tree
then no bridges exist in $C_i(G)$
else

> **if** v_j is a root in SF'_i and not a root in ST_i
> **then** the edge (v_j,parent(v_j)) is a bridge in $C_i(G)$; {parent function is relative to ST_i}

end.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem 11. *Procedure bridges correctly determines the bridges within a connected component in* $O(\frac{M_i}{P_i} \log n)$ *time.* □

Corollary 11.1. *Any component that generates a second single spanning tree is a* **biconnected component.** □

Thus procedure bridges not only identifies bridges, but also biconnected components if they exist.

3.6.3 Fundamental Cycles in a Connected Component

A fundamental cycle is the cycle formed by adding one additional edge to the spanning tree of a graph. All fundamental cycles are found by adding and then removing edges from the spanning tree. The procedure presented here employs the optimal least common ancestor procedure found above. This procedure allows direct solution to each the cycle caused by the addition of a edge to the spanning tree. Each end of the added edge need only find their least common ancestor and thus the path of the cycle is known. More formally the procedure for a single edge is as follows:

Procedure Fundamental Cycle;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

{input: the spanning tree, ST, of a graph and an edge, uv, to be added to the tree.

output: the cycle, FC, in the tree caused by the edge addition.}

begin

$w \leftarrow$LCA(u, v); {each vertex along the path u to w and v to w need only be marked that it is on that path.}

FC $\leftarrow \{u \ldots w\} + \{w \ldots v\}$

end

Theorem 12. *Procedure Fundamental Cycle correctly determines a fundamental cycle in a connected component in* $O(\frac{N_i}{P_i} \log N_i)$ *time.*

Proof The correctness is straightforward.

Complexity Analysis: given P_i processing elements to examine a spanning tree of size N_i the time required to find a single fundamental cycle is $O(\frac{N_i}{P_i} \log N_i)$.

□

Corollary 12.1. *The fundamental cycles generated by the* M_i *edges in the component but not in the spanning tree, may be found in* $O(\frac{M_i N_i}{P_i} \log N_i)$ *time.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

Other Applications of the linear array with reconfigurable global buses

4.1 Low Level Image Processing

The area and perimeter of a component of a binary image are fundamental problems in low level image processing. Labeling multiple connected components in a binary image is a fundamental mid-level image processing task. We address all of these tasks on the Linear Array with Reconfigurable Global Buses and achieve $O(\log n)$ time algorithms. The $O(\log n)$ time algorithm for component labeling matches the performance of the fastest known algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.1.1 Introduction

Tasks in computer vision may be separated into three categories, those considered low-level operate directly on the pixels of an image and those considered high-level operate on structures with in the image. The mid-level tasks are those which bridge the low-level to high-level range. Their main task is to identify the symbols in the pixels which the high-level task can then use as input.

The determination of the area and perimeter of a binary figure are primary low-level tasks. These operations provide very basic statistical information about the image at hand [53, 65].

The identification and labeling of components in a binary image is a fundamental midlevel task commonly referred to as *Component Labeling*. The component labeling problem has received considerable study on both sequential and parallel architectures [8, 18, 26, 27, 37, 52, 61, 63, 65].

Due to the ease of mapping a two dimensional image onto a two dimensional mesh it is obvious why the mesh connected topology is often applied to image computations. The regular physical structure of the mesh also causes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

it to be particularly well suited for VLSI implementation. See [29, 73] for an overview of parallel architectures and implementations.

While simple meshes easily map binary images, they have a large communication diameter that tends to slow computation that requires communication across the mesh. In order to overcome this communication slowdown we employ the linear array with reconfigurable global buses.

4.1.2 Measuring the Fundamental Properties of an Image

Before we can actually process an image, the method by which an image is mapped onto the linear array must be fixed. In most high performance image processing applications this is not necessary since the square mesh architecture which is most often applied to image processing directly maps at least a portion of the image. Therefore, the input will be an $n \times n$ pixel image which is to be mapped to n^2 processing elements of the linear array. The image is mapped in the obvious manner, the first row of pixels from the input will be held in the first n processing elements, the second row of input is held in the second n processing elements, etc. Thus, the i^{th} row of input will begin at $\mathrm{PE}_{i \times n+1}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Area and Perimeter of a Component

We begin with the elementary image properties, area and perimeter. Additionally we assume a single component in the current image. In a binary image the area is a straightforward sum of the '1' bits in the figure. This sum may be accomplished by the standard prefix sum method on a linear array with reconfigurable global buses .

To find the perimeter[1] of the figure, the processing elements which hold perimeter pixels must be identified and then summed as in the area computation. In order for a PE_i to identify if it is on the perimeter of the figure, it must first hold a '1' and then check each of its neighbors to find out if any hold a '0'. Due to the mapping method the left and right neighbors of PE_i are PE_{i-1} and PE_{i+1} respectively, except in the case where PE_i lies on the edge of the image. In this case, one of the neighbors will not exist. Since this boundary condition can be found in $O(\log n)$ time, the computation of the perimeter will not be effected by this one-time computation.

The neighbors above and below PE_i are found in a similar manner, except

[1]This perimeter is a relative value, related to the image resolution which provides a useful measure, but it is not an exact measure of the object's true physical perimeter

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that they are PE_{i+n} and PE_{i-n}. Here the boundary conditions are $[0,n^2]$ in that any processing element in the first row will try to read a PE_j where $j < 0$ and any processing element in the last row will try to read a PE_j where $j > n^2$. This can be easily prevented and adds no complexity.

Now, with the perimeter processing elements identified, all that remains is to count these identified processing elements . Which is simply a matter of marking all perimeter processing elements with a '1', non-perimeter processing elements with a '0' and accomplishing the sum along the linear array as in the area computation.

Given that the prefix sum can be accomplished in $O(\log n)$ time on n processing elements, it is obvious that both the area and perimeter of a figure can be found in $O(\log n)$ time. This yields the following result.

Theorem 13. The area and perimeter of an $n \times n$ image can be found in $O(\log n)$ time on an n^2 processing element LARGB. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

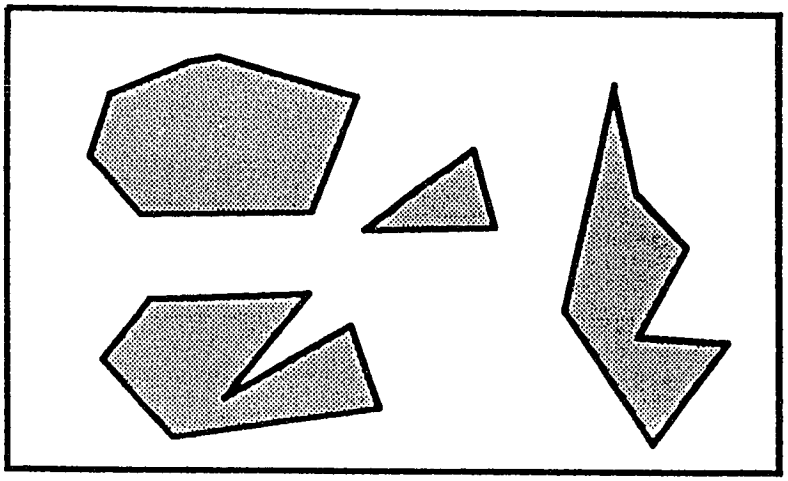

Figure 4.1: Multiple Components of an Image

4.1.3 Component Labeling

The goal of component labeling is, given a binary image with multiple closed shapes, or components (see figure 4.1), assign to each component a unique identification label. The following steps describe the general component labeling process for k distinct components:

1. Identify which processing elements hold a perimeter pixel;

2. Form a linked list of processing elements clockwise around each component (forming k independent linked lists);

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. For each list, identify the processing element with the lowest processing element identification (PEID) number, and inform each processing element in that list;

4. Inform each processing element with in each component of the value found in step 3;

5. Mark the k minimum processing elements with a '1' and all others with a '0', prefix sum these values.

6. Each minimum processing element broadcast the value it received from the prefix sum to each processing element with in its component.

Theorem 14. *The LARGB Component Labeling algorithm assigns a unique value in the range* $[1 \ldots k]$ *to each of* k *solid components of the input image in* $O(\log n)$ *time.*

Proof. Correctness: The identification of perimeter processing elements for multiple components is the same as in Theorem 13 for a single component. The remainder of the algorithm is based on the ability to form multiple linked lists via the reconfigurable global buses of the LARGB. PE_i identifies PE_{i+1} in the linked list by checking each neighbor to identify which processing ele-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ments are adjacent on the perimeter of the figure. The location of the body of the component allows each processing element to determine which perimeter neighbor is next in the clockwise direction around the figure.

The minimum processing element in each linked list is propagated through the list starting from every processing element which is a potential minimum. A potential minimum is any processing element which has a lower PEID than the processing elements on either side of it on the perimeter. Multiple candidates are resolved by any processing element which receives multiple values, just retaining and forwarding the minimum value. It should be noted that at any single step a processing element will only receive one value.

With the minimums identified, the prefix sum method of obtaining unique component identification numbers is straightforward. All that remains is to accomplish the broadcast of the component identification number to each processing element within each component. Since each processing element "knows" the value of the minimum processing element in its component, the minimum processing elements place the component identification number on to the horizontal bus which matches its own PEID. This allows all other processing elements to concurrently read the component identification number

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for their particular component.

Time Complexity: The time required to accomplish each step is as follows:

1. All neighbors of a processing element can be examined in O(1) time;
2. Same as in step [1], O(1) time;
3. Using the technique of pointer doubling to traverse the linked lists, this is accomplished in $O(\log n)$ time;
4. May be accomplished by a local application of multiple source in $O(\log n)$ time;
5. Prefix sum is $O(\log n)$ time];
6. The values may be read in O(1) time.

Thus, the algorithm requires $O(\log n)$ time to identify and label multiple components in a binary image. □

This allows components to be efficiently identified, allowing for efficient further processing if necessary.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

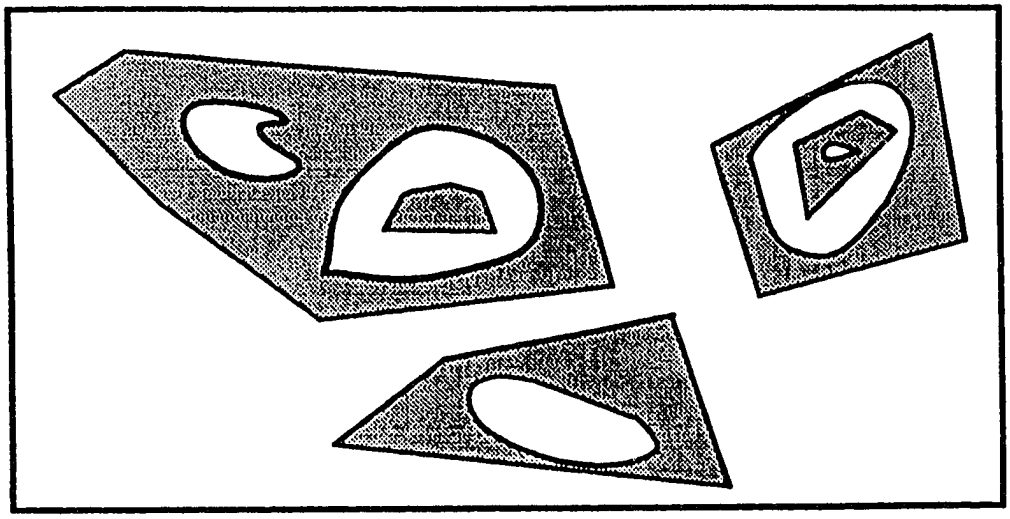

Figure 4.2: Multiple Components of an Image with Holes

The limitation of this algorithm is that it cannot handle components with "holes" (see figure 4.2). This limitation is derived from the multiple source method used to inform each processing element of its identification number. Additional work needs to be done for components with "holes." This work is similar to the process used by Olariu et. al. [52] in their component labeling algorithm for the parallel array with reconfigurable buses. Informally, each minimum processing element, as identified in the previous algorithm, searches to its left until it finds one of the following boundaries:

1. The grid edge;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. A hole perimeter processing element;

3. An exterior perimeter processing element.

Using this information an exterior perimeter processing element can tell if it is with in a "hole" or not. Additionally, each hole perimeter which finds a hole can link to the minimum processing element for that hole, since each processing element on the perimeter knows the minimum processing element in its own list. Furthermore, this now connects all holes in a specific component to the component perimeter. Now using this connection all processing elements around all holes can be directly informed of the component identification number when it is found. Thus the component labeling algorithm for k components with l holes is as follows:

1. Identify which processing elements hold a perimeter pixel;

2. Form a linked list of processing elements clockwise around each perimeter (forming $k + l$ independent linked lists);

3. For each list, identify the processing element with the lowest (PEID) number;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Inform each processing element on the perimeter of the value found in step 3;

5. Each processing element identified in step 3, search to the left until it finds a boundary;

6. Each processing element which is a hole boundary and finds a hole boundary, form a secondary link to the minimum processing element on that boundary;

7. For each list of secondary links, propagate the PEID for the minimum processing element on the exterior boundary to which the last processing element in the list connects;

8. Each hole boundary processing element broadcast into the component the value of the minimum processing element on the exterior boundary, additionally all exterior boundary processing elements broadcast into the component this same value;

9. Mark the k minimum processing elements with a '1' and all others with a '0', prefix sum these values;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

10. Each minimum processing element broadcast the value it received from the prefix sum to each processing element with in its component.

Theorem 15. *The LARGB Component Labeling algorithm assigns a unique value in the range* $[1 \ldots k]$ *to each of* k *components of the input image in* $O(\log n)$ *time.*

Proof. Correctness: Since the majority of the correctness is addressed in the proof of Theorem 13, we will only address the hole handling steps here. In most cases, any minimum processing element can tell if it is on a hole or a component by examining the processing element directly below it, if this processing element holds a '0' then it is on a hole otherwise it is an exterior boundary. This test will fail, if:

the minimum PE falls on the perimeter path list twice, it will detect a hole below it, but actually be a component boundry or vice versa.

the minimum PE lies at the extreme end of a component a similar case will occur.

The checks for both cases are straightforward and add no additional complexity. As already noted, a minimum processing element searching the pixels to its left can only find one of the following three cases:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. The grid edge;

2. A hole perimeter processing element;

3. An exterior perimeter processing element.

Items 2 and 3 are straightforward, and item 1 is any processing element whose PEID is a multiple of $\sqrt{n} + 1$. Those which find the component exterior boundary have the value of the minimum processing element on the boundary directly available. Those which find other holes, however, must form the secondary linked list which at the end does connect to the boundary. Through the same technique of pointer doubling used above, each processing element in this secondary list can obtain the value of the minimum processing element on the exterior boundary. Once the minimum processing element on each hole perimeter has the value of the exterior minimum processing element it can inform all of its perimeter. Thus no matter the complexity of the system of holes within a component, every processing element with in the component can be directly contacted by some perimeter processing element which knows the value of the minimum processing element in that component. Components with in holes are perfectly similar.

Time Complexity: Since the additional communication depends on pointer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

doubling and then constant time reads, the algorithm still requires $O(\log n)$ time to identify and label components with holes in a binary image. □

4.2 Hough Transform

4.2.1 Introduction

One of the fundamental problems in image processing is the detection of shape. A key subproblem in shape detection is the identification of lines and curves. The Hough Transform is often employed in the detection of lines [31, 38, 62, 65]. In short, the Hough Transform involves the converting of lines in a two-dimensional image space into points in a certain parameter space. Thus, a straight line L can be uniquely represented by two parameters θ and ρ, where θ is an angle determined by the normal to the line L and the positive direction of the x axis, and ρ is the signed distance from the origin to the line L. The θ-space is divided into k_θ sections representing the *angular resolution* of the θ-space.

For the purpose of computing the Hough Transform an $n \times n$ binary image is assumed. Initially, n^2 processing elements are assumed with the image mapped one pixel per processor. Recently, a number of Hough Trans-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

form algorithms have been shown for various parallel architectures [25, 57, 61, 63, 62]. Of these architectures, the mesh has emerged as a natural choice for image computations. An image naturally maps to the processor array, but due to the irregularity of images processor communication between non-neighboring processors is often required. This potentially large communication diameter is a primary reason for the addition of bus systems to meshes. The further addition of reconfigurable bus systems helps to overcome the irregular communication patterns of images.

With sufficient reconfigurability, the actual arrangement of the processing elements becomes less important. Thus we shall show that the linear array with reconfigurable global buses can offer fast efficient solution to the Hough Transform.

4.2.2 Basics of the Hough Transform

We begin by reviewing the basics of the Hough Transform. The reader is referred to [31, 38, 65] for further details. Let L be a straight line in the plane. Directly, L can be represented by the following two parameters (refer to figure 4.3):

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

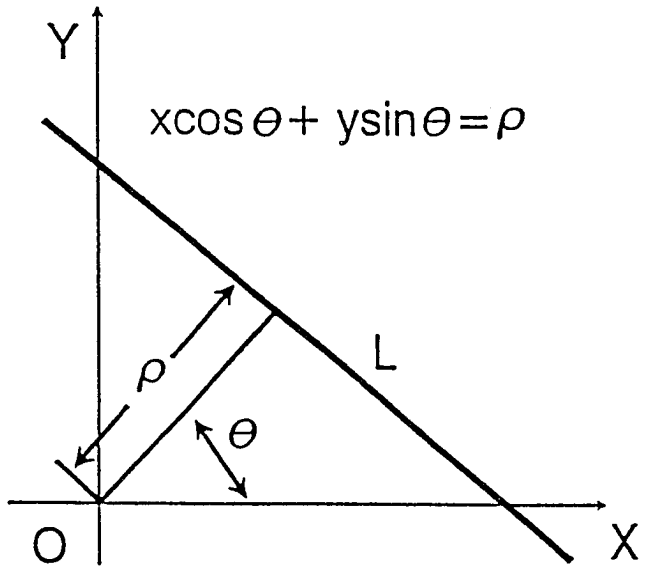

Figure 4.3: A line L in the Normal form.

θ_L the angle determined by the normal to L and the positive direction of the x axis;

ρ_L the signed distance from the origin of the Cartesian coordinate system to the line L.

When no loss of clarity is possible, we simplify the notation by writing θ and ρ. It is obvious that if θ is restricted to the range $[0 \ldots \pi]$, the ordered pair uniquely determine the line L. Additionally, a point (x, y) of the plane belongs to L whenever:

$$x \cos\theta + y \sin\theta = \rho. \tag{4.1}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This equation is the heart of efficiently detecting straight lines in an image.

Standard Hough Transform

First, the θ-space is quantified: let $\theta_1, \theta_2, \ldots, \theta_{k_\theta}$ be the angles in the quantization. Naturally the k_θ different angles in the quantization are a determining factor in the accuracy of the output. While increasing k_θ increases the accuracy of the system, it also increases the computation time complexity accordingly. We also select k_θ so that for all i, $\theta_i = \frac{\pi i}{k_\theta}$.

Next observe some constant properties of all images that are mapped to any matrix. For clarity we shall restrict the range of the angle θ to $[0, \frac{\pi}{4}]$. It turns out that handling all other angles is perfectly similar. Therefore we consider any angle satisfying

$$0 \leq \theta \leq \frac{\pi}{4}. \tag{4.2}$$

Given this range, it is easy to confirm that

$$\frac{\sqrt{2}}{2} \leq \cos\theta \leq 1 \tag{4.3}$$

and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$0 \leq \sin\theta \leq \frac{\sqrt{2}}{2}. \tag{4.4}$$

Thus for every entry$[i,j]$ in the image space we may write

$$\rho_{i,j} = \lfloor i\cos\theta + j\sin\theta \rfloor. \tag{4.5}$$

We have the following observations which can easily be derived from equations 4.2 - 4.5.

Observation 1[54]. For all $j(0 \leq j \leq N-1)$, $\rho_{0,j} \leq \rho_{1,j} \leq \ldots \leq \rho_{N-1,j}$. Furthermore, no three consecutive elements in row j have the same value of ρ.□

Observation 2[54]. For all $i,j(0 \leq i,j \leq N-1)$, $0 \leq (\rho_{i+1,j} - \rho_{i,j}) \leq 1$. Thus no two consecutive ρ values in row j differ by more than 1. □

Observation 3[54]. For all $i,j(0 \leq i,j \leq N-1)$, $\rho_{i,j} \neq \rho_{i+1,j+1}$.□

Observation 4[54]. For all $i,j(0 \leq i,j \leq N-1)$,

$$\rho_{i,j-1} = \rho_{i,j+1} \Longrightarrow \rho_{i,j-1} = \rho_{i,j} = \rho_{i,j+1}.\square$$

Figure 4.4 helps to illustrate these properties, and is added to provide clarity to the argument. A simplifying observation may be made from figure 4.4 and equation 4.2, the values of ρ are completely independent of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

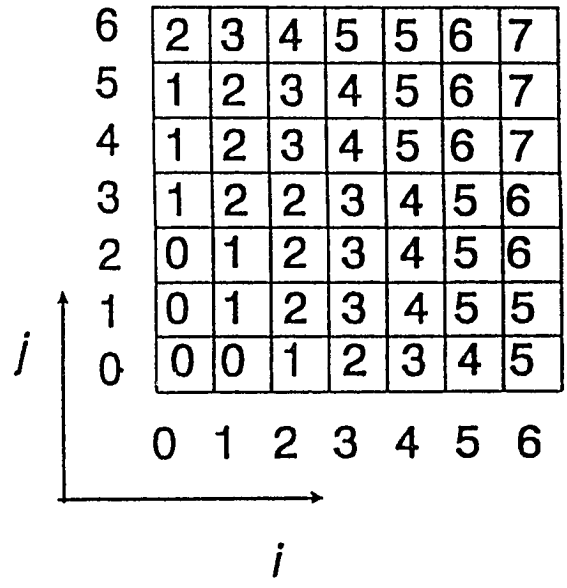

Figure 4.4: The ρ-values of a 7×7 grid for $\theta = \frac{\pi}{8}$.

image being processed. Thus these values need only be computed once, even for the examination of multiple images, provided sufficient storage exists to maintain the template.

Consider the shaded areas in figure 4.5, each ρ value may be considered as an independent linked list of processing elements. This representation causes each processing element to only have to store the ID of the next processing element in its particular list.

With the lists for each value of θ identified, we examine the information of the image itself. Of interest are pixels which form the edges of compo-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

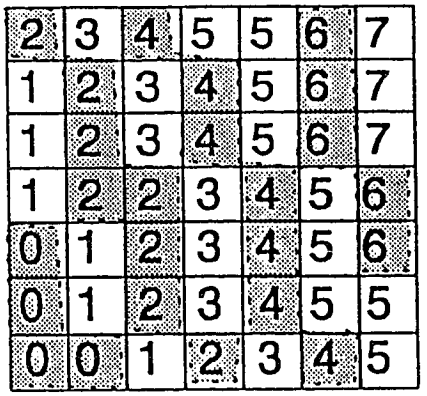

Figure 4.5: The linked lists formed for the values of ρ.

nents with in the image. This is done in the straightforward manner. each processing element which stores a non background pixel need only examine the information stored by processing elements holding adjacent pixels of the image. If a processing element finds a neighboring pixel is the background (usually assumed to be of value 0) it identifies itself as an edge pixel. marking itself with a 1 otherwise it marks itself with a 0.

All that remains is to sum the marking values along the list determined by the current value of θ. If the sum exceeds a threshold, τ, the parameters $[\rho, \theta]$ are said to determine a valid straight line in the image.

Thus for all lists which produce a sum which exceeds τ we have a poten-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tial line and the following result:

A linear array with reconfigurable global buses with n^2 processing elements can find the Hough Transform of an $n \times n$ binary image in $O(k_\theta \log n)$ time, even if the entire image were to be considered edges. This sum can be found efficiently in $O(\log n)$ time [42].

The Hough Transform using Image Components

The communication flexibility of the linear array with reconfigurable global buses offers at least another method by which to accomplish the Hough transform. To begin this modification of the Hough Transform, we identify the pixels in the image which have a chance of belonging to a straight line. These are the pixels which form the edges of components. Additionally, we form these edge pixels into linked lists around each component, the details of this operation may be found in [41].

We depart from the procedure described above by removing the floor function from equation 4.5 to obtain ρ such that

$$\rho_{i,j} = i \cos\theta + j \sin\theta. \tag{4.6}$$

Thus we have for every angle θ_i, all the edge pixels (x, y) that have the same

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

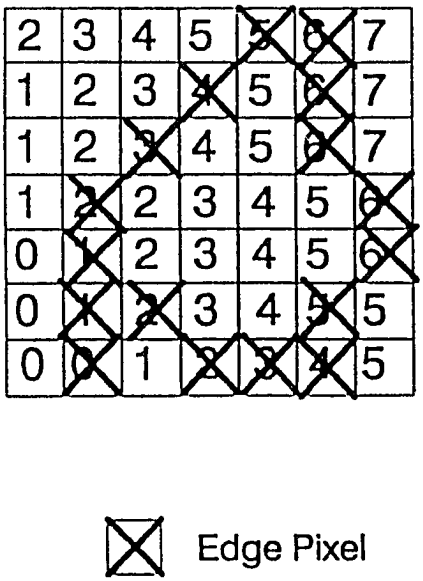

Figure 4.6: $\rho = 2$ may not be a "significant" line, but $\rho = 6$ is definitely significant.

ρ value[2] in 4.6 define a possible line in the image space.

To complete out modification of the standard procedure, we introduce one additional resolution parameter, φ, to represent the lower threshold for a line. ϕ is an integer number of pixels away from the current pixel that the processing element will look, to compare ρ values. A large ϕ value will cause only long straight edges to be detected. A smaller value will detect shorter, but possibly insignificant edges. Refer to figure 4.6.

Each processing element holding an edge pixel, finds ρ for its pixel and

[2]within some ϵ that also helps to determine the resolution of the output.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

θ_i. It then compares its ρ (referred to as ρ_j) to the ρ found by the processing element ϕ pixels clockwise along the edge (referred to as ρ_ϕ). If $|\rho_j - \rho_\phi| < \epsilon$, a potential straight line exists between the pixel at PE_j and PE_ϕ.

This method is an improvement over the basic Hough transform in that it localizes the computational effort on the component edges. This would actually allow fewer than n^2 processing elements. It additionally provides increased resolution control through the ϵ and ϕ parameters. Related straight edges of separate components which will be in separate lists will not be detected by this method.

While this method does have the draw back noted above, it does have the flexibility of not requiring n^2 processing elements. It is obvious that the method can process a single component at a time requiring only p processing elements, where p is the number of pixels which from the edge of the component. If the processing elements are assumed to be powerful enough, each processing element could handle a section of the perimeter of the component, or even an entire component. This type of processing is more MIMD in nature and will be a topic of later study.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3 The Linear Array with Reconfigurable Global Buses Efficiently Simulates Concurrent Read PRAM Models

The purpose of this section is to show that a linear array with reconfigurable global buses with $\max N, M$ processing elements is at least as powerful as the Concurrent-Read Exclusive-Write Parallel Random Access Machine, CREW-PRAM, model with N processors and M memory locations. Additionally, it is shown that the linear array with reconfigurable global buses is as powerful as the Concurrent-Read Concurrent-Write PRAM, CRCW-PRAM, under a certain concurrent write resolution scheme.

4.3.1 Introduction

Shared memory model SIMD computers have received considerable algorithmic examination in many areas of computation, see [4] for an overview. We are interested in the two most powerful shared memory models, the Concrurrent-Read Exclusive-Write or CREW and the Concurrent-Read Concurrent-Write or CRCW. These models produce the fastest theoretical algorithms.

While the models only assume very primitive processors, the concurrent

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reads and writes often cause additional complexity when simulating these models on other models and architectures. Thus any model which can efficiently handle this concurrency, can also inherit the vast quantity of algorithms which are already developed. The linear array with reconfigurable global buses is such a model. In addition to inheriting the algorithms, the model also inherits the corresponding lower bounds, many of which are based on quite complex proofs.

4.3.2 The linear array with reconfigurable global buses vs the CREW-PRAM

The CREW-PRAM model permits multiple processors to read a memory location, but only one processor may write to a particular memory location at any step. We now consider a problem Π which requires time $O(t)$ on a CREW-PRAM with N processors and M memory locations. The LARGB will simulate the CREW-PRAM as follows, the first N processing elements will function as the N processors of the CREW-PRAM. Since each processing element can maintain its own local memory, the first M processing elements are responsible to maintain the M memory locations. Thus, any computational step which occurs in the CREW-PRAM is directly ported to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the processing elements of the LARGB. When processor$_i$ writes to memory location j, PE$_i$ accomplishes the write by placing the value to be written onto HB$_j$ and PE$_j$ then writes the value to its memory. No collisions occur because of the exclusive write constraint of the pram model, thus all processing elements can write to a horizontal bus and then accomplish the store in parallel. Similarly, to accomplish a read operation, each processing element places the value in its memory onto its own horizontal bus, then any number of processing elements can read any horizontal bus. Thus the LARGB solve problem Π in $O(t)$ steps with $\max(N, M)$ processing elements .

We also consider a problem Π which can be solved by an LARGB, that does not allow more than one processing element to write to a single horizontal bus, with N processing elements each with m memory locations in $O(t)$ steps. The processors of the CREW-PRAM can directly accomplish any computational step that the processing elements of the LARGB can accomplish. To model the memory of the LARGB, we have a CREW-PRAM with N processors and Nm memory locations. If PE$_i$ broadcasts a value it does so by placing the value onto HB$_i$, then each processing element reads HB$_i$. The CREW-PRAM model accomplishes this by causing processor i

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to write to memory location j then each processor reads memory location j. The LARGB accomplishes multiple reads by placing the candidates to be read on to the horizontal bus corresponding to the processing element which stores the value. Then any processing element needing the value reads the appropriate horizontal bus, In the CREW-PRAM, multiple reads are directly accomplished by each processor reading the memory location it requires.

Thus we have proven the following result:

Theorem 16. *Any problem Π_1 which a CREW-PRAM with N processors and M memory locations can solve in $O(t)$ time, the LARGB with $\max(N, M)$ processing elements can also solve in $O(t)$ time and conversely any problem Π_2 which the LARGB that does not allow multiple processing elements to write to the same horizontal bus with N processing elements, each with m memory locations, can solve in $O(t)$ time can be solved by the CREW-PRAM with N processors and Nm memory locations in $O(t)$ time.*

□

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3.3 The linear array with reconfigurable global buses vs the CRCW-PRAM

The CRCW-PRAM model allows multiple processors to either read or write to a single memory location. While multiple processors reading the same memory location cause no algorithmic difficulty, multiple processors writing to a single location require a specific resolution method. A number of resolution methods have been proposed, these include:

- allow the processor with the minimum(maximum) identification number to succeed;
- allow the processor with the minimum(maximum) value to succeed;
- allow any processor to succeed randomly;
- allow any processor to succeed only if all processors are trying to write the same value, otherwise the result is undetermined.

We adopt the last resolution method because it makes the most algorithmic sense in that, a single memory location should only be assigned a single value, and the algorithm designer needs to be able to predict this value. These requirements definitely rule out the third resolution scheme, and identifications

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of minimums(maximums) not only imply additional computation, but also that different values are candidates for a single memory location. Additionally, this resolution method is cohesive with the architecture of the linear array with reconfigurable global buses.

Consider a problem Π which requires t steps on a CRCW-PRAM with N processors and M memory locations. These steps will fall in to one of the following categories:

1. Processor$_i$ accomplishes computation λ, $i \in \{1 \ldots N\}$;
2. Processor$_i$ reads from memory location j, $i \in \{1 \ldots N\}, j \in \{1 \ldots M\}$;
3. Processor$_i$ writes to memory location j, $i \in \{1 \ldots N\}, j \in \{1 \ldots M\}$.

A processing element of a LARGB can accomplish a computation λ in k steps just as a processor of the CRCW-PRAM requires k steps to accomplish λ. Additionally, each processing element stores a single value. For PE_i to read memory location j two steps are required to occur:

1. PE_j places the value it holds on HB_j;
2. PE_i reads HB_j.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PE_i write to memory location j in a similar manner:

1. PE_i places the value to be written onto HB_j;

2. PE_j reads HB_j and stores the value.

The read and write methods above, again show how a single processing element accomplishes the task, but do not directly address how concurrency effects the model. Concurrent reads are accomplished in a straightforward manner, and as noted above any quantity of processing elements can read from the same horizontal bus. Recall the concurrent write resolution method, if a subset of the processing elements all attempt to write to memory location j, the write only succeeds if all processing elements attempt to write the same value, otherwise an undetermined result occurs. Concurrent write on the LARGB is accomplished in a similar manner to the concurrent read, in that, all processing elements which want to write to memory location j place the value to be written onto HB_j. Then PE_j reads HB_j and stores the value. If all processing elements did not place the same value on the horizontal bus , the stored result is undetermined.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

All that remains is to address the number of horizontal buses requires to handle all communication without conflict and maintain all the power of the CRCW-PRAM. Since the CRCW-PRAM model employs communication through the shared memory, sufficient buses must be available to handle all memory accesses. Thus M horizontal buses will be required to fully model the CRCW-PRAM.

We have proven the following result:

Theorem 17. *A LARGB with K processing elements and K horizontal buses , where $K = \max(N, M)$, is at least as powerful as a CRCW-PRAM with N processors and M memory locations, in that, the LARGB solves problem Π in $O(t)$ steps if the CRCW-PRAM solves Π in $O(t)$ steps.* □

4.4 Linear Arrays with Reconfigurable Global Buses more powerful than any non-Reconfigurable Architecture

We now examine the power of the linear array with reconfigurable global buses through direct comparison between this model and some currently

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

implemented architectures. We make comparison to the mesh, the mesh with global buses, the tree-of-processors and the hypercube. We shall show that the linear array with reconfigurable global buses is at least as powerful as these actual architectures, additionally a problem that the linear array with reconfigurable global buses handles more efficiently than any non-reconfigurable architecture is presented.

4.4.1 Introduction

Many comparisons have been done between parallel architectures since their initial development. Some of these comparisons have results such as, architecture α_1 is strictly better than α_2, while others find that α_1 more efficiently solves problem Π_1, but α_2 more efficiently solves Π_2. This is not unexpected, given that design criteria for α_i is often directed towards the class of problems Π_i [13, 46, 45, 29, 73, 81].

We intend to examine the linear array with reconfigurable global buses (LARGB) versus the mesh, mesh with global buses, hypercube, and tree architectures. Each of these architectures are well published and have actual implementations in current use [29].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

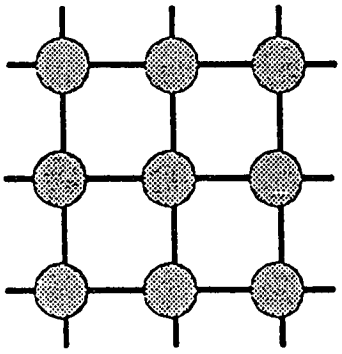

Figure 4.7: A simple mesh architecture

The LARGB is a linear array of processing elements[3]. Augmented by a bus interconnection scheme, refer to figure 4.7. Each processing element controls the vertical bus (VB) to which it is connected. At each intersection of horizontal (HB) and vertical buses is a smart interconnection point (SIP). At any step of computation, PE_i can set $SIP_{i,j}$ to connect to HB_j. A processing element can only read/write to a single horizontal bus at any given step.

The processing elements are considered to be able to accomplish el-

[3]To avoid confusion, the processing elements of the LARGB will be referred to as such and for the other architectures they shall be referred to as processors.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ementary mathematical and logical operations only. We assume that the processors of the architecture being simulated are of equal power. For example, when we model a hypercube we assume one with simple processors not those used in the iPSC/860 [44]. Each processing element maintains a local memory which it can directly access.

4.4.2 LARGB Vs. Mesh Architecture

The general mesh architecture consists of an $n \times m$ array of processors with only nearest neighbor communication ability. Thus, a processor may only communicate with 4 or fewer[4] other processors.

We begin with a LARGB with nm processing elements. The $n \times m$ structure of the mesh is directly mapped onto the linear array. That is, the first row of processors are mapped to the first n processing elements, ..., the i^{th} row of processors are mapped to processing elements $(i-1)n+1$ through in.

Local communication is accomplished in the straightforward manner. Left and right neighbors are still adjacent on the linear array, thus the

[4]We consider the basic mesh not to have "wrap-around" connections.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mapping of $P_{i,j}$ communicating with $P_{i,j+1}$ is $PE_{n(i-1)+j}$ communicates with $PE_{n(i-1)+j+1}$. The mapping for $P_{i,j}$ to communicate with $P_{i+1,j}$ is equally as direct, $PE_{n(i-1)+j}$ communicates with PE_{ni+j}.

Since any correct algorithm for a simple mesh would not communicate with a neighbor that is not a neighbor these conditions need not be checked in the simulation. Since the bus system of the LARGB allows any two processing elements to communicate at any step, we have the following result:

Observation 1. *Any problem* Π *that a simple* $n \times m$ *mesh can solve in O(t) time, a LARGB with* nm *processing elements can solve in O(t) time or less.*

□

4.4.3 LARGB Vs. Mesh with Global Buses

The mesh with global buses is a simple mesh augmented by row and column buses, see figure 1.3. This mesh is mapped onto the linear array in exactly the same manner as the simple mesh.

In order to simulate the n global row buses and the m global column buses we need only assign specific horizontal buses to serve as those buses. Specifically, let $HB_1 \ldots HB_n$ serve as row buses and $HB_{n+1} \ldots HB_{n+m}$ serve

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as column buses. All that remains is for each processing element to identify the row and column to which it belongs, this will allow it to read/write to the correct row and column buses.

First, we must identify each n^{th} PE. This is accomplished by the standard pointer doubling technique starting with PE_n and progressing in multiples of n. These processing elements can then divide their own PEID by n to obtain their row number. To inform each PE of their row ID we accomplish the multiple source problem with each n^{th} PE functioning as a source. The column ID is a direct computation from $PEID - n(RowID - 1)$.

Thus we have established the ability to accomplish any communication that the mesh with global buses can accomplish. We have the following result:

Observation 2. *Any problem* Π *that an* $n \times m$ *mesh with global buses can solve in* $O(t)$ *time, a* $LARGB$ *with* nm *processing elements can solve in* $O(t)$ *time once an initial pre-processing is accomplished which requires* $O(\log n + \log m)$ *time.* □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.4 LARGB Vs. Hypercube

As noted earlier, we assume simple processors and accordingly an SIMD implementation at this point. An n-dimensional hypercube has 2^n processors, each with direct communication with n other processors.

For the LARGB to simulate the hypercube, each processing element needs to be able to identify which other processing elements are its "neighbors." This is accomplished in the obvious manner since in a hypercube, each PID differs from its neighbors by a single bit in the binary representation. This translates to differing by integer powers of 2. Thus the neighbors of P_i $(i < n)$ are P_j, where $j = i + 2^k, k\epsilon\{0 \ldots n-1\}$. Thus the LARGB with 2^n processing elements can map the communication links of an n-dimensional hypercube.

Modeling the total communication processes of a hypercube is not as direct. Local broadcasts in which a processor receives multiple values cannot be accomplished in O(1) time on the LARGB. It is, however, obvious that any point to point communication on a hypercube can be accomplished on the LARGB in the same number of steps. Actually, many point to point communications on the LARGB can be accomplished in fewer steps than on

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the hypercube. Thus we have the following result:

Observation 3. *A problem* Π *which an n-dimensional hypercube with simple processors solves in t steps using only point to point communication, can be solved on a LARGB with* 2^n *processing elements in* $t-\tau(t)$ *steps where* $\tau(t)$ *is the function of multiple step communications that the LARGB accomplished in one step.* □

4.4.5 LARGB Vs. Tree machines

The fixed interconnections of the tree architecture can be directly mapped on the LARGB. A binary tree may be mapped as follows:

- The root is PE_1;
- The level of PE_i is $\lfloor \log_2 i \rfloor$;
- The children of PE_i are PE_k and PE_{k+1} where $k = i \times 2^{\lfloor \log_2 i \rfloor - 1}$.

Thus the modeling of a binary or other regular degree tree is direct. An initial preprocessing cost of $O(\log_k n)$ steps is all that is required for the LARGB to model a balanced k-tree. The cost results from the need to indirectly compute the $\log i$ and k^i functions. The procedure is similar to that used

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to pre-process for global buses and will not be repeated. Thus we have the following result:

Observation 4. *A problem* Π *which is solved on an* n *processor balanced* k*-tree in* t *steps can be solved on the LARGB in* $t+O(\log_k n)$ *steps.* □

4.4.6 The List Ranking Problem

A particularly dynamic and unstructured problem is that of list ranking. This problem is solved in Chapter 2. It may be directly observed that any of the fixed architectures examined above cannot handle the general case of the list ranking problem as efficiently as the linear array with reconfigurable global buses.

The key to the efficiency of the list ranking algorithm is that the order of the input is of no concern. At each iteration the "next" processing element is just as easy to contact as the previous one. There is no need to pass messages or consider the large communication diameter of a mesh. The linear array with reconfigurable global buses is even more efficient that a fully connected architecture [19] since the location of values is direct, no messages are required. When a processing element sends out information, it

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

does not care which processing elements need to read it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

Conclusions

We have presented a new approach to the hybrid architecture with the goal of improving the potential power of an array of processing elements without requiring an excessive number of processing elements to achieve acceptable performance.

5.1 Fundamental Algorithms

Algorithms for the basic prefix computations, list ranking and parenthesis matching on this architecture are presented. These algorithms begin to demonstrate the potential power and flexibility of the linear array with reconfigurable global buses. It is this flexibility that allows all of these algorithms to accomplish the desired task in $O(\log n)$ time on $O(n)$ processing elements.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2 Graph Theory

All of the algorithms presented here solve important fundamental graph problems. The algorithms are designed to handle very large sparse graphs as might arise in real applications such as finite element analysis. No adjacency matrix representation is required, thus storage efficiency is improved considerably. Similar problems are solved by S.K. Das et al. in [15], but their algorithms are more complex with less flexibility.

We have shown an optimal method for obtaining a spanning forest for a very large sparse graph, tailored to the linear array with reconfigurable global buses. The spanning forest directly yields solution to the connected components question. The algorithm handles the minimum volume of information possible to obtain the result, thus maximizing the potential size of the input.

The bridge algorithm is extended to obtain bi-connected components, something that was not accomplished by S.K. Das et al. The graph algorithms offered are stable or even improve with the scarcity of the input graph. The examination of the Least Common Ancestor problem further

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

demonstrates that the linear array with reconfigurable global buses can handle dynamically changing data structures with no loss of performance. All of the results relative to the LCA problem are shown to be optimal.

5.3 Image Processing

We addressed some low level and medium level image processing problems. Specifically, we have presented $O(\log n)$ time algorithms for finding the perimeter and area of binary images. We have also presented a fast component labeling algorithm on an n processing element LARGB. We utilize the extreme flexibility of the reconfigurable linear buses of the LARGB to generate an $O(\log n)$ time algorithm for an $\sqrt{n} \times \sqrt{n}$ binary image. This matches the performance of the fastest parallel array with reconfigurable buses algorithm which was recently presented in [52].

The detection of shape in a computer image is a fundamental problem of computer and robot vision. The Hough transform is a well-know tool for the detection of lines within an image. We utilized the communication flexible, linear array with reconfigurable global buses, to accomplish the Hough transform on an $n \times n$ binary image in $O(k_\theta \times \log(n))$ time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

While the linear array with reconfigurable global buses offers excellent performance on the image processing tasks examined, it does not appear that the communication flexibility is truly utilized to an extent that justifies the cost of the bus system. For dedicated image computations, a mesh based architecture is recommended.

5.4 Simulating Parallel Random Access Machines

The massive flexibility of communication in the linear array with reconfigurable global buses allows it to fully simulate any operation which the CREW-PRAM can accomplish in comparable time with a comparable number of processing elements. Conversely, any operation that the linear array with reconfigurable global buses can accomplish without allowing multiple processing elements to write to the same horizontal bus , the CREW-PRAM can accomplish in comparable time with a comparable number of processors. This provides a known set of lower bounds for LARGB algorithms.

If the linear array with reconfigurable global buses does allow multiple processing elements to write to a single horizontal bus , we have shown that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the linear array with reconfigurable global buses with at least $\max(N, M)$ processing elements is at least as powerful as the theoretical CRCW-PRAM model which employs the concurrent write resolution scheme of having a successful write attempt only if all processors are attempting to write the same value. In the alternate case, the result of the write is undetermined. Since many algorithms, which employ concurrent writes to the same memory location, are only concerned with some result having occurred at least one processor, this scheme is perfectly feasible.

5.5 The LARGB versus Actual Architectures

We have shown how the excellent communication flexibility of the linear array with reconfigurable global buses allows it to fulfill the communication requirements of a range of architectures. It should be obvious that the relationships offered are not reflexive. The direct reconfigurable communication of the LARGB allows us to identify a problem which it can solve in fewer steps than any of the architectures examined. The list ranking problem is an elementary problem that benefits from this model. More complex problems should achieve greater benefit from the model.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.6 Overall Conclusion About the Proposed Model

The linear array with reconfigurable global buses was shown to be at least as powerful as the Parallel Array with a Reconfigurable Bus System in many applications while requiring less physical space to implement. With a considerable portion of architecture cost being related to physical space, this architecture definitely appears to be a viable proposal to meet the increasing demand for computational performance without excessive complexity. .

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6

Open Questions

6.1 The LARGB as an MIMD Machine

This work addressed the implementation of a wide range of algorithms on the linear array with reconfigurable global buses under an SIMD model. There should be no doubt that the architecture could definitely be implemented as an MIMD machine. It is however uncertain if an MIMD model would have as much of an advantage over less fully connected architectures as the SIMD version did. With processor independence comes the need for some level of communication synchronization. The inherent lack of tight synchronization in an MIMD application is only an advantage to a system that possesses communication limitations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.2 Speed-up available from the LARGB

The exact speed-up that the linear array with reconfigurable global buses offers over architecture α remains open. As does the identification of what classes of problems the LARGB is particularly suited for or poorly suited for. It is suspected that many numerical problems which are efficiently solved by a simple mesh will gain little from the added communication power of the LARGB and thus the additional cost would be unwarranted in these cases. Problems requiring large quantities of data driven communication should be best suited to this architecture.

6.3 Very Large Numerical Problems

With the power of many RISC based processing elements in tight communication with each other, the linear array with reconfigurable global buses could also be employed as a very high performance numerical processing engine. The high speed interprocessor communication considerably increases the potential for real-time solution to problems which are both computationally and communication intensive.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bibliography

[1] Active Memory Technology, Irvine, Ca. *DAP Series: FORTRAN-PLUS Language (man002.03)*, 1988.

[2] John P. Hopcroft Aho, Alfred V. and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] John P. Hopcroft Aho, Alfred V. and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[4] Selim G. Akl. *Design and Analysis of Parallel Computer Algorithms.* Prentice-Hall, Inc., 1989.

[5] Mikhail J. Atallah. On Multidimentional Arrays of Processors. *IEEE Transactions on Computers*, 37(10), October 1988.

[6] G.H. Barnes, R. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17, 1968.

[7] R. Bernard. RISCs - Reduced Instruction Set Computers - Make Leap. *Systems Software*, pages 81–84, December 1984.

[8] W.E. Blanz, D. Petkovic, and L.C. Sanz. *Signial Processing Handbook*, chapter Algorithms and Architectures for Machine Vision. M. Dekker, New York, 1989.

[9] C. Bruno and S. Bradley. THe RISC Factor. *Datamation*, June 1986.

[10] Shyh-Kwei Chen. n+ Cube: The ExtraDimensional nCube. *1990 International Conference on Parallel Processing*, pages 583–584, 1990.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[11] Y-C Chen, W-T Chen, G-H Chen, and J-P Sheu. Designing efficient parallel algorithms on mesh-connected computers with multiple broadcasting. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[12] C. Chin and W. Lin. A massively Parallel Processing System Based on A Hyper-Crossbar Network. *2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 463–466, October 1988.

[13] Suresh Chittor and Richard Enbody. Hypercubes Vs. 2D Meshes. *Proceedings of the Fourth SIAM Conference on Parallel Processing of Scientific Computing*, pages 313–318, 1989.

[14] Charles Clos. A Study of Non-Blocking Switching Networks. *The Bell System Technical Journal*, pages 406–424, March 1953.

[15] S.K. Das, N. Deo, and S. Prasad. Parallel graph algorithms for hypercube computers. *Parallel Computing*, 13:143–158, 1990.

[16] Keith Diefendorf and Michael Allen. The motorola 88110 superscalar risc microprocessor. *Proceedings of CompCon 92*, pages 157–162, February 1992.

[17] Kshitij A. Doshi and Peter J. Varman. Optimal graph algorithms on a fixed-size linear array. *IEEE Transactions on Computers*, C-36(4), April 1987.

[18] Mary M. Eshaghian. Parallel Algorithms for Image Processing on OMC. *IEEE Transactions on Computers*, 40(7):827–833, July 1991.

[19] Tse-yun Feng. A Survey of Interconnection Networks. *Computer*, pages 12–27, December 1981.

[20] M.J. Flynn. Very High Speed Computing Systems. *Proceeding of the IEEE*, 1966.

[21] C. J. Georgiou. Fault-Tolerant Crosspoint Switching Networks. *Proceedings of the Fourteenth International Symposium on Fault-Tolerant Computing*, pages 240–245, July 1984.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[22] J. Ghosh and A. Varma. Reliable Design of Multichip Non-blocking Crossbars. *1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 70–73, September.

[23] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen. SPLASH: A Reconfigurable Linear Logic Array. *1990 International Conference on Parallel Processing*, pages 5226–532, 1990.

[24] Martin C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

[25] C. Guerra and S. Hambrush. Parallel Algorithms for Line Detection on a Mesh. *Proceedings of IEEE Workshop on Computer Architectures for Pattern Analysis and Machine Intelligence*, pages 99–106, 1987.

[26] S.E. Hambrush and M. Luby. Parallel Asyncronous Connected Components in a Mesh. *Information Processing Letters*, 38:257–263, 1991.

[27] S.E. Hambrush and L. TeWinkel. A Study of Connected Component Algorithms on the MPP. *Proeedings of the Third International Conference on Supercomputing*, pages 477–483, 1988.

[28] Dov Harel and Robert E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal of Computing*, pages 338–355, May 1984.

[29] R.W. Hochney and C.R. Jesshope. *Parallel Computers 2.* Adam Hilger, Bristol and Philadelphia, 1988.

[30] R. Michael Hord. *Parallel Supercomputing in SIMD Architectures.* CRC Press, 1990.

[31] J. Illingworth and J. Kittler. A Survey of the Hough Transform. *Computer Vision, Graphics and Image Processing*, 44:87-116, 1988.

[32] A.V. Kulkarni and D.W.L. Yen. Systolic processing and an implementation for signal and image processing. *IEEE Transactions on Computing*, C-31, October 1982.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[33] V.K. Prasanna Kumar and C.S. Raghavendra. Array processor with multiple broadcasting. *Journal of Parallel and Distributed Computing*, 4, 1987.

[34] H.T. Kung. Systolic algorithms for the CMU WARP processor. *in Proceedings of 7th International Conference on Pattern Recognition*, July 1984.

[35] F.T. Leighton. New lower bound techniques for VLSI. *in Proceedings of 22nd Annual IEEE Symposium Foundations of Computer Science*, October 1981.

[36] F.T. Leighton. Parallel computation using meshes of trees. *in Proceedings 1983 international workshop on graph theoretic concepts in Computer Science*, 1983.

[37] S. Levialdi. On Shrinking Binary Picture Patterns. *Communications of the ACM*, 15:7–10, 1972.

[38] M.D. Levine. *Vision in Man and Machine.* Mcgraw-Hill, 1985.

[39] Hungwen Li and Massimo Maresca. Polymorphic-torus network. *IEEE Transactions on Computers*, 38(9), September 1989.

[40] R. Lin, S. Olariu, J.L. Schwing, and J. Zhang. Sorting in O(1) time on an nxn reconfigurable mesh. *Proceedings of the European Workshop on Parallel Computing*, March 1992.

[41] Peter J. Looges. Image Processing on the Linear Array with Reconfigurable Global Buses. *IEEE Transactions on Pattern Analysis and Machine Intelligence, submitted*, 1992.

[42] Peter J. Looges and Nathan R. Sharp. Fundamental Algorithms on Arrays with Reconfigurable Global Meshes. *Parallel Computing, submitted*, January 1992.

[43] Massimo Maresca and Hungwen Li. Connection autonomy in SIMD computers: A VLSI implementation. *Journal of Parallel and Distributed Computing*, 7, 1989.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[44] C.L. McCreary, M.E. McArdle, and J.D. McCreary. Broadcast Communication Delay Metric for the iPSC/2 and iPSC/860 Hypercubes. *Proceedings of 30th Annual ACM Southeast Conference*, 1992.

[45] R. Miller and Q. Stout. Efficient parallel convex hull algorithms. *IEEE Transactions on Computers*, 37:1605–1618, 1988.

[46] R. Miller and Q. Stout. Mesh computer algorithms for computational geomentry. *IEEE Transactions on Computers*, 38:321–340, 1989.

[47] Russ Miller, V.K. Prasanna Kumar, Dionisios Reisis, and Quentin F. Stout. Data movement operations and applications on reconfigurable VLSI arrays. *Proceedings of International Conference on Parallel Processing*, 1988.

[48] Russ Miller and Quentin F. Stout. Data movement techniques for the pyramid computer. *SIAM Journal of Computing*, 16(1), February 1987.

[49] N. Mokhoff. New RISC Machines Appear as Hybrids with both RISC and CISC Features. *Comput. Des.*, pages 22–25, April 1986.

[50] D. Nath, S.N. Maheshwari, and C.P. Bhatt. Efficient VLSI networks for parallel processing based on orthagonal trees. *IEEE Transactions on Computing*, C-32, June 1983.

[51] nCUBE, Beaverton, Oregon. *nCUBE 2 Supercomputers: Systems Technical Overview*, 1990.

[52] Stephan Olariu, James L. Schwing, and Jinyuan Zhang. Fast component labeling on reconfigurable meshes. *Old Dominion University*, Janurary 1992.

[53] Stephan Olariu, James L. Schwing, and Jinyuan Zhang. Fast computer vision algorithms on reconfigurable meshes. *Proceedings of IPPS*, 1992. to appear.

[54] Stephan Olariu, James L. Schwing, and Zhang, Jingyuan. Computing the Hough Transform on Reconfigurable Meshes. *Proceedings of Vision Interface*, 1992.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[55] Michael J. Phillip. Performance issues for the 88110 risc microprocessor. *Proceedings of CompCon 92*, pages 163–168, February 1992.

[56] Steve Plimpton, Sudip Dosanjh, and Randy Krall. Is SIMD Enough for Scientific and Engineering Applications on Massively Parallel Computers. *Proceedings of CompCon 92*, pages 95–102, February 1992.

[57] V.K. Prasanna-Kumar and D. I. Reisis. Image Computations on Meshes with Multiple Broadcast. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1194–1202, November 1989.

[58] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, 1987.

[59] R. Ragan-Kelly and R. Clark. Applying RISC Theory to a Large Computer. *Comput. Des.*, pages 191–198, November 1983.

[60] I.V. Ramakrishnan and P.J. Varman. Modular matrix multiplication on a linear array. *IEEE Transactions on Computing*, C-32, October 1983.

[61] A.P. Reeves. Parallel Computer Architecture for Image Processing. *Computer Vision, Graphics, and Image Processing*, 25:68–88, 1984.

[62] A. Rosenfeld, J. Ornelas, and Y. Hung. Hough Transform Algorithms for Mesh-Connected SIMD Parallel Processors. *Computer Vision, Graphics and Image Processing*, 41:293–305, 1988.

[63] Azriel Rosenfeld. Parallel Image Processing using Cellular Arrays. *IEEE Computer*, 16:14–20, 1983.

[64] Azriel Rosenfeld. The Impact of Massively Parallel Computers on Image Processing. *2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 21-27, October 1988.

[65] Azriel Rosenfeld and Avinash C. Kak. *Digital Picture Processing*, volume 1–2. Academic Press, 1982.

[66] P. Sadayappan and F. Ercal. Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes. *IEEE Transactions on Computers*, C-36(12):1408–1424, Dec 1987.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[67] Howard J. Seigal. *Interconnection Networks for Large-Scale Parallel Processing and Case Studies.* McGraw Hill, second edition, 1990.

[68] M. Seither. Pyramid Challenges DEC with RISC Supermini. *Mini-Micro Systems*, pages 33–36, August 1985.

[69] C.L. Seitz. Concurrent VLSI architectures. *IEEE Transactions on Computing*, C-33, December 1984.

[70] Yossi Shiloach and Uzi Vishkin. An O(log n) Parallel Connectivity Algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[71] Howard J. Siegal, James B. Armstrong, and Daniel W. Watson. Mapping Computer-Vision-Related Tasks on Reconfigurable Parallel Processing Systems. *Computer*, pages 54–63, Feburary 1992.

[72] William Stallings. *Reduced Instruction Set Computers.* IEEE Computer Society Press, second edition, 1990.

[73] Harold S. Stone. *High-Performance Computer Architectures.* Addison-Wesley, Reading, Ma., 1990.

[74] H.S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computing*, C-20, Feburary 1971.

[75] Quentin F Stout. Mesh connected computers with multiple broadcasting. *IEEE Transactions on Computers*, C-32(9), September 1983.

[76] Robert E. Tarjan and Uzi Vishkin. An efficient Parallel Biconnectivity Algorithm. *SIAM Journal of COmputing*, pages 862–874, November 1985.

[77] Robert Tobias. The LR33020 GraphX Processor: A Single Chip X-Terminal Controler. *Proceedings of CompCon 92*, pages 358–363, February 1992.

[78] Mike Uhler. High Performance Single Chip VAX Microprocessor. *Proceedings of CompCon 92*, page 215, February 1992.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[79] L.G. Valiant and G.J. Brebner. Univeral schemes for parallel computation. *in Proceedings of 13 ACM Symposium of Theory of computing*, 1981.

[80] B-F. Wang, G-H. Chen, and F-C. Lin. Constant time sorting on a processor array with a reconfigurable bus system. *Information Processing Letters*, 34:187–192, April 1990.

[81] H.C. Wang and Kai Hwang. An Augmented Tree Multiprocessor for Parallel Execution of Multigrid Algorithms. *Proceedings of the Third SIAM Conference on Parallel Processing of Scientific Computing*, pages 424–428, 1987.

[82] R. Weiss. RISC Processors: The New Wave in Computer Systems. *Comput. Des*, pages 53–73, May 1987.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Autobiographical Statement

Peter John Looges was born on March 4, 1963 in East Orange, New Jersey. Graduated from Rensselaer Polytechnic Institute with a Bachelor of Science Degree in Computer Science in May of 1985 and was commissioned an Ensign in the United States Navy. In May of 1991, he graduated from Old Dominion University, receiving a Master of Science Degree in Computer Science.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Publications

Papers Accepted

Greedy Recognition and Coloring Algorithms for Indifference Graphs, (with S. Olariu), **Computer Science and Operations Research: New Developments in Their Interfaces**, Pergamon Press, 1992.

Optimal Greedy Algorithms for Indifference Graphs, (with S. Olariu), **Proceedings of IEEE Southeastcon `92**.

Single Row Routing with Indifference Graphs on the DAP, (with D. Bhagavathi and S. Olariu), **International Journal of Computer Systems Science and Engineering**, vol. 7, no. 4 October 1992, pp 147-154.

Sorting and Merging on the DAP, (with D. Bhagavathi, W.M. Denny, C.E. Grosch, and S. Olariu), **Proceedings of 30th Annual ACM Southeast Conference**.

High Speed Querying with the DAP 510, **4th International Conference on Computing and Information**, May 28-30, 1992.

Efficient Solution to the Convex Hull Problem on the DAP, **Active Memory Technology Technical Report Series**, to appear.

A Fast Selection Algorithm for Meshes with Multiple Broadcasting, (with D. Bhagavathi, S. Olariu, J.L. Schwing, and J..Zhang), **1992 International Conference on Parallel Processing**.

The Hough Transform on the Linear Array with Reconfigurable Global Buses, **Conference on Vision Geometry at OE/Technology 92.**

Papers Submitted

A survey of the Convex Hull Problem on the DAP, **Parallel Computing**, under revision.

Selection on High Performance Architectures, **Supercomputing 1992**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Fundamental Algorithms on Arrays with Reconfigurable Global Buses, (with Nathan R. Sharp), **Parallel Computing**.

Linear Arrays with Reconfigurable Global Buses Efficiently Simulate Concurrent Read PRAM Models, **Parallel Processing Letters**, under revision.

Image Processing on Linear Arrays with Reconfigurable Global Buses, **IEEE Transactions on Pattern Analysis and Machine Intelligence,** under revision.

Optimal Solution to the Least Common Ancestor Problem on the Linear Array with Reconfigurable Global Buses, **Parallel Processing Letters.**

Linear Arrays with Reconfigurable Global Buses. more powerful than any non-Reconfigurable Architecture, **IEEE Transactions on Computers.**

Graph Properties of Very Large Sparse Graphs on the Linear Array with Reconfigurable Global Buses, **Parallel Processing Letters.**

Finding Connected Components of Very Large Sparse Graphs on the Linear Array with Reconfigurable Global Buses, **Parallel Processing Letters.**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.