THE FUNCTION OF A CONNECTION NETWORK BETWEEN HOST AND PROCESSING
ELEMENTS IN MASSIVELY PARALLEL COMPUTER SYSTEMS

Timothy Bridges

Computer Science Department
Indiana University
Bloomington, IN 47405
tbridg@iuwax.cs.indiana.edu (812) 339-3753

Massively Parallel Arch. Corp.
P.O. Box 904
Bloomington, IN 47402
(812) 339-3753

Abstract

Massively parallel computation systems are routinely characterized, identified, and studied based on the connection topology of the processing elements (PEs). This method of classification is flawed since massively parallel computation systems are not stand-alone computers; they are invariably packaged with a standard von Neumann host. This paper examines the function of a connection network between massive numbers of PEs and a single host by comparing the Data Structure Machine (DSM), whose major connection network is a computationally powerful binary tree with the host connected at the root, and the Connection Machine, which provides a very rich and general PE to PE connection network, but whose connection to the host is little more than a buffered wire. The binary tree network used in the Data Structure Machine can be utilized to achieve asymptotic improvements in speed for algorithms that maintain, locate and exploit data parallelism in data structures that can be characterized by a high degree of locality. For example, SUM, MAX, LEFTMOST, and INDEX are all constant time operations on DSM lists, while corresponding Connection Machine algorithms can require linear time to simply find and mark lists.

Keywords: SIMD, Massively Parallel, Connection Topology, Host Interface, Connection Machine, Data Structure Machine

1. Introduction

Of the two architectures being compared here, the Connection Machine is widely known, understood, and available as a commercial product. The Data Structure Machine is clearly less known, therefore requiring the following introduction to the parent architecture, APSA, and an overview of the DSM.

The Applicative Programming Systems Architecture, APSA, extends the traditional von Neumann design by including a special purpose functional unit that serves as a Data Structure Memory, DSM [4,6,7]. The DSM is a massively parallel computing system that utilizes thousands, or even millions, of processing elements in an SIMD paradigm to exploit data structure parallelism. A custom VLSI prototype containing 128 Processing Elements, PEs, has been constructed and is under study at Indiana University (16 chips with 8 PEs/chip). The simple tree-based connection topology used in this DSM allows for extending the system by adding more chips of the same design or by increasing the number of processors per chip. A single chip containing upwards of 128 processors is feasible by using the denser and more expensive technologies available today. Also, the total number of PEs in the system does not affect the specification of a individual PE nor does it increase the complexity of the connection network wiring. A project is underway to construct a DSM of an interesting size, 2K - 4K processing elements, and to implement bank switching so that larger machines can be emulated for study.

Because of its simple binary tree connection topology, the APSA DSM will benefit from economies of scale that could lead to implementations containing over a million processors in the near future. However, the use of such a simple connection topology raises serious questions concerning possible applications for such a machine. PE ↔ PE communication is obviously limited by the bottleneck at the root of the tree. What is less obvious, and what I attempt to show in this paper, is that the tree topology that is a bottleneck for PE ↔ PE communication provides an extremely rich Host ↔ PEs connection network. Further, this network can be used to efficiently implement algorithms of a 1 to many or many to 1 nature. I will present examples of these algorithms and compare their performance to Connection Machine [1], CM, algorithms. The CM was chosen for this comparison because of its rich PE ↔ PE communication capabilities and its relative lack of a Host ↔ PEs connection network. Figures 1 and 2 depict system level layouts of the Connection Machine and the Data Structure Machine architectures.

2. APSA Data Structure Memory Overview

The original APSA architecture was designed for efficient implementation of list processing operations by recognizing that most of these operations involve Host ↔ PE
communication and computation [4,5]. In this context, the major function of the APSA communication structure is to support efficient operators that 1) find, or mark, a list in the heap, 2) update marked lists, and 3) operate on marked lists as if they were compact linear data structures.

Two simple regular connection networks combine to support these three types of operations. First, a binary tree of processing elements connects a leaf level, formed by PE/ME pairs, to the Host processor connected at the root. The store of the machine is mapped into the leaf level of the tree. Only intermediate status information is stored in the non-leaf nodes of the tree. Using the accepted method of classifying massively parallel systems based on the connection topology of the PEs, the DSM would be classified as having a one dimensional array of processing elements. The binary tree network would be ignored.

A second network connects the leaf level horizontally across the breadth of the tree, allowing bidirectional shifting of data. This second, linear, connection topology implements limited PE ↔ PE communication, that when combined with the tree network can implement a total ordering on the data stored in the leaf level. In this context, a total ordering on the data items in the leaf level means that if a node, \( N_i \), occurs before a node, \( N_j \), in a traversal of the list, then the node \( N_j \) is physically stored to the left of node \( N_i \). The linear connections in the leaf level are used to effectively shift a leaf cell from the avail pool, in a single cycle, to a position that maintains the total ordering.

The process of shifting data cells can destroy the validity of explicit, address based, pointers in the heap, which could take significant time to update in a heterogeneous system. APSA solves this problem by using unique labels, instead of addresses, for explicit pointers. A pointer is dereferenced by having all cells perform a match function on the label. This content addressable pointer solution is efficient since updating of pointers is not necessary after a shift operation. A similar system can be implemented on the Connection Machine, but PE ↔ PE communications would then be based on a similar content addressable pointer system which would render most of the CM’s routing circuitry useless since explicit pointers would be dereferenced by content-based addressing, while the CM router only works with absolute location-based addressing. Though implementable, such a system wastes most of the bandwidth available, and paid for, on the Connection Machine. In other words, an efficient emulation of the DSM can be implemented on the CM, but fails to exploit the scaling advantage of the tree topology and wastes much of the CM’s resources. For the purposes of the comparison in this paper, the CM is assumed not to be running in a DSM emulation mode.

### 3. Timing guidelines for SIMD algorithms

It is necessary to clarify the guidelines used for timing comparisons of algorithms running on von Neumann architectures with those running on SIMD architectures. This clarification is needed because of the dual use of the time complexity of a memory access/update operation on RAM. Any computer scientist asked to analyze the time complexity of the FETCH or STORE instructions for RAM will quickly and easily return the answer \( O(\log(n)) \). However, in analyzing algorithms running on these machines, analysts invariably consider the time complexity of these instructions to be \( O(1) \). Historically, this shortcut was taken because all algorithms were run on von Neumann machines; by eliminating this extra \( \log \) from all equations, the comparison between algorithms was clearer. This technique has become so widely accepted that many computer scientists forget that RAM instructions are not constant in time. Several critics of the tree topology have wrongly pointed out that the tree must pay a \( \log \) cost for sweeps up or down the tree. The router in the Connection Machine has probably received similar baseless criticism. In both of these cases the logarithmic delay is analogous to the logarithmic delay for address decoding in RAM. A fair comparison will treat time through the CM router and up or down the APSA tree as constant time operations, asymptotically equivalent to FETCH and STORE times.

---

**Figure 1.** Connection Machine high level architecture

**Figure 2.** DSM high level architecture
4. Examples

In the limited space this short paper provides I will present an analysis of two sample algorithms that realize a performance gain by utilizing the Host ↔ PEs network of the APSA DSM. The first example shows an algorithm to sum the elements of an array. The CM algorithm has appeared [2], and works by building a tree, using pointers, within the array to calculate and store intermediate results. This example illustrates the use of the APSA DSM tree as a purely computational network. The second example computes the sum of the contents of a linked list structure and illustrates the value of being able to utilize a total ordering to support implicit pointers. For these examples let \( n \) be the number of elements in the array or list, and \( e \) be the number of explicit pointers in the CM list. Note that \( e \) would be 1 for an APSA DSM list.

**Example 1 - Summing elements of an array**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Mark array in PEs</td>
<td>( 1 )</td>
</tr>
<tr>
<td>2. Build pointer tree</td>
<td>( \log(n) )</td>
</tr>
<tr>
<td>3. SUM by pointers</td>
<td>( - )</td>
</tr>
<tr>
<td>4. SUM by network</td>
<td>( - )</td>
</tr>
<tr>
<td>Total Algorithm</td>
<td>( \log(n) )</td>
</tr>
</tbody>
</table>

**Example 2 - Summing elements of a list**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Mark list in PEs</td>
<td>( e )</td>
</tr>
<tr>
<td>2. Build pointer tree</td>
<td>( e + \log(n) )</td>
</tr>
<tr>
<td>3. SUM by pointers</td>
<td>( - )</td>
</tr>
<tr>
<td>4. SUM by network</td>
<td>( - )</td>
</tr>
<tr>
<td>Total Algorithm</td>
<td>( e + \log(n) )</td>
</tr>
</tbody>
</table>

The worst case time for the CM algorithm in Example 2 is \( O(n) \) for a severely fragmented list. Also the programmer may choose, for other efficiency reasons, not to implement all APSA DSM pointers implicitly. It is possible to maintain a balance between the number of explicit pointers and sharing between data structures. This can be accomplished under programmer control at runtime. These examples also assume that the CM is not running as a virtual APSA DSM through emulation routines. As mentioned previously, such an emulation could not use the CM's address based routing circuitry for general PE ↔ PE communication.

The decrease in the performance of the CM algorithm for linked lists compared to arrays is the result of the CM's inability to maintain locality within dynamic data structures. An array can be thought of as a list with predefined length and all pointers represented implicitly. The CM can store and efficiently operate on this structure, however, because the CM utilizes absolute location-based addressing, and therefore cannot efficiently shift large blocks of data, it must represent updates to lists using explicit pointers. Dereferencing explicit pointers serializes data parallel operations and accounts for the decrease in performance.

5. Other Architectures

The DSM unit of the APSA machine is the only architecture described in the literature that devotes significant power and circuitry to a structure that is not part of the memory map. Other tree-based topologies have been described, yet in each case long-term storage is allowed at all levels. At first glance, this architecture may seem similar to the FFPM [3] since it is a tree based topology that uses a different design for processors for the leaf level. However, there are two major distinctions. First, the granularity of the FFPM is much bigger than that of the DSM; FFPM processors are approximately two orders of magnitude bigger than DSM processing elements. Second, the tree network is not used as an interface to a host. In fact the FFPM is an MIMD machine that could potentially operate without a traditional von Neumann host.

6. Conclusions

It is clear that characterizing a massively parallel processor solely on the connection topology between processing elements is not sufficient. In this paper the linear connection topology of the Data Structure Machine's processing elements is able to outperform the binary n-cube topology of the Connection Machine processing elements by utilizing an additional connection network to the host. In this case a binary tree network is able to maintain and exploit locality properties within data structures stored in the array of processing elements at the leaf level.

Just as von Neumann machines may be optimal for 1 to 1 or few to few operations, and general communications networks in massively parallel computer systems, such as the Connection Machine, may provide great speedups for many to many and many (1 to 1) operations, this work shows that a limited topology, such as a tree, can yield improvements for classes of 1 to many and many to 1 operations. Algorithms that operate on data structures that maintain a locality property are good candidates for inclusion in this class, and can find an asymptotic speedup on the DSM. We have recently extended this class to include nested relational databases, circuit simulation, and low-level image analysis operations. Further, the scalability of the machine can lead to an extremely large PE per chip ratio. This development would allow a moderate number of processors in a very small space or could be used to build a machine with an enormous number of processors.

Acknowledgements

I would like to thank John T. O'Donnell who developed the architecture and the programming paradigm for the Data Structure Machine, and who shared them with me as my research advisor.
References


458