Highly Parallel Processing of Relational Databases (Thesis)

Ching-Chih Hsiao

TR #460

HIGHLY PARALLEL PROCESSING OF RELATIONAL DATABASES

A Thesis

by

Ching-Chih Hsiao

Dedicated to my parents, my wife, and my family

This work is part of the Blue CHiP Project. It is supported in part by the Office of Naval Research Contracts N00014-80-K-0816 and N00014-81-K-0360. The latter is Task SRO-100.

ACKNOWLEDGEMENTS

Deepest appreciation is expressed to my major professor, Lawrence Snyder. His encourgement and guidance in my research and writing has been invaluable. I am very grateful to Professors Janice Cuny, Dennis Gannon, and Vincent Shen for serving on my graduate committee and their help throughout this research work. I am also indebted to Jeremy Epstein for some early, stimulating discussions.

Thanks are also extended to Professor S. Bing Yao who was my advisor before he left Purdue and Tom Putnan who was my supervisor when I worked at CINDAS.

Many thanks go to all the friends that have made my stay in West Lafayette so enjoyable, especially Kye Hedlund, Tom Rafetto, Steve Thebaut, Andrew Wang, and all the Blue CHiPpers. I would also like to thank Julie K. Hanover, the secretary of the Blue CHiP project.

Last but not least, I want to thank my parents, my brothers, and my sister for their support. My wife Nien-Tsu also deserves special thanks for her love and understanding.

This research is partially supported by the Office of Naval Research under Contract N00014-80-K-0816 and Contract N00014-81-K-0360, Special Research Opportunities Program Task SRO-100.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Ching-Chih Hsiao, Ph.D., Purdue University, December 1982. Highly Parallel Processing of Relational Databases. Major Professor: Lawrence Snyder.

New computer architectures are feasible because of the advances in VLSI design and fabrication technologies. Among them, highly parallel structures coordinate hundreds of thousands of processing elements that function cooperatively. These structures are especially useful in solving computationally intensive problems. This thesis applies the highly parallel approach to improve the efficiency in processing relational database queries. High-performance algorithms for basic relational operations are explored. Efficient composition of these algorithms to process whole queries is also investigated.

Regularity and uniformity are necessary in order to make the highly parallel computing cost-effective. An efficient primitive, called POP-SORT, is proposed to unify the relational operations such as sorting, duplicate-removal, union, intersection, and difference. The three latter operations are even allowed to have multisets as operands. POP-SORT is based on an easy scheme which adapts any highly parallel and regular sorting algorithm to perform all these database operations. The primitive is compared favorably,

This work is part of the Blue CHiP Project. It is supported in part by the Office of Naval Research Contracts N00014-80-K-0816 and N00014-81-K-0360. The latter is Task SRO-100.

in terms of time complexity, with existing algorithms for the five operations. The optimality of POP-SORT is also proved for a restricted but reasonable type of parallel computation. Furthermore, sublinear time performance is possible for join operations if argument relations are preconditioned by POP-SORT.

For processing a whole query, the operation tree parsed from the query can be executed by composing individual algorithms for the operations. The Configurable, Highly Parallel (CHiP) computers have the flexibility to provide programmable processor interconnections for composing algorithms. Query embedding is a method of executing whole operation trees to explore maximum parallelism on the CHiP computers. It involves the processor allocation and the embedding of appropriate interconnections. With the bitonic POP-SORT, which is a generalization of Batcher's bitonic merge sort, the query embedding can be simplified significantly.

CHAPTER 1

INTRODUCTION

Computer architects have been attempting to avoid the von Neumann structure that a single CPU serially fetches, processes, and restores data items. Due to the advances of VLSI fabrication and design technologies, computer architectures are no longer strictly confined by the cost of computing hardware. In the near future it will be feasible to implement highly parallel computers consisting of hundreds of thousands of processing elements [Hayn82]. With the use of so many processing elements operating cooperatively, a speed-up ratio as substantial as many orders of magnitude is possible.

The highly parallel structures are known to be useful for solving some computationally intensive problems in the areas like meteorology, cryptography, image processing, ...etc. However, integrating many processing elements to implement a reliable and cost-effective system is extremely difficult. Problems suitable for highly parallel computing must show a high degree of regularity and uniformity.

Relational data model [Codd70] not only provides a simple view of databases but also calls for a particular feature named relational processing capability [Codd82]. This feature entails the definition of relational operations which treat whole relations as operands. It is of interest to study the application of highly parallel architectures and algorithms to the implementation of relational operations.

Historically, efficiency of database processing has been stressed, but convenience and expressiveness have been of less concern. Application programmers' productivity is thus far behind the demands from end users of database systems. A relational data model, by raising the user interface from physical details to a higher logical level, provides improved convenience and expressiveness. E. F. Codd [Codd82] also remarked that the relational processing capability is a key factor leading the relational model toward a practical foundation for improved productivity. It is therefore very important to implement a relational processing capability that achieves high performance.

1.1 Goal and Methodology

The goal of this work is to take advantage of the VLSI computation power and the highly parallel architectures to improve relational database processing. We are concerned both with high-performance implementations of individual relational operations and efficient processing of whole queries.

Highly parallel computing relies crucially on efficient communication to achieve a successful exploitation of parallelism. For solving problems with parallel computation, more communication time is often required than the actual computation time [Lint81]. Processor interconnections, hardwired or software-controlled, on highly parallel computers are usually selected to support efficient communication. Therefore, it is important to identify communication schemes which are efficient for solving many problems.

Sorting is a necessary operation in many applications. Highly parallel sorting has been vigorously studied and several efficient algorithms exist [Batc68, Ston71, Thom77, Nass79; Mull75, Hirs78, Prep78]. For highly parallel processing of relational databases, we unify several operations on a single communication scheme by reducing those operations to sorting. The primitive operation POP-SORT (Primitive OPeration SORT) is thus proposed for the database operations such as sorting, union, intersection, difference, and duplicate-removal. We also apply POP-SORT to solve join operations in sublinear time.

POP-SORT presents the possibility of adapting any sorting algorithm to become a primitive for the five database operations. For merge-oriented sorting methods, the adaptation can be easily done by replacing the simple comparison function with a slightly modified one. The simple comparison function is extended to have marking capability that marks one of the two argument items when they are found to be equal. Comparison functions acting only in the local computation at processing elements do not effect the communication among the processing elements at all. For sorting methods in general, the adaptation can be done by two marking processes that both take constant time. The marking processes require communication only as simple as a linear array.

The efficiency of POP-SORT in performing the five database operations is demonstrated by an instance called the bitonic POP-SORT. It is a generalization of Batcher's bitonic merge sort [Batc68]. The performance of the bitonic POP-SORT compares favorably with existing algorithms (upper bounds) for the five database operations. To further evaluate the optimality of POP-SORT, we look into the reducibility relationships between it and the database operations.

The CHiP (Configurable Highly Parallel) computers are capable of providing dynamic and programmable interconnections [Snyd82]. It is thus possible to embed required connections for processing whole queries. To expand the spectrum of parallelism to process whole queries, we explore the feasibility of the query embedding on the CHiP computers. In [Snyd82] Snyder showed that the CHiP computers have the flexibility to compose algorithms to solve large and computationally intensive problems. Employing the bitonic POP-SORT as a primitive for several database operations, the composition of algorithms to process whole queries can be simplified significantly.

1.2 Definitions and Notation

A relation is normally a set of unique tuples and each tuple consists of an ordered sequence of components. As duplicates are artifacts of certain relational operations, we allow relations to be multisets consisting of duplicate tuples. Basic relational operations like sorting, restriction (selection), join, Cartesian product, and quotient are defined as those in text books (see, for example, [Ullm80]). Projection, duplicate-removal, union, intersection, and difference are defined slightly differently in this work.

For *remove-duplicates* we do not insist on discarding the duplicate items. Given n data items $x_0, x_1, \ldots, x_{n-1}$, the goal of duplicate-removal is to compute the mark bits $\mu(0), \mu(1), \ldots, \mu(n-1)$ for these items. In the sequence $x_0^{\mu(0)}, x_1^{\mu(1)} \ldots, x_{n-1}^{\mu(n-1)}$, we distinguish $x_i^{\mu(i)}$ as a duplicate item if $\mu(i) = 1$. An additional operation *segregation* can be used to pack and separate marked and unmarked data items in the sequence [Schw80]. To perform *projection* on a relation, we assume that duplicate-removal is not automatically invoked. The operations *union*, *intersection*, and *difference* may be relaxed to allow multisets as operands. Without further notice, they are just set operations as usual.

A highly parallel processor is a processing device which integrates many processing elements. By "processor" we may refer to a single processing element or a system of coordinated processing elements. Usually it means a processing element unless further indicated by the context. For example, the CHiP "processor" is a "highly parallel processor" in the collective sense.

The following notation is used throughout this thesis.

$\log^x y$	the base two logarithm $(\log_2 y)^x$.
$\lceil x \rceil$	the least integer greater than or equal to x.
$\lfloor x \rfloor$	the greatest integer less than or equal to x.
t_R	time required for one data routing step.
t_C	time required for one comparison step.
PE	processing element which may have some local memory.
$A \cup B$	the union of two sets A and B.

$A \cap B$	the intersection of two sets A and B.
$A - B$	the difference of two sets A and B.
$union(A,B)$	the union of two multisets A and B.
$inter(A,B)$	the intersection of two multisets A and B.
$differ(A,B)$	the difference of two multisets A and B.
$rmdup(A)$	the duplicate-removal on multiset A.

1.3 Organization of the Thesis

In Chapter 2, we look at the conventional approaches of database machine designs. The conventional approaches do not solve the compute-bound operations satisfactorily. Several highly parallel structures for solving the compute-bound operations are thus proposed by researchers. We also discuss those structures and the algorithms proposed to be executed on them.

Chapter 3 presents a methodology to apply parallel sorting to solve other problems. By reducing union, intersection, difference, and duplicate-removal to sorting, these operations are unified by the primitive operation POP-SORT. Two adaptations are shown to extend merge-oriented and other sorting methods to become POP-SORT. The adaptation overhead is shown to be negligible. We also show that POP-SORT can be used to perform join operations in sub-linear time. This application of POP-SORT is especially suitable for easy join operations that produce only small result relations.

The efficiency of POP-SORT is investigated in Chapter 4. A complexity hierarchy showing the reducibility relationships among POP-SORT and the five database operations is first established. The complexity hierarchy indicates that the optimality of POP-SORT relies on the reducibility of sorting to duplicate-removal. We therefore look into the reducibility of sorting to duplicate-removal by considering two types of comparison functions, the weak comparison $(=, \neq)$ and the strong comparison $(<, =, >)$.

Chapter 5 deals with some interesting aspects of performing the bitonic sort with the mesh interconnection on the CHiP computers. We design an efficient algorithm that rearranges n sorted data items among three major indexing schemes in less than $(3\sqrt{n})\, t_R$ time. Sorting with shadow regions is a technique that allows the allocation of exactly n processing elements for sorting n data items (n is an arbitrary integer). We also demonstrate how data communication can be improved by properly programming the switching elements on the CHiP computers. Two different methods of sorting $k * n$ data items on a CHiP region of n processing elements are also analyzed.

Processing whole queries on the CHiP computers is the subject of Chapter 6. Relational algebraic queries are considered. The idea of embedding whole operation trees parsed from database queries is explored. With the bitonic POP-SORT, we demonstrate that query embedding is simplified significantly. We also discuss several optimization strategies to improve query embedding on the CHiP computers.

CHAPTER 2

HIGHLY PARALLEL DATABASE MACHINES

Database machines are specialized computers dedicated to executing database management functions. They are usually connected to general-purpose computers as back-end machines. If a database machine is enhanced with a highly parallel processor to solve compute-bound database operations, we call it a highly parallel database machine. In Figure 2-1 we show the configuration of a back-end system consisting of a highly parallel database machine.

In the back-end system, the host computer acts as the interface between users and the database machine. It is responsible for taking users' requests, translating the high-level data manipulation programs into database machine commands, instructing the database machine to perform the commands, and returning the response to the users. Besides the highly parallel processor, there are two major components in the database machine: the back-end controller and the mass storage. The back-end controller serves as the interface to the host computer. The mass storage is content addressable in order to perform searching and update operations as well as other I/O-bound database operations efficiently. Between the mass storage and the highly parallel processor there is a wide data channel to support rapid data loading and unloading. This bandwidth is also needed in associative processor systems [Berr79] and array processor systems [Batc80].

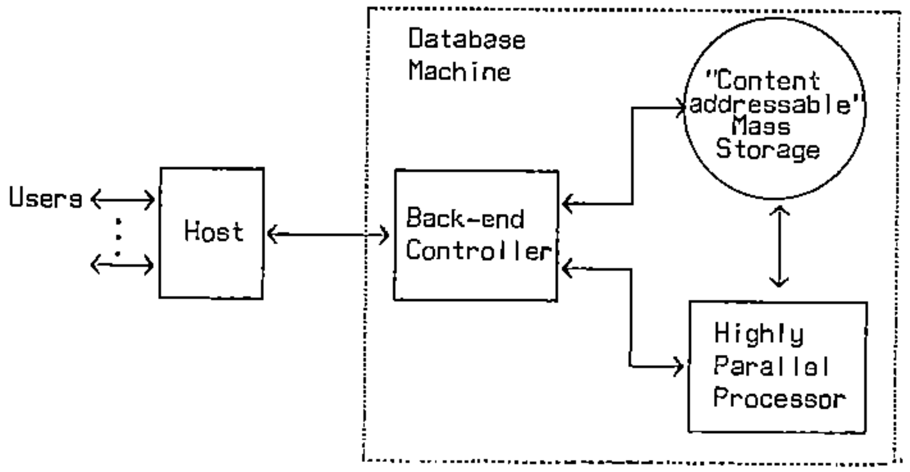


Figure 2-1. The system configuration of highly parallel database machines.

This chapter presents a brief overview of the principal approaches in conventional database machine designs. The inability of conventional approaches to solve compute-bound database operations is discussed. Highly parallel processors are then proposed as a means of extending the computation power of database machines. Next, we review some highly parallel structures and their algorithms that have been reported to be useful for database applications. All this serves as a benchmark for evaluating our research work.

2.1 Background

As database management techniques are shown to be helpful, users want them to be larger and more inclusive. But as databases become progressively larger, conventional general-purpose computers fail to meet the response time requirements of many applications. With the adoption of high-level data models and data manipulation languages, high-performance implementation of database management systems becomes even more crucial. Two well-known implementations of relational database management systems, System R [Astr76] and INGRES [StoB76], amply demonstrate the complexity and difficulty of query processing under these circumstances.

Since software techniques on conventional, general-purpose computers cannot implement database management systems efficiently enough, researchers have turned to alternative computer architectures and special-purpose hardware. Canaday [Cana74] proposed that database management functions be placed on a dedicated back-end processor which has exclusive access to the database. By limiting the back-end processor to the performance of only database management functions, it can have the advantage of efficiency through specialization. But the implementation of the eXperimental Database Management System (XDMS) [Cana74] failed to show that the use of a general-purpose computer as back-end is a good approach. Specialized database machines are, therefore, designed to serve as the back-end computers [Bane79, DeWi79, Schu79].

Many hardware organizations have been proposed to facilitate database processing although they are not all complete designs of database machines. Two objectives are involved. One is to improve the non-query aspects of processing such as searching, retrieval, insertion, deletion, and modification. The other is to speed up the query aspects of processing which may involve some compute-bound operations.

There is a consensus that content addressable memory is desirable for efficient searching and updating. But storing databases entirely in associative memory is infeasibly expensive. Fortunately, the "logic-per-track" approach proposed by Slotnick [Slot70] provides a practical solution for implementing a large-volume memory with content addressability. Many designs have applied some type of the logic-per-track approach to achieve the associativity and parallelism for fast searching and updating [Lang78]. Among them are the Content-Address Segment Sequential Memory (CASSM) [Su75, Su79], the Content Addressed File Store (CAFS) [Babb79], the Data Base Computer (DBC) [Bane78, Bane79], the Relational Associative Processor (RAP) [Ozka75, Schu79], and the Rotating Associative Relational Store (RARES) [Lin76].

One useful strategy to reduce the overhead of data movement is to process data in place if it is possible. By placing some processing capability at the mass storage level, the logic-per-track approach performs not only searching and updating effectively but other operations as well. I/O-bound relational operations like restriction and projection (without removing duplicates) can be performed at the memory level. Other operations, however, are not easily supported [Song81, DeWi82]. Sorting, duplicate-removal, union, intersection, difference, join, and Cartesian product all require that one data item interact with many others. These operations require complex processor interconnections that cannot be easily implemented using the logic-per-track approach. This is because of the physically dispersed character of the read/write heads. Implementing these operations on the secondary storage level, it seems to require some kind of looping or iteration.

Several techniques help to improve query processing on compute-bound operations. The overhead incurred by the time-consuming secondary memory accesses can be reduced by using intelligent file systems and memory management. Unnecessary database information can be filtered out before it is submitted to the processor. The use of special processing devices is yet another weapon with which researchers attack the compute-bound problems. Much special-purpose hardware has been proposed for performing the operations join and sorting. In addition, in the DBC design several compute-bound functions or "post-processing functions" [HsiD79] are performed by a multiprocessor system. These post-processors are linearly connected, and each has its own local memory. Also in [DeWi79] a multiprocessor architecture called DIRECT was designed to support relational query processing.

Special hardware for a few operations respectively do not solve the problem completely or uniformly. The multiprocessor systems proposed demonstrate reasonably good, but restricted, performance improvement. Application of highly parallel processors has thus been proposed for database processing [Kung80, Song80, HsiC81, Lehm81].

2.2 Highly Parallel Processors

A highly parallel processor may consist of hundreds of thousands of processing elements which function cooperatively to solve compute-bound problems. The computation power of the processing elements is limited to that required by database management queries. The instruction set is thus small and can be tuned to perform query processing more efficiently. When the highly parallel processor is implemented by VLSI chips, less area for computing logic implies that more area can be dedicated to the local memory logic or the processor interconnection circuitry. Being more important, a larger scale integration of processing elements is possible if more chip area is available for processor interconnections.

In highly parallel structures, inter-processor communication is the key to successful exploitation of the available computing power. The processor interconnection problem has motivated much research recently. An important question that needs to be addressed for general computation and data processing alike is:

> *What are the most effective interconnection paths for communicating PEs to process database queries*?

This section discusses several structures of highly parallel processors and their algorithms. The highly parallel processors addressed here are: the systolic array system, the double tree machine, the Ultracomputer, and the CHiP computer. The first three represent different processor interconnections, and the last one has the flexibility to provide them (as well as the mesh interconnection).

In Table 2-1 we first summarize the time complexities of certain database operations on these machines. POP-SORT is the primitive operation proposed in this thesis which can perform the other five operations (Chapter 3). The complexity is measured by assuming that the argument relations have n tuples. Except for the systolic arrays and the tree machine, we assume that the data is already in the processing device. The effect of propagation delay is ignored here for the tree machine and the Ultracomputer.

Table 2-1. Algorithms of database operations on highly parallel machines.

operations	$\cup$	$\cap$	$-$	rmdup	sort	POP-SORT [*]
-						
Systolic arrays	$O(n)$	$O(n)$	$O(n)$	$O(n)$	-	-
Tree machine	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Mesh computer	-	-	-	-	$\Theta(\sqrt{n})$	$\Theta(\sqrt{n})$
Ultracomputer	$O(\log^2 n)$	$O(\log^2 n)$	$O(\log^2 n)$	-	$O(\log^2 n)$	$O(\log^2 n)$
CHiP	-	-	-	-	-	$O(\frac{\sqrt{n}}{s})$ [**]

* An instance of POP-SORT, the bitonic POP-SORT, is used to calculate the time complexities (Chapter 3.1).

** A technique is applied on the CHiP computers to achieve the speed-up factor s over the mesh-connected computers (Section 5.3), where $s \leq w*c$.

Systolic Arrays

Systolic arrays have been proposed for many applications [Kung79, Fost80, Kung82]. Kung and Lehman [Kung80] used systolic arrays to implement relational database operations. Lehman [Lehm81] also applied systolic arrays to processing simple queries.

They presented two types of systolic arrays to implement database operations (Figure 2-2). A two-dimensional comparison array and a one-dimensional accumulation array were used for union, intersection, difference, and duplicate-removal. The comparison array alone is used for join operations. Argument relations are "staged" into the comparison array in a component-parallel and tuple-serial fashion. Tuples from different relations flow in the opposite directions in the comparison array so that they will always pass by each other. The comparison results move from left to right. They are recorded as a bit matrix for join or shifted to the accumulation array to generate a bit string for the other operations.

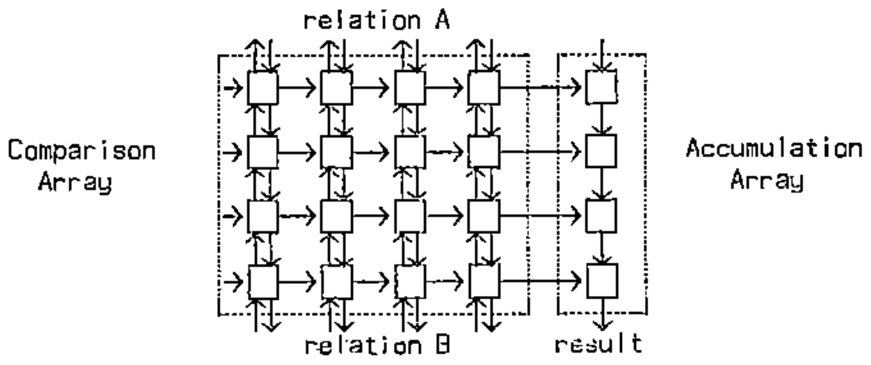


Figure 2-2. The systolic array system for performing database operations.

In the systolic arrays, the processing elements perform only simple functions and the interconnections are very regular. Both of the arrays can be implemented with only a few types of simple cells. Another advantage is that computations are pipelined elegantly so that the processing time is completely overlapped with the I/O time. However, from an algorithmic point of view, the benefit of data ordering is totally ignored in [Kung80]. The systolic arrays are fundamentally structures of linear time performance.

Systolic arrays are algorithmically specialized processors [Snyd82]. The functions performed by systolic arrays are predetermined and rigidly manufactured into VLSI products. Programmability is minimal. To implement all the operations required by query processing, an integrated system containing several systolic arrays is needed [Song81].

The BK-tree Machine

The BK-tree, or double tree, was proposed by Bentley and Kung [Bent79] for pipelining searching operations such as retrieval, insertion, deletion, and modification. On an n-processor version of this machine, a set of n data items can be maintained such that all the searching problems are processed in $2\log n$ steps. Tree-structured machines have also been proposed as general-purpose processing devices by Browning [Brow80] and many other researchers. In [Song80, Song81] this architecture was applied to implement additional basic database functions.

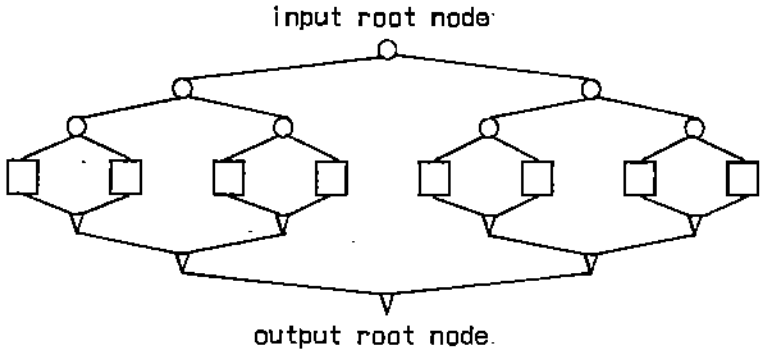


Figure 2-3. The BK-tree machine.

A BK-tree† machine is composed of three kinds of processing elements: O-nodes, []-nodes, and V-nodes (see Figure 2-3). The []-nodes contain the data items to be processed. The O-nodes are responsible for broadcasting sequences of instructions and data to the []-nodes. Parallel computation is carried out by the []-nodes. Partial results produced are then collected by

† An interesting interpretation of "BK" is that "B" is mnemonic for broadcasting information and "K" for collecting information.

the ∇-nodes. At last the final result emerges from the output root node.

Sorting can be easily implemented on a tree machine using the heap sort algorithm [Mead80]. To perform union, intersection, and join on relations A and B, Song employed two solutions [Song80]. One is to sort the two argument relations using the tree machine and to perform further processing elsewhere. The other is to load one relation in □-nodes and then broadcast the other relation onto the □-nodes to perform the required operation. Partial results produced in the □-nodes may have to be saved before they can be accepted by the ∇-nodes (e.g. in performing join). The potential bottlenecks were resolved by a request/acknowledge communication convention [Song80].

The BK-tree machine is very efficient in pipelining successive searching operations which take a single data item as the operand. However it does not perform as well on database operations which take whole relations as operands. Again, the BK-tree machine is fundamentally a linear time bounded structure. The performance barrier is inherited from the general restriction of tree structures that only one data value at a time can flow into and out of the tree through the root node. Furthermore, the VLSI layouts of large trees are susceptible to the propagation delay problem [Pate81].

The Ultracomputer

Ultracomputers [Schw80] are those with powerful and physically realized interconnection patterns. They are composed of a large number of processing elements each connected with a fixed number of others. The Ultracomputer in [Schw80] is based on the perfect shuffle interconnection

[Ston71]. Other powerful interconnections like the Cube-Connected-Cycles (CCC) [Prep81] are also in this category which we refer to as ultracomputers.

On the Ultracomputer with the perfect shuffle interconnection, all the permutations of data among processing elements can be realized in $\log n$ routing steps. Sorting, union, intersection, and difference can thus be solved in logarithmic time. No results about duplicate-removal and join are reported in [Schw80]. While the ultracomputer is efficient in solving certain compute-bound operations, it is expensive to implement. Expandability is poor because the interconnection complexity grows at least as a function $n^2/\log^2 n$ of the number of processing elements n [Thom80]. Moreover propagation delay problems and synchronization difficulties can become more severe when n is large.

The CHiP Computer

A Configurable, Highly Parallel (CHiP) [Snyd82] processor permits the processor interconnections to be dynamically programmed. It does not limit the communication to one fixed structure among the processing elements. Nor does it rely on a single interconnection capable of simulating others to achieve the flexibility of communicating processing elements. It provides a lattice of programmable switching elements with which dynamic and flexible interconnections can be specified.

The processing elements are connected to the switch lattice at regular intervals. The interval determines an important parameter w of the switch lattice which is called the corridor width. Two more parameters of the switch lattice which are important to this research work are the degree (or the number of incident data paths) d and the cross-over capability c of the switches. The cross-over capability denotes the maximum number of independent data paths that can pass through a switching element. In Figure 2-4 we show two structures of the switch lattice. The circles represent switches and the squares represent processing elements.

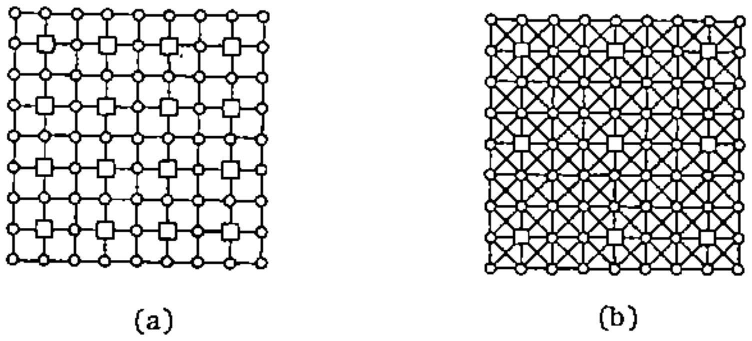


Figure 2-4. Two structures of the switch lattice: (a) $w = 1$, $d = 4$; (b) $w = 2$, $d = 8$.

At each switching element there is some local memory for storing a fixed number of switch settings. The controller broadcasts a command to the switches and the switches then make connections according to a particular switch setting stored. The total effect of making connections at the switches constitutes the designated interconnection. The processing elements then communicate with each other assuming that the right interconnections are realized by the switches. (See [Snyd82] for more detailed description of the CHiP computer; See [Snyd81] for a discussion of programming processor interconnections.)

The CHiP computer can be easily configured to be a mesh-connected computer. With the mesh interconnection, sorting can be done in $O(\sqrt{n})$ time using adapted algorithms of Batcher's bitonic sort [Batc68, Kung77, Nass79]. In Chapter 3 we shall present a primitive operation POP-SORT which can perform the five database operations listed in Table 2-1. POP-SORT does not require the special architecture of the CHiP computer. On the contrary, it represents a methodology of applying parallel sorting to solve other database operations. POP-SORT can be implemented on the tree machine, the Ultracomputer, the mesh-connected computer, and the CHiP computer. If sorting can be implemented with systolic arrays then the systolic arrays can also be easily modified to implement POP-SORT.

CHAPTER 3

AN EFFICIENT PRIMITIVE OPERATION

Highly parallel algorithms for database operations have been widely studied. Several algorithms that perform sorting in sub-linear time exist [Batc68, Ston71, Thom77, Nass79; Mull75, Hirs78, Prep78]. The set operations union, intersection, and difference are best solved by performing sorting first [Schw80]. For duplicate-removal and join, there are linear-time bounded algorithms [Kung80, Song81]. Mentioned above are different algorithms and different machine architectures (see Table 2-1).

VLSI implementation of specialized devices has been vigorously proposed [Kung79, Fost80, Kung80, Kung82]. However, cost-effectiveness of VLSI implemented systems depends fundamentally on regularity and uniformity. The initial development expenses of VLSI systems must be offset by volume production. Thus, for VLSI implementation of highly parallel versions of database operations, it is important to identify a nucleus of processing steps common to the many database operations.

On general-purpose, highly parallel computers, programmability of algorithms again depends on regularity and uniformity. It is extremely expensive to develop software for highly parallel computers. Therefore, for performing database operations on highly parallel computers, it is also important to identify an efficient primitive operation.

Much work on highly parallel sorting has been reported and has demonstrated some efficient solutions [Batc68, Ston71, Thom77, Nass79; Mull75, Hirs78, Prep78]. To identify primitive processes for database operations, we thus apply algorithmic approach to reduce many database operations to a sorting-based primitive. Whatever sorting algorithm and machine architecture are chosen, we then always have a unified treatment of those operations by implementing them with the primitive operation.

In this chapter we shall present POP-SORT (Primitive OPeration SORT) as a primitive operation for sorting, duplicate-removal, union, intersection, and difference. The latter three operations are relaxed to have multisets as operands. This relaxation, surely based on the versatility of POP-SORT on the one hand, has much practical merit in the context of query processing on the other hand. For natural join and equi-join, sub-linear time algorithms are possible if relations are preconditioned by using POP-SORT.

In Section 3.1 we present a special family of POP-SORT which is based on merge-oriented sorting methods. Employing a new comparison function, any merge-oriented sorting method becomes POP-SORT. An efficient implementation of the new comparison function and the overall performance of the POP-SORT are shown in Section 3.2. In Section 3.3 we present a general adaptation scheme that modifies any sorting algorithm to become POP-SORT. We then show the application of POP-SORT to the natural join and equi-join operations in Section 3.4.

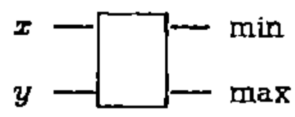
3.1 POP-SORT, a Special Example

Among the fast and highly parallel sorting algorithms, we are most interested in constructive, potentially logarithmic time, and non-probabilistic algorithms. There are two categories of comparison-based sorting algorithms that rely fundamentally on pairwise comparisons. One category can be modeled as sorting networks [Knut73, p.220] that are constructed from comparator modules [Batc68, Ston71, Thom77, Nass79]. The other is based on the enumerating comparison method that each item is compared with each of the others [Mull75, Prep78]. While the former is conjectured to require $O(\log^2 n)$ levels of network depth, the latter is able to reduce the time complexity to $O(log\ n)$. However a considerable drawback with the enumeration sort is the requirement of $O(n^2)$ computing components or the assumption of a shared, random access memory.

Batcher's bitonic merge sort [Batc68], described as a sorting network in Appendix A-1, is one of the most famous. There are many adapted versions of the bitonic sort. It requires $O(\sqrt{n})$ time using mesh interconnection [Thom77, Nass79] or $O(\log^2 n)$ steps using shuffle interconnection [Ston71]. The number of computing components needed for these adapted algorithms may be as small as $O(n)$.

In this section we shall present a special example of POP-SORT called the bitonic POP-SORT. This instance of POP-SORT uses a new comparison function in Batcher's bitonic sorting method. The scheme that adapts the bitonic sort to become POP-SORT relies on the merge-oriented nature of the bitonic sort. Therefore, the adaptation scheme is immediately extended to all the merge-oriented sorting methods.

Bitonic sort is based on a simple local operation together with a regular and efficient way of pairwise data communication (Figure A-1). The local operation is a simple comparison function which can be described as:



$$(x, y) \rightarrow (\min(x, y), \max(x, y));$$

$$\text{when } x = y, \min = \max.$$

The communication scheme, from another point of view, is actually a sequence of perfect shuffle on different numbers of data items. Perfect shuffle is so powerful that it can simulate many important communication functions in time proportional to the logarithm of the number of data items [Schw80]. It should be able to solve other database operations if the simple comparison is replaced by more sophisticated ones.

Definition The *compare-and-mark*$_1$ operation performs comparison as well as marking duplicates, and the marking process is idempotent:

(1) $(x, y) \rightarrow (\min(x, y), \max(x, y))$ when $x \neq y$;

(2) (x, x), (x^-, x), or $(x, x^-) \rightarrow (x^-, x)$ and

$(x^-, x^-) \rightarrow (x^-, x^-)$.

The basic operation *compare-and-mark*$_1$ preserves the ordering among distinct elements as usual. By marking a duplicate of x as x^- the basic operation enforces an ordering rule such that x^- is a little smaller than x but never smaller than any y for $y < x$. The ordering among the marked duplicates x^{-}'s is arbitrary. The marking capability of the basic operation is to magnify the computation power of the bitonic sort to performing duplicate-removal. Rather than proving this for the bitonic sort only, we would prove a more general application of *compare-and-mark*$_1$ to all the merge-oriented sorting methods in the following theorem.

Theorem 3-1. Using the *compare-and-mark*$_1$ operation, any merge-oriented sorting method can mark off all the duplicates.

[Proof] Consider any comparison-based method which merges two ordered sub-lists. Every pair of neighboring elements in the result list must have been compared directly, unless both elements are from the same sub-list. If both sub-lists have duplicates marked off before merge then the result list must have all the duplicates marked by using the *compare-and-mark*$_1$ operation. For any merge-oriented sorting method that starts with merging sub-lists of length one, it guarantees no duplicates at all in the very beginning. By induction, all the duplicates must have been marked off in the final sorted list. ■

In addition to performing duplicate-removal, any merge-oriented sorting method using *compare-and-mark*$_1$ is able to perform union. Performing union is the same as performing duplicate-removal on the totality of the two groups of data items. If our purpose is to unify sorting, duplicate-removal, and union then *compare-and-mark*$_1$ is powerful enough. However we are aiming at identifying a primitive for more database operations. Intersection and difference take two sets of data items as operands. One fundamental requirement is that we must be able to distinguish data items from the two groups in order to perform these two operations. We therefore extend the *compare-and-mark*$_1$ operation to handle two groups of data items.

Definition Let A and B be multisets, $a \varepsilon A$ and $b \varepsilon B$. The *compare-and-mark*$_2$ operation, in addition to performing the simple comparison, enforces marking duplicates and three ordering rules:

(1) Idempotent marking-minus:

(a, a), (a^-, a), or $(a, a^-) \rightarrow (a^-, a)$;

$(a^-, a^-) \rightarrow (a^-, a^-)$.

(2) Idempotent marking-plus:

$(b, b) \rightarrow (b^+, b)$, or $(b, b^+) \rightarrow (b, b^+)$;

$(b^+, b^+) \rightarrow (b^+, b^+)$.

(3) Quasi-stability:

(a, b) or $(b, a) \rightarrow (a, b)$ for $a = b$ (marked or unmarked).

Similar to that shown in Theorem 3-1 the marking capability of the *compare-and-mark*$_2$ extends the computation power of the bitonic sort to performing duplicate-removal and union. Moreover, two separate marking rules allow us to mark duplicates for two multisets separately. The rule of quasi-stability insists that A-elements precede B-elements if they all have the same value. With the local operation having two separate marking mechanisms and being quasi-stable, the execution of the bitonic sort will end up with a sorted sequence like $\ldots a^-a^-a^-a\ b\ b^+b^+\ldots$, where $a = b$. We then can detect and manipulate all the $..a\,b..$ pairs in constant time. The bitonic communication scheme together with the *compare-and-mark*$_2$

operation therefore can also implement the intersection and difference operations too. The three operations union, intersection, and difference are even relaxed to have multisets as operands. We have thus proved the following theorem.

Theorem 3-2. (POP-SORT) With the *compare-and-mark*$_2$ operation, any merge-oriented sorting method can be used for duplicate-removal, union, intersection, and difference.

How do we unify the operations that take a single multiset as operand and the others that take two multisets? It requires some initial processing on input operands. In algorithm 3-1, execution of the database operations is partitioned into three phases: initialization, primitive, and completion. The input to the algorithm may be one or two multisets. The input conflict is resolved in the initialization phase. Only in the completion phase may the database operations invoke different constant-time post-sorting processing. The output from the algorithm is that all the undesired data are marked off, either marked as x^- or x^+.

Algorithm 3-1: The bitonic:POP-SORT.

INTPUT: Data items from one or two multisets A and B.

OUTPUT: All the unmarked data items.

A. Initialization phase

1. Data items are arbitrarily labeled as A-elements for sorting, duplicate-removal, and union.
2. For intersection and difference, A-elements and B-elements are labeled differently in order to distinguish them throughout the whole processing.

B. Primitive phase

1. Run the bitonic sort using the ***compare-and-mark***$_2$ operation.

C. Completion phase

1. Remove-duplicates, sorting, and union do not need any further processing.

2. For intersection and difference, the constant-time processing in this phase is shown as a program segment in the following.

```
(* completion phase *)
for all $i$ do  (* $x_{n+1} = \infty$ is a dummy *)
    compare $x_i$ with $x_{i+1}$
    if both unmarked then
      case
        intersection: mark $x_i^-$;
                if not equal then mark $x_{i+1}^+$;
        difference: mark $x_{i+1}^+$;
                if equal then mark $x_i^-$;
```

The relaxation that union, intersection, and difference take multisets as operands of course relies on the versatility of POP-SORT. The practical consideration is that multisets are artifacts of operations such as projection and concatenation. Evidently many query languages (SEQUEL, QUEL, and QBE [Ullm80]) provide operators for working with multisets. On many occasions in database query processing, duplicate-removal and union (intersection, or difference) are executed subsequently. For example, projection is first requested before two relations are to be joined, $\Pi\,(R_1) \cup \Pi\,(R_2)$, where Π denotes projection. In order to perform the set operation $\cup$, duplicate tuples produced by the operation projection must be removed. We have the following:

$$union(A,B) = rmdup(A) \cup rmdup(B),$$

$$inter(A,B) = rmdup(A) \cap rmdup(B),$$

$$differ(A,B) = rmdup(A) - rmdup(B),$$

With the relaxation the two operations are combined together and a single run of sorting is enough. However, without the relaxation, performing the two operations sequentially is necessary. The sequential execution in this case may imply more data movement and programming overhead.

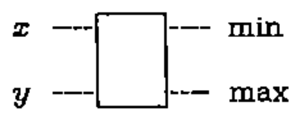
The result sequence could be sparse due to the marked-off duplicates. The marked-off duplicates can be filtered out while outputting the sequence. Alternatively, in some applications one might want to compress the sequence internally so that the marked duplicates are squeezed out. Schwartz presented an ingenious method to separate and pack marked data on the Ultracomputer in O(log$n$) time [Schw80]. If the shuffle-exchange interconnection is available the compression job can then be best done by Schwartz's pack algorithm. A desirable solution might be running POP-SORT again using another comparison function which treats the marked duplicates as $+\infty$.

3.2 Implementation and Performance

Different interconnection patterns among processing elements for the bitonic sort and their implementations have been reported in the literature [Batc68, Ston71, Thom77, Nass79, Schw80, Prep81]. The local operation at each processing element is the crucial part that may extend a merge-oriented sorting algorithm to perform other database operations. We shall consider only the implementation of the local operation in this section.

An efficient implementation of the *compare-and-mark*$_1$ operation uses one extra bit for marking. The mark bit, initially set to be 1, is appended to each data item as the least significant bit. The operation works simply to clear one least significant bit whenever two elements are found to be equal.

Similarly the *compare-and-mark*$_2$ can be implemented using two mark bits, one for distinguishing A-elements from B-elements and the other for marking duplicates. The mark bits are tagged to each data item as the two least significant bits. Let a and b be l-bit data items concatenated with the two mark bits, $a \varepsilon A$ and $b \varepsilon B$. Their binary representations are $(a_{l-1}, a_{l-2}, \ldots, a_1, a_0, a_{-1}, a_{-2})$ and $(b_{l-1}, b_{l-2}, \ldots, b_1, b_0, b_{-1}, b_{-2})$ respectively. Initially, we have the mark bits set in such a manner that $(a_{-1}, a_{-2}) = (0,1)$ and $(b_{-1}, b_{-2}) = (1,0)$. The *compare-and-mark*$_2$ function can be described as:



```
if $x = y$ then $x_{-2} \leftarrow x_{-1}$;
min $\leftarrow$ min$(x, y)$, max $\leftarrow$ max$(x, y)$;
```

The *compare-and-mark*$_2$ function can be interpreted more clearly using a state diagram as shown in Figure 3-1. Define the state of a data item as the value of its two mark bits. There are only four possible states, with (0,1) the initial state for all the A-elements and (1,0) for B-elements. The rule of marking-minus changes the state (0,1) to (0,0) for A-elements. Since the marking is idempotent, once a data item reaches the (0,0) state it remains in that state. The rule of idempotent marking-plus works in the same way for B-elements.

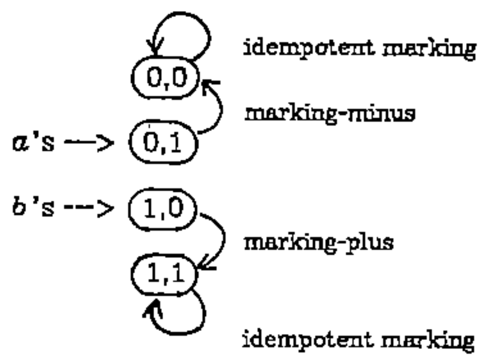


Figure 3-1. State digram for two idempotent marking functions.

After the completion phase of POP-SORT, all the desired data items may be in the state either (0,1) or (1,0). Suppose we arbitrarily choose (0,1) as the final state of all the desired data items. To separate and pack all the unmarked data using POP-SORT again, we need some more bit manipulation capability.[†] First, reset the states (1,0) and (1,1) to (0,0). We then may rotate each data item such that all the desired data has the most significant bit 1. Alternatively, we may design the second mark bit with some flexibility so that it may be programmably tagged to each data item as the least or most significant bit.

Several adapted versions of the bitonic sort show that more data routing time is required than comparison time. Suppose that a merge-oriented sorting algorithm takes $T_1(n) * t_R + T_2(n) * t_C$ time, where $T_1(n)$ is the number of data routing steps and $T_2(n)$ the number of comparison steps. The POP-SORT based on this sorting algorithm then requires $T_1(n) * t_R + T_2(n) * t'_C$ time. The only difference is the step size t'_C. That is,

[†] Unfortunately the bitonic sort is not stable. Otherwise, performing sorting on the two mark bits would be able to separate marked and unmarked data items.

the processing time for one local operation is changed. The marking function, on one or two bits, of the local operation usually takes less time than the comparison function (l bits). The ratio of t'_c to t_c is bounded by a small constant, actually close to one.

$$\rho \leq \frac{t'_c}{t_c} < 2$$

$$\text{where } \rho = \frac{l+2}{l} \text{ for bit serial design,}$$

$$\text{or } \rho = \frac{log(l+2)}{\log l} \text{ for bit parallel design.}$$

In summary, the bitonic POP-SORT, based on Batcher's bitonic merge sort, performs as well as the bitonic sort. It compares favorably with other algorithms known for the five basic database operations (Table 2-1). The bitonic POP-SORT outperforms Kung's and Song's duplicate-removal algorithms dramatically. For the other operations, we do not sacrifice any efficiency by using it. Since the bitonic POP-SORT serves as a primitive for many operations, the overall system performance may improve substantially (e.g. query embedding in Chapter 6). The program loading is no longer necessary for every single operation. Data movement can be reduced because data may stay longer for more processing.

3.3 POP-SORT, in General

Batcher's bitonic merge sort has been shown easily adaptable to become POP-SORT. The s^2-way merge sort performs even better than bitonic sort on a mesh-connected computer when the number of data items is large [Thom77]. According to Theorem 3-2, we already have the first order generalization that any merge-oriented sorting algorithm can employ the *compare-and-mark*$_2$ operation to become POP-SORT. Of course s^2-way merge sort can be another base sorting algorithm for POP-SORT. However, can we also adapt other sorting methods to become POP-SORT?

In this section, we shall show a general scheme to employ any sorting algorithm as a POP-SORT. The general scheme again involves extending some marking capability to a base sorting algorithm. In its most general sense, POP-SORT thus presents an idea to adapt any sorting algorithm to become an efficient primitive for many database operations.

The computation power of the basic operation *compare-and-mark*$_2$, in addition to the simple comparison function, comes from enforcing the ordering rules of quasi-stability, marking-minus, and marking-plus. For a sorting algorithm that is not merge-oriented, it might not be able to incorporate all the ordering rules into the comparison function. Nevertheless, given a sorted sequence of data items, a "shift-copy and compare" scheme, shown in Figure 3-2, is able to detect and mark all the duplicates. If the linear interconnection is available then the marking process requires only $O(1)$ time.

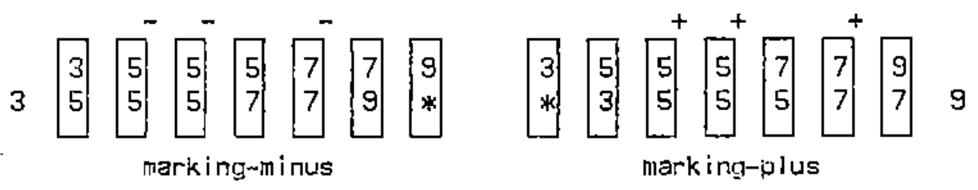


Figure 3-2. A "shift-copy and compare" scheme for detecting duplicates in a sorted sequence.

Suppose that newSORT is a new and faster-than-ever parallel sorting algorithm. Whether newSORT is merge-oriented or not, it can be adapted to become POP-SORT according to the general scheme described in Algorithm 3-2. The general scheme is composed of four phases: initialization, sorting, marking, and completion. A general POP-SORT is exactly the same as a merge-oriented POP-SORT in the initialization and completion phases. For a merge-oriented POP-SORT, the second and the third phases of a general POP-SORT is combined together due to the reinforced computation power of *compare-and-mark*$_2$.

Algorithm 3-2: A general POP-SORT.

INTPUT: Data items from one or two multisets A and B.

OUTPUT: All the unmarked data items.

A. Initialization phase

1. Data items are arbitrarily labeled as A-elements for sorting, duplicate-removal, and union.
2. For intersection and difference, A-elements and B-elements are labeled differently in order to distinguish them throughout the whole processing.

B. Sorting phase

1. Sort the data items, labeled as A-elements or B-elements, according to the quasi-stability rule using newSORT.

C. Marking phase

1. If not performing sorting then continue.
2. Mark duplicates according to the rules of marking-minus and marking-plus using the "shift-copy and compare" scheme.

D. Completion phase

1. If not performing duplicate-removal then continue.
2. Intersection and difference will invoke constant-time but different processing as in Algorithm 3-1.

The theoretical lower bound of the time complexity of newSORT is $\Omega(log\ n)$. The marking phase requires only $O(1)$ time if linear interconnection is provided. The first and the last phases also requires only constant processing time. Therefore the POP-SORT using newSORT as its base also shares the same time complexity as newSORT. This even generalizes Theorem 3-2 -- Any sorting algorithm can be adapted to a four-phased POP-SORT without introducing any significant overhead.

Similar to a merge-oriented POP-SORT, an efficient implementation for a general POP-SORT needs two mark bits. One of the mark bits is used for distinguishing two multisets, and the other is for marking duplicates. In a general POP-SORT the quasi-stability, marking-minus, and marking-plus rules are still enforced using the two mark bits. The bit manipulation capability needed in a general POP-SORT is thus no less than that in a merge-oriented one.

3.4 Application to Join Operations

The number of result tuples after joining two relations A and B denotes the minimum totality of computing work needed for join. Assuming each relation of size n for simplicity, the figure may rarely become as large as $O(n^2)$. Using $O(n)$ processing elements, Kung's [Kung80] and Song's [Song81] linear time algorithms are optimal in the sense of handling the worst case. For most situations, the result relation has many fewer tuples. An A-tuple may have to join with only some B-tuples. By applying POP-SORT to precondition the relations, a join system shown in this section can perform the natural join and equi-join operations in sublinear time.

Any sorting algorithm can bring together all the elements of the same value. The groups of elements of the same values are called *aggregates*. We first sort the relations over the joining attributes using POP-SORT. The primitive operation is quasi-stable. It produces aggregates as well as insists that all the A-tuples precede B-tuples in each aggregate. We then can perform natural join and equi-join simply by shifting all the B-tuples in one direction to join with A-tuples. This process is called "easy-catch".

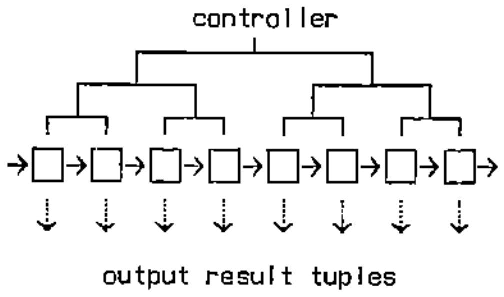


Figure 3-3. Logical structure of the easy-catch system for performing join operations.

Define d as the longest distance that a B-tuple needs to shift in order to catch all the joinable A-tuples. For easy-catch d is the largest size of the aggregates. To reach the goal of having sublinear time performance the catching process is better terminated after d shift steps. Unfortunately d is usually not known beforehand. In Figure 3-3 we show a solution to halting the catching process by superimposing a tree interconnection on top of the processing elements. A halting controller located at the root of the tree interconnection supervises all the processing elements. The tree interconnection provides the communication paths between the controller and the processing elements. Each processing element is responsible for reporting its activity by sending a "busy" or "idle" message up to the controller. The controller will broadcast the "halt" message when it decides all the processing elements are idle. If a halting message is received, the processing elements stop.

The programming of the join system is extremely simple. All the processing elements execute the same program and the program is nothing but a looping over after some initialization. Suppose there are two registers, a and b, capable of holding A and B tuples[†] in each processing element. The processing elements execute the looping program as follows:

```
for all $i$ do
(* initialization *)
    $a_i$, $b_i$ $\leftarrow$ nil;
    if $A$-tuple then load $a_i$ else load $b_i$;

(* easy-catch: shift and join *)
    repeat forever
        receive(msg);
        if msg = "halt" then stop;
        shift; (* $b_i \leftarrow b_{i+1}$ *)
        if $a_i$ match $b_i$ then {perform join; send("busy")}
            else send("idle");
```

The controller detects that all the processing elements are idle after a $\log n$ time delay. Another $\log n$ time delay is necessary for broadcasting the "halt" message to all the processing elements. The time for performing the natural join and equi-join is thus the total time for POP-SORT, easy-catch, and the halting delay.

$$T = T(POP-SORT) + O(d) + O(\log n) \text{ where } d \leq n.$$

† The tuple may only consist of tuple-id and the values for the joining attributes.

Since POP-SORT needs only sublinear time, the join operations can be done in sublinear time as long as d is less than $O(n)$. If $d = O(\sqrt{n})$ then the join operations can be done in $O(\sqrt{n})$ time using the bitonic POP-SORT.

The CHiP computers are good candidates for implementing the join system. Suppose that data items from both A and B are sorted by POP-SORT in a quasi-stable fashion into snake-like row-major order (see Chapter 5.) Two co-existing configurations shown in Figure 3-4 are feasible if there is a cross-over capability on switches. We assume that fan-in on switches behaves like a logic "AND", and switches also have fan-out capability to perform broadcasting. The linear and tree interconnections for the join system hence are provided by the two configurations.

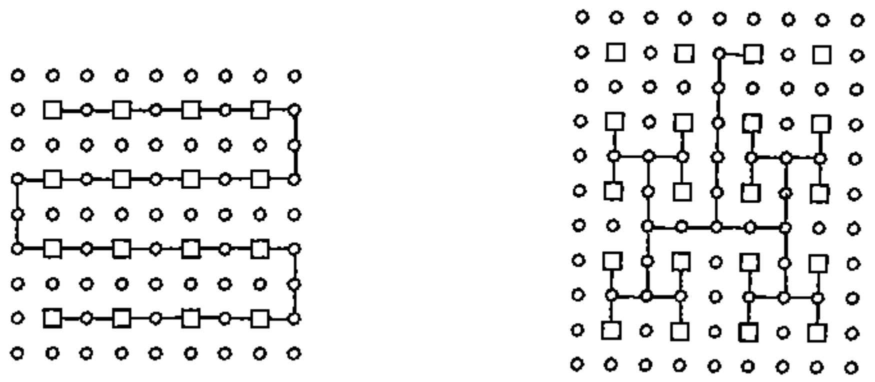


Figure 3-4. Two configurations on a CHiP computer for implementing the easy-catch system.

For the purpose of area-economy, the join system is implemented as above in a square CHiP region. Unfortunately, only perimeter processing elements have I/O ports to the peripheral storage devices. There would be a problem of non-uniform distribution of result tuples since they would accumulate at some PEs. We call this the *hot spots problem.*

If there is enough memory space in processing elements, the hot spots problem does not do any harm as long as the result relation is to be dumped out of the CHiP processor. For some cases, the result relation is to be processed further (see query embedding in Chapter 6.) Then the hot spots problem can be solved by the Sprinkle Algorithm as shown in Appendix B. The Sprinkle Algorithm employs the same communication scheme as a single stage of the bitonic merge. Let k be the maximum number of result tuples at hot spots. The Sprinkle Algorithm requires $O(\frac{k}{2} * \sqrt{n})$ time using mesh interconnection. The algorithm works especially well when k has small values.

This join system can perform other join operations too. The le-join and ge-join can be implemented exactly in the same way as natural join and equi-join, except that d is no longer the largest size of the aggregates. To perform ne-join, we need "two-way-catch", shifting B tuples in both directions to join with A tuples. The join system is especially suitable for natural join and equi-join because the value of d is more likely small for the two types of join operations.

In summary, the join system in Figure 3-4 provides adaptive performance for join operations. "Easy" joins that requires B-tuples join with only limited numbers of A-tuples are suitable for easy-catch implementation. They can be done with much better performance by avoiding executing them as "difficult" joins.

CHAPTER 4

OPTIMALITY OF THE PRIMITIVE OPERATION

The order of data items often has a profound influence on the speed and simplicity of algorithms which manipulate them [Knut73]. As a consequence, sorting has been found to be very useful as a pre-processing step for a wide variety of applications. It is well known that a considerable portion of the computer running time was and still is spent on sorting.

Although sorting is useful, in some cases it is overused. For example, selection of the median of n data items requires only $\Theta(n)$ comparisons, although the more expensive sorting is a common way to solve it. Moreover, sorting is completely useless in some other cases. Researchers found that the benefit of data ordering yields its ground to the computing power of parallel hardware on the searching problems (insertion, deletion, and update) [Bent79]. Despite these observations, the usefulness of sorting might be underestimated in the context of parallel computation.

While the usefulness of sorting might be over-emphasized in the sequential case, the feasibility of applying sorting in the parallel case needs more careful exploration. POP-SORT presents a mechanism to extend sorting to performing many other database operations. A methodology for applying parallel sorting to the solution of other problems is thus demonstrated. In order that POP-SORT be an optimal primitive, parallel sorting must be an optimal way to implement those database operations. However, is parallel sorting an optimal way of performing those database operations?

In this chapter we shall investigate the optimality of the primitive operation POP-SORT. We show how the reducibility of sorting to duplicate-removal plays a crucial role in determining the optimality. We then concentrate on studying the reducibility of sorting to duplicate-removal. Two comparison functions are considered: the strong comparison $(<, =, >)$ and the weak comparison $(=, \neq)$. We prove the reducibility for all the computations based on the weak comparison function. We also prove the reducibility for a subclass of computations based on the strong comparison function.

Section 4.1 establishes a time-complexity hierarchy representing the reducibility relationships among POP-SORT and the other five database operations. These relationships show that the hierarchy would collapse if sorting is reducible to duplicate-removal. A collapsed hierarchy implies the optimality of POP-SORT. The important relationship between sorting and duplicate-removal is then studied. A special model of parallel computation suitable for our study and two types of comparison functions are discussed in Section 4.2. In Section 4.3 and 4.4, we investigate the reducibility of sorting to duplicate-removal on the computation model with the two comparison functions respectively.

4.1 Collapsing the Complexity Hierarchy

By enforcing some extra ordering rules, any sorting algorithm can be extended to become POP-SORT without any significant overhead. POP-SORT serves as a primitive operation for sorting, duplicate-removal, union, intersection, and difference. The bitonic POP-SORT, an instance of the primitive operation, improves the upper bound for duplicate-removal over the algorithms in [Kung80] and [Song81]. Also, the fastest algorithms known for union, intersection, and difference apply sorting as a pre-processing step [Schw80]. Therefore POP-SORT does not sacrifice any efficiency for unifying these operations.

However, is POP-SORT an optimal primitive for performing these five database operations? To evaluate the optimality of the primitive operation, we investigate the complexity relationships between it and the five operations. The relationships are measured in terms of reducibility. Let P_1 and P_2 be two problems, and ψ_1 be any algorithm for solving the problem P_1. The problem P_2 is said to be reducible to P_1 iff there is an algorithm ψ_2 which applies ψ_1 to solve P_2. We are most interested in the case when both algorithms have time complexities of the same order, i.e. $O(T(\psi_1)) = O(T(\psi_2))$.

Some important reducibility relationships are summarized in the following:

- *All the five operations are reducible to POP-SORT.* Chapter 3 presents POP-SORT as a primitive operation which can perform sorting, duplicate-removal, union intersection, and difference.
- *POP-SORT is reducible to sort.* A "shift-copy and compare" scheme is shown in Chapter 3 to perform the marking-minus and marking-plus functions. A general mechanism based on the scheme is presented to

adapt any sorting algorithm to POP-SORT. The "shift-copy and compare" scheme takes only constant time. The adaptation overhead is thus negligible.

- *Duplicate-removal is reducible to union, intersection, and difference.* The operations union, intersection, and difference are allowed to take multisets as operands. Duplicate-removal thus can be implemented as: $rmdup(A) = union(A,\varphi) = inter(A,A) = differ(A,\varphi)$, where φ is the empty set.

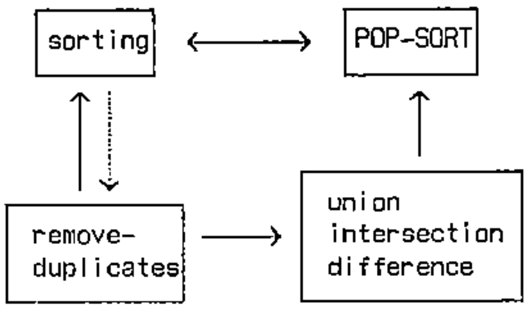


Figure 4-1. Collapsing the time complexity hierarchy implying the optimality of POP-SORT.

The above reducibility relationships are also depicted as a time complexity hierarchy in Figure 4-1. The arrow "—>" in the figure denotes the relationship "is reducible to". To collapse the complexity hierarchy would imply the optimality of POP-SORT. The relationship represented by the dotted arrow "·····>" therefore plays an important role in collapsing the complexity hierarchy. For POP-SOPT to be an optimal primitive, sorting must be an optimal way to perform duplicate-removal. The key to unifying the five operations by POP-SORT is the extension of sorting to mark off duplicate items. Hence there is no surprise that the optimality of POP-SORT relies on the optimality of sorting to perform duplicate-removal.

Muller and Preparata [Mull75] showed a constructive switching network of $O(\log n)$ depth which performs sorting. The switching network is an implementation of the enumeration comparison method in which each data item is compared with any other one. This is an evidence that the the benefit of parallel hardware supercedes that of data ordering. The switching network can be used to implement POP-SORT achieving the theoretical lower time bound $\Omega(\log n)$. This is actually an immediate proof that POP-SORT based on Muller and Preparata's network is optimal. It is also a proof that sorting is reducible to duplicate-removal. However the switching network requires $O(n^2)$ comparators and switches. In the following sections we investigate further the reducibility of sorting to duplicate-removal in the context of fewer processing components.

4.2 Comparison Functions and Computation Models

This section discusses comparison-based computation on parallel machines. We point out that there are two types of comparison functions that must be considered. We also present a universal model of parallel machines to facilitate our study on the reducibility of sorting to duplicate-removal.

Comparison between two elements is a primitive instruction for both sorting and duplicate-removal. According to the law of trichotomy, exactly one of the possibilities $x < y$, $x = y$, $x > y$ is true. However circuit level implementations of the pairwise comparison can provide this information in one of the following four ways: (1) $<, =, >$; (2) $\leq, >$; (3) $<, \geq$; and (4) $=, \neq$. They all involve different switching logic functions. The first three are the strong comparison functions which can be shown equivalently powerful.[†] The last one, called the weak comparison function, is not adequate for sorting though it is for duplicate-removal.

A sorting algorithm should use one of the strong comparison functions in order to come out with a total ordering. For a duplicate-removal algorithm, it is not necessary to assess any ordering information. It may use the data ordering to some extent, or it may completely ignore the data ordering. That is, duplicate-removal algorithms may use the weak comparison alone, the strong comparison alone, or the mixture of both comparison functions.

A variety of models of parallel computation have been proposed. They may be grouped into two classes: shared memory machines and fixed connection networks [Prep81, Boro82]. The former class assumes a large random access memory shared by all the processing elements or an equivalent system (see examples in [Fort78, Gold78, Lev81].) The latter assumes a fixed interconnection among processing elements, or between processing elements and memory modules (see examples in [Brow80, Schw80, Prep81].)

In terms of the restrictions on accessing memory modules, shared memory machines may be classified into three categories: concurrent read or write, concurrent read but exclusive write, exclusive read or write. Execution time on shared memory machines is usually measured as the number of operation steps performed, assuming that the memory access time is free. This type of computation model overlooks technological feasibility.

† Two $(\leq, >)$ or $(<, \geq)$ comparisons are equivalent to one $(<, =, >)$ comparison.

While shared memory machines are suitable for deriving lower time bounds, they are not appropriate for studying data movement realistically.

For current hardware technologies, fixed connection networks are more reasonable. However a single interconnection cannot provide optimal hosts for all the important algorithms. Furthermore, many problems require only infrequent and irregular processor communication. Fixed connection networks are too restricted to study the reducibility relationships between sorting and duplicate-removal.

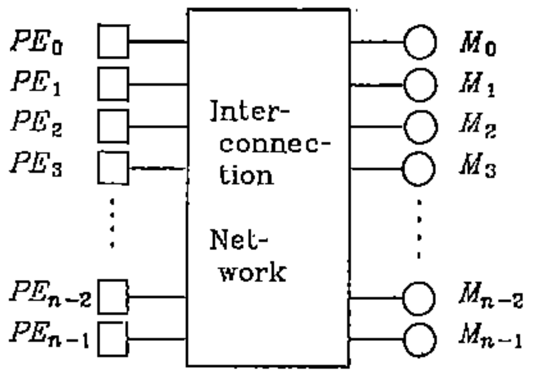


Figure 4-2. PIM machine as a model of parallel computation.

In order to study sorting and duplicate-removal on a general base, we need a universal model of parallel machines. The universal model must be able to represent each specific machine model and is suitable for studying data ordering and data movement. For these purposes, we present a computation model called the PIM machine shown in Figure 4-2.

The PIM machine has three components: a group of processing elements, an interconnection network, and a collection of memory modules (which may be as small as single memory words.) Separate memory modules enables us to "observe" data items being processed. We assume that the interconnection network has all the flexibility and power which enables the PIM machine to emulate any parallel machine.

The interconnection network provides communication paths between the processing elements and the memory modules. At one extreme, we may assume that the interconnection network is so powerful that the PIM machine behaves like a shared memory machine. At or near the other extreme, we may assume that the interconnection network provides fixed communication paths as simple as those for the linear array connection. For emulating reconfigurable computers, the interconnection network has the reconfigurability to provide different interconnection patterns.

Communication overhead is important on parallel machines, especially when the interconnection network becomes less powerful. On the PIM machine, the time complexity is measured by taking both comparison count and data movement steps into account. Data communication time may be absorbed by providing feasible interconnections between processing elements and memory modules. Transmission time is assumed independent of the lengths of communication paths; the propagation delay problem is not an issue here. For example, sorting needs $O(\sqrt{n})$ data routing steps and $O(\log^2 n)$ comparison steps using the mesh interconnection [Thom77, Nass79], or $O(\log^2 n)$ routing and comparison steps using the shuffle-exchange interconnection [Ston71].

4.3 On Enumeration Comparison

Based on the weak comparison function, duplicate-removal requires $\frac{1}{2}n(n-1)$ comparisons since every pair of data items must be compared directly. Taking advantage of data ordering, or using the strong comparison functions, the total comparison count may be reduced. However, the total processing time is not necessarily decreased because the time complexity is measured as the sum of parallel comparison steps and parallel data movement steps. The absolute requirement of the $\frac{1}{2}n(n-1)$ weak comparisons therefore does not exclude the possibility of a fast parallel algorithm for duplicate-removal.

In this section, we shall prove that sorting is reducible to any duplicate-removal algorithm that is based on the weak comparison function. This is not unreasonable because enumeration comparison methods have been proposed for sorting [Knu73, Mull75, Prep78] in which each data item is compared with every one of the others. Naturally, sorting requires the application of one of the strong comparison functions.

Let ψ_1 be a duplicate-removal algorithm using the weak comparison function. The execution of the algorithm may be functionally partitioned into two stages: (1) performing enumeration comparisons, and (2) determining mark bits (assuming there is a mark bit corresponding to each data item.) The algorithm thus may be visualized as making the weak comparisons to fill up a triangular table (upper triangular bit matrix) and figure out the mark bits as shown in Figure 4-3.

Notice that the mark bits are obtained by "ORing" all the bits on each row. The following program segment describes the abstract function of the algorithm ψ_1. It is not required that ψ_1 be actually executed this way.

```
(* $i$, $j$ : indices;  M : matrix *)

(* perform enumeration comparisons *)
for all $i < j$ do
   if $x_i = x_j$ then M[$i$,$j$] := 1 else M[$i$,$j$] := 0;

(* determine mark bits *)
for all $i$
   $m_i$ := $OR_{all\ j>i}$(M[$i$,$j$]);
```

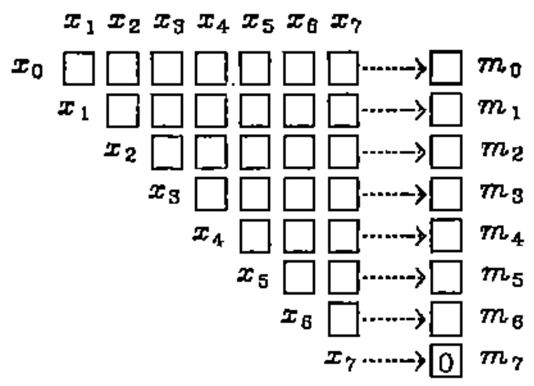


Figure 4-3. The function of enumeration comparison methods: table filling and row computation.

Now, perform the following procedure to modify the algorithm ψ_1:

1. Substitute the weak comparison function with the strong comparison function ($\leq$, $>$).
2. Fill up the whole matrix rather than just the upper triangular half by entering two entries to the matrix for each comparison performed.
3. Substitute the "OR" operation with a "SUM" operation.

The abstract function of the new algorithm, say ψ_2, may be described as the following program segment:

```
(* perform enumeration comparisons *)
for all $i < j$ do
    if $x_i > x_j$ then {M[$i,j$] := 1; M[$j,i$] := 0}
        else {M[$i,j$] := 0; M[$j,i$] := 1};

(* determine unique ranks *)
for all $i$
    $r_i := \sum_{j=1}^{n} M[i,j]$;
```

The two algorithms, ψ_1 and ψ_2, are not necessarily implemented in two clearly separated stages as described in the program segments. The program segments, however, manifest the required computation that must be done by the algorithms. No matter how the two functional stages of ψ_1 are actually executed on PIM machines, ψ_2 is executed in the same way. The algorithm ψ_2 would compute unique ranks for all the data items in spite of duplicates. Both algorithms share exactly the same time complexity, assuming all the operations take unit time. We have thus proved the following Lemma.

Lemma 4-1. From any duplicate-removal algorithm based on the weak comparison function, we can find an algorithm to compute unique ranks for all the data items in the same time.

Let $x_0, x_1, \ldots, x_{n-1}$ be a sequence of data items, $x_i \varepsilon$ [0, m-1] and $m \gg n$. The sequence can be transformed into a sequence of unique ranks $\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{n-1}$ (where $\hat{x}_i$ is the unique rank of x_i) by the algorithm ψ_2. Sorting the sequence of unique ranks is much easier than sorting the sequence of original data items. Thus, a two-phased sorting scheme is indicated. It first determines unique ranks and then redistributes data items according to their unique ranks.

Data redistribution given the unique ranks can be done in $O(\log n)$ time with the switching network in [Mull75]. With the assumption of a shared memory, it takes only constant time [Prep78]. For a problem that has one of its outputs determined by all the n inputs, the theoretical lower time bound is $O(\log n)$. Remove-duplicates or determining unique ranks therefore requires no less time than redistributing data items. Hence we have proved the following theorem.

Theorem 4-1. Sorting is reducible to any duplicate-removal algorithm that is based on the weak comparison function.

4.4 On Establishing Total Orderings

In this section we investigate if sorting is reducible to duplicate-removal based on the strong comparison function $(<, =, >)$. Although the weak comparison function is adequate, duplicate-removal algorithms using the strong comparison function take advantage of data ordering. To show the reducibility, we need to prove that the information of data ordering collected by duplicate-removal algorithms can be easily transformed to an explicit total ordering as produced by sorting.

We first define *semi–digraph* to represent the minimum set of comparisons required for duplicate-removal. By showing that the semi-digraph must contain a total ordering, we prove that the comparisons required for duplicate-removal are also adequate for sorting. While this is enough to show the sequential reducibility of sorting to duplicate-removal, it is not sufficient for the parallel reducibility. We therefore pursue the matter further and show that the parallel reducibility is true at least for a useful type of *homogeneous* computation.

Let X = $\{ x_0, x_1, \ldots, x_{n-1}\}$ be a multiset consisting of n elements from a totally ordered set. Define C_m as the minimum set of comparisons required for the elimination of duplicates. A *semi–digraph* which contains both directed and undirected edges can represent the set C_m:

$$x_i \bullet\!\!\longrightarrow\!\!\bullet\, x_j \text{ if } x_i > x_j \text{, or}$$

$$x_i \bullet\!\!-\!\!\!-\!\!\bullet\; x_j \text{ if } x_i = x_j .$$

The semi-digraph is composed of n vertices and no more than $\frac{1}{2}n(n-1)$ edges. For a path between a pair of vertices x_i and x_j, the path is undirected if it contains only undirected edges, or the path is directed if it contains at least one directed edge. In the semi-digraph, directed paths denote the ordering relationship "is greater than" or "is less than", and undirected paths denote the relationship "is equal to".

To guarantee that all the duplicates are found, there must exist a path between any pair of vertices x_i and x_j, $i \neq j$. Otherwise the ordering relationship between them is not known. The path is either undirected or one-way directed. The graph is conflict free because of the uniqueness of the ordering relationship between any vertex pair. In other words, exactly one of the possibilities $x_i < x_j$, $x_i = x_j$, $x_i > x_j$ is represented in the graph for each pair of vertices. The semi-digraph should contain a subgraph equivalent to that shown in Figure 4-4. Lemma 4-2 is thus proved.

Lemma 4-2. The semi-digraph, representing the minimum set of the comparisons needed for duplicate-removal, contains a total ordering.

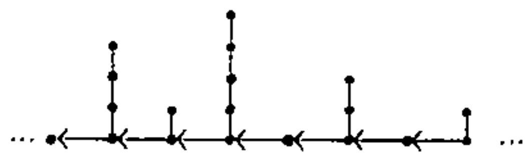


Figure 4-4. The total ordering contained in the semi-digraph.

By Lemma 4-2 elimination of duplicates always needs those comparisons which are sufficient to come out with a total ordering. Elimination of duplicates must then have done the comparisons required for sorting. This is enough to show the sequential reducibility of sorting to duplicate-removal, since the sequential time complexity can be reflected by the comparison count alone.

On PIM machines, data communication time is important. Although duplicate-removal requires at least the same comparison work as that for sorting, it does not require that data items be arranged in any particular order. To arrange data items in order may entail more data movement time than the total processing time for duplicate-removal. Without further study, it is not possible to say that sorting is reducible to duplicate-removal. Nevertheless the existence of the total ordering is guaranteed after performing any duplicate-removal algorithm.

We shall prove the parallel reducibility for a special type of parallel computation that insists on a *"homogeneous sequence of execution"* [Knu73, p.220]. Whenever we compare x_i with x_j the subsequent execution for the case $x_i < x_j$ is exactly the same as for the case $x_i > x_j$, except with the data values interchanged. This type of computation is widely applied in practical parallel computation since the complexity of the decision structure is extremely simple. In the following, we first define general comparison networks to simulate the execution of duplicate-removal algorithms on PIM machines. We then derive versatile comparison networks with fixed connections which are able to sort and identify duplicates.

Comparison Network Model

Execution of algorithms on PIM machines can be traced by recording activities at processing elements and value changes at memory locations. For comparison-based computation, processing elements primarily perform comparison and data movement. To study data ordering, we focus on observing the memory part and further impose time-variant ordering relationships ($<, =, >$) among different memory locations. Concurrent writes to a memory location are prohibited lest the ordering information should be disrupted. A comparison network is thus presented to simulate one execution of a comparison-based algorithm on PIM machines.

The execution of a comparison-based algorithm on an input permutation can be recorded as a sequence of comparison steps and data movement steps. One can visualize the execution as applying processing elements to memory locations as many times as the number of operation steps. A comparison network has four important parameters:

n - problem size or the number of data items,

m - storage capacity or the maximum number of data copies at any instant,

t - depth of network or the number of parallel comparison/routing steps,

k - degree of parallelism or the largest number of comparisons that can be performed at a parallel comparison step.

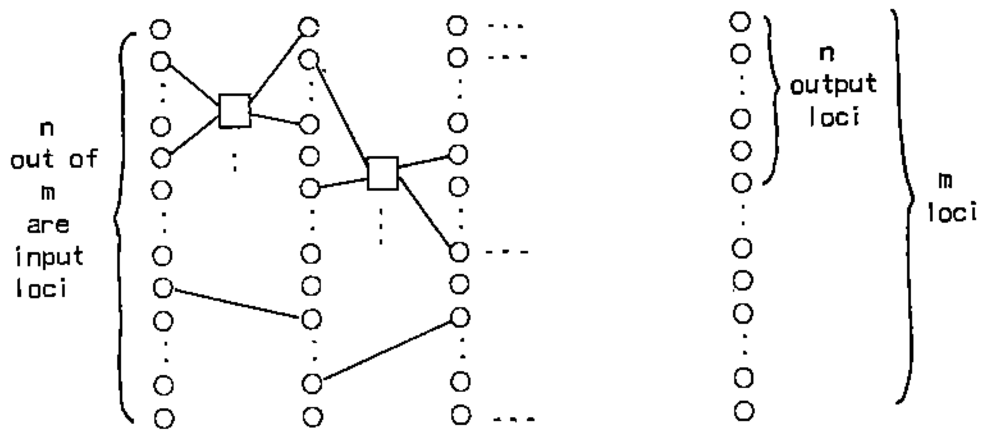


Figure 4-5. A general comparison network.

As shown in Figure 4-5, a comparison network consists of two types of components: loci (in circles) and comparators (in squares). A "locus" is a memory location capable of holding one data copy. There are totally t instances of the m loci in the network. The data value at a locus may (1) retain its previous value, (2) copy from another locus, or (3) receive a value from a comparator. Thus, a data value may fan out to have multiple copies (concurrent reads), but fan-in of many data values is undefined (exclusive write). A comparator reads two data values from its source loci, compares them, and writes them out to its object loci. We assume, for generality, the order of the two outputs (inputs) of a comparator is not important. A comparator may route the two data in arbitrary order to its object loci.

The I/O of the comparison network is where-oblivious [Lipt81]. The first n loci are arbitrarily defined as output loci. In the very beginning, n (input loci) out of the m loci have the n data items. After t comparison and data routing steps, the n data items are in the output loci with all the duplicates marked off.

General comparison networks allow broadcasting and multiple copies of data items. A special example of the comparison networks is called *rw-conflict-free network*. Rw-conflict-free networks do not allow fan out of data values, therefore do not have any memory conflicts, neither read conflicts nor write conflicts. The *sorting network* in [Knu73, pp. 220], or *network of comparator modules*, is a restricted form of the rw-conflict-free network. Sorting networks have exactly n data copies (i.e. $n = m$) and strictly route data items in a pre-defined way. To each comparator the source loci and object loci are the same in sorting networks.

Versatile Network N_s

A duplicate-removal algorithm does not have to arrange data items in any particular order. However, our approach is, for any duplicate-removal algorithm ψ, to derive a "compatible" algorithm ψ_s that is able to sort as well as to remove duplicates. The derived algorithm does not increase the time complexity, whereas it would move data items in such a manner as to come out with an explicit total ordering. This approach originates from the the potential existence of a total ordering described in Lemma 4-2.

Suppose that the execution of ψ of a given input permutation is recorded as a comparison network N. A restricted form of the comparison network N_r can be derived from N such that N_r emulates N in the same comparison and data routing steps. In the network N_r the comparators are restricted in the sense that the larger inputs are always routed to the upper object loci. (Notice that N_r shares the four parameters with N, but enforces data routing differently.)

Define O_i as the ordering relationship among the m loci at the i-th stage in the network N_r. The initial relationship O_0 contains no information. The restricted network N_r has rigid interconnection and rigid data routing. By induction, the ordering relationships $O_0, O_1, \ldots, O_t$ are all known. The ordering relationship O_t must contain a total ordering according to Lemma 4-2. We can then rearrange the first n loci in N_r according to the ranks determined by the total ordering O_t and come out with a new network N_s. Hence data items are sorted in descending order in N_s. Based on this observation, we prove the following theorem.

Theorem 4-2. Given any algorithm ψ for duplicate-removal which performs a homogeneous sequence of execution on exclusive-write PIM machines, there exists a compatible algorithm ψ_s which runs as fast and is able to sort.

[Proof] Input permutations are not relevant to the execution sequence because of the homogeneity of execution. Therefore there is a single comparison network N which simulates the execution of ψ on any input permutation. Following the procedures mentioned before, the network N can be transformed into N_r then into N_s

without changing the network depths. The algorithm ψ_s corresponding to the network N_s can then perform duplicate-removal and sorting in the same time as ψ can perform duplicate-removal. ■

Without assuming the homogeneous sequence of computation we may have different comparison networks with different depths for the input permutations. Although they all complete the comparison work represented in the semi-digraph, the final relationship O_t may not be fixed. Whether Theorem 4-2 is true for non-homogeneous execution needs further investigation. As a conclusion, if there exists a single comparison network to model the execution of a duplicate-removal algorithm for all the input permutations then we can derive a compatible network to perform sorting. However, the reducibility of sorting to duplicate-removal does not imply the existence of the single comparison network.

Applying the proof technique for Theorem 4-2 to an exclusive-write, exclusive-read network, we have Corollary 4-1.

Corollary 4-1. Given any rw-conflict-free network for duplicate-removal, there exists a rw-conflict-free network which has the same depth and is able to sort.

Some well known parallel machines can be modeled by PIM machines with fixed interconnection patterns between n processing elements and n memory modules. Examples are the ILLIAC IV computer [Barn68], tree machines, and the ultracomputer. The execution of duplicate-removal algorithms on a particular machine can then be modeled by a especially regular sorting network. If we rearrange the horizontal data lines then the new sorting network may not preserve the original pattern of connections. That is, the interconnection pattern is changed.

Corollary 4-2. Given any algorithm for duplicate-removal on a exclusive-write PIM machine with interconnection pattern I, there exists an interconnection pattern I_s which enables the algorithm to perform sorting.

It is not necessary that the interconnection patterns I and I_s in Corollary 4-2 be different. On the machine with a tree interconnection or a d-dimensional mesh interconnection, we can prove that both sorting and duplicate-removal can be solved using basically the same algorithms. On the tree machine, implementation of the heap sort requires $O(n)$ time [Mea81]. When the ordered sequence is removed from the root node, all the duplicates can be found. On the mesh-connected machine, Batcher's sorting scheme can be implemented in $O(n^{1/d})$ time, where d is the degree of the dimension [Thom77]. The bitonic POP-SORT which is based on Batcher's sorting scheme and a new comparison function performs duplicate-removal in the same time.

Due to the I/O bottleneck at the root node, linear time is the best performance obtainable from the tree machine. Based on the argument on the longest distance that data may need to move on the mesh-connected computer, $O(n^{1/d})$ is the optimal time [Thom77]. These two examples show that duplicate-removal can be best solved by optimal sorting algorithms on the particular parallel machines.

CHAPTER 5

BITONIC SORT ON THE CHiP COMPUTERS

Batcher's bitonic sort has been conjectured to be the best network sorting method † [Prep78]. It has been intensively studied and several adapted algorithms for machines of different processor interconnections are available [Ston71, Thom77, Nass79, Prep81]. The bitonic sort can be done in $O(\sqrt{n})$ time on mesh-connected computers [Thom77, Nass79]. It requires only $O(\log^2 n)$ time with the shuffle-exchange interconnection [Ston71] or the cube-connected-cycles (CCC) [Prep81].

The bitonic POP-SORT, based on Batcher's bitonic merge sort, is an important example of POP-SORT. It can simplify the processing of whole queries on the CHiP processors significantly (see Chapter 6). Implementation of the bitonic POP-SORT directly refers to the implementation of the bitonic sort. On a CHiP computer, it is feasible to embed all those interconnections on the switch lattice and perform the adapted algorithms.

C. D. Thompson proved that any layout of the shuffle-exchange graph requires at least $O(n^2/\log^2 n)$ area [Thom80]. There are layout algorithms which achieve $O(n^2/\log^{\alpha} n)$ area, α = 1/2, 1, 3/2, or 2 [Thom80, Hoey80, Klei81]. However, the layouts are complicated and unavoidably require

† $O(\log^2 n)$ depth and $O(n \log^2 n)$ processing components.

large areas. The CCC improves the shuffle-exchange layouts on regularity, but still requires large embedding areas [Prep81]. On the CHiP computers, embedding the shuffle-exchange layouts tends to require even larger areas, because the CHiP processors are not simple grids.

A lacing technique can be used to exploit the cross-over capability of switches for embedding layouts on the CHiP processors (see Appendix B). The lacing technique is very useful in embedding complicated interconnections. Nevertheless, embedding powerful interconnections like shuffle-exchange and CCC would leave a large portion of processing elements unused. Furthermore, there are long connection paths in any layout of the shuffle-exchange graph or the CCC. Long data paths are vulnerable to the propagation delay problem [Bila81]. In this chapter we therefore emphasize the bitonic sort with the mesh and mesh-like interconnections.

We shall report some interesting aspects about performing the bitonic sort on the CHiP computers with the mesh interconnection or its variations. Embedding the mesh interconnection is straightforward. The performance of $O(\sqrt{n})$ matches the I/O time required for a square CHiP region anyway. Taking advantage of the switch corridors and the cross-over capability of switches, one can further improve the communication power over the simple mesh interconnection.

In Section 5.1, an efficient Rearrangement Algorithm is presented to reorder sorted data items from one indexing scheme to another. A technique, called sorting with shadow regions, is shown in Section 5.2. With this technique, allocation of exactly n processing elements is sufficient for sorting n data items with the bitonic sort (n is any integer, not necessarily a power of 2). In Section 5.3, we discuss methods to improve the data routing over the mesh interconnection with the switch lattices. We then address the problem of sorting more data items than the number of processing elements used in Section 5.4.

5.1 Reordering Between Indexing Schemes

For sorting with the mesh interconnection, there are three important schemes of indexing the processing elements:

(1) Shuffled row-major indexing, shown in Figure 5-1(a).

(2) Row-major indexing, shown in Figure 5-1(b).

(3) Snake-like row-major indexing, shown in Figure 5-1(c).

Data items are sorted into particular orders defined by the indexing schemes. The choice of a particular indexing scheme depends on how the sorted items are to be used.

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

(a)

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

(b)

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

(c)

Figure 5-1. (a) Shuffled row-major indexing. (b) Row-major indexing. (c) Snake-like row-major indexing.

The shuffled row-major indexing comes from an optimal adaptation of the bitonic sort to the mesh-connected computers [Thom77]. With this indexing scheme, the more often the processing elements are required to communicate with each other, the closer they are physically located. If the sorted sequence is the final result, or when the sorted items are to be stored in secondary memories, the row-major indexing is perhaps preferred. For the snake-like row-major indexing, it would simplify the embedding of the linear array connection for any after-sorting processing.

Thompson and Kung [Thom77] designed the optimal adaptation of the bitonic sort with the shuffled row-major indexing scheme. Their algorithm takes $(14\sqrt{n})\, t_R + (2\log^2 n)\, t_C$ time.† They also proved that data items can be rearranged to obey other indexing schemes with a relatively insignificant extra cost of $(4\sqrt{n})\, t_R$ time, provided that each processing element can store $\sqrt{n}$ data items. On the other hand, Nassmi and Sahni [Nass79] proposed different adapted algorithms of the bitonic sort to sort data items into the row-major order and the snake-like row-major order. Their algorithms require $(14\sqrt{n})\, t_R + (2\log^2 n)(t_C + t_I)$ time, where t_I is the time to interchange the contents of two registers.

The three indexing schemes all have their own advantages. It is not unusual for more than one indexing scheme to be needed. One may then employ different algorithms for different indexing schemes. For query embedding on the CHiP computers, the shuffled row-major indexing is chosen for the bitonic POP-SORT for a simple and efficient realization (see Chapter 6). To perform join operations using the bitonic POP-SORT, we proposed a join system using a linear array connection (see Chapter 3). Thus, rearranging data items into the snake-like row-major order is needed.

We shall present an "easier" Rearrangement Algorithm which transforms the shuffled row-major order into the row-major order in less than $(2\sqrt{n})\, t_R$ time. The algorithm requires only two registers at each

† The lower order terms are truncated.

processing element. To translate between the row-major order and snake-like row-major order, $(\sqrt{n} + 1)\, t_R$ time is sufficient. Since the rearrangement algorithm is reversible, transformation between any two indexing schemes can be done in $(3\sqrt{n})\, t_R$ time without the requirement of extra memory space.

Suppose there are two square regions, left and right regions, each of i^2 data items. Data items are already sorted in row-major order in both regions. All the data items in the right region are larger than those in the left one. Let $r_{1,0}, r_{1,1}, \ldots, r_{1,i-1}$ denote the rows in the left region and $r_{2,0}, r_{2,1}, \ldots, r_{2,i-1}$ in the right region. Algorithm 5-1 describes a rearrangement procedure which merges the two regions into a 1:2 rectangular region with the row-major indexing again. The rearrangement procedure, composed of simply swapping rows and unshuffling columns, constitutes a basic step for the Rearrangement Algorithm. In Figure 5-2, an example of merging two 4×4 regions is shown. The triangular interchange scheme shown in Figure 5-3 can be used to unshuffle columns concurrently.

Algorithm 5-1: A basic rearrangement step.

1. Swap odd rows in the left region with even rows in the right region; $r_{1,2j+1} \leftrightarrow r_{2,2j}$, for $j = 0,1,\ldots,(\frac{i}{2}-1)$. Time: $(i+1)\, t_R$.
2. Unshuffle each column. Time: $2(\frac{i}{2}-1)\, t_R$.

   Total time: $(2i-1)\, t_R$.

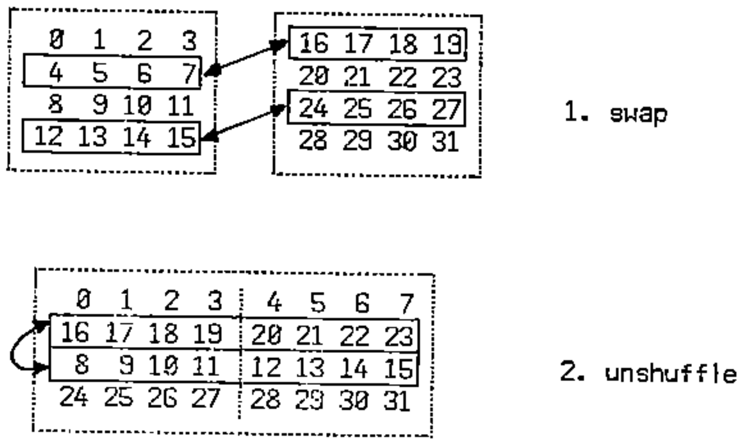


Figure 5-2. Rearrangement merge of two 4×4 regions.

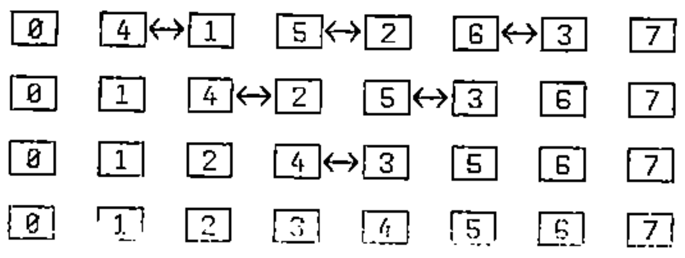


Figure 5-3. A triangular interchange scheme to perform unshuffle.

For a square region of n data items with the shuffled row-major indexing, the Rearrangement Algorithm simply applies the rearrangement merge step for $\log\sqrt{n} - 2$ times. The Rearrangement Algorithm starts by merging 2×2 regions, 4×4 regions, ..., and at last $\frac{\sqrt{n}}{2} \times \frac{\sqrt{n}}{2}$ regions. The total rearrangement time of $(2\sqrt{n})\, t_R$ is calculated as follows:

$$
\begin{aligned}
&(2 * \frac{\sqrt{n}}{2} - 1) + (2 * \frac{\sqrt{n}}{4} - 1) + \cdots + (2 * 2 - 1) \\
&= (\sqrt{n} + \frac{\sqrt{n}}{2} + \cdots + 4) - (\log\sqrt{n} - 2) \\
&< 2\sqrt{n} - \log\sqrt{n}.
\end{aligned}
$$

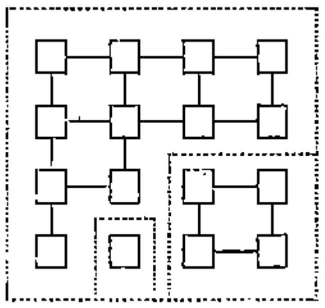
In the case of query embedding, rearranging data items is a requirement. Sorting of a large number of data items can also have benefit from the efficiency of the Rearrangement Algorithm. The s^2-way merge algorithm of [Thom77] is faster than the bitonic sort algorithms of [Thom77] and [Nass79] when $n > 512$. The s^2-way merge algorithm sorts data items into snake-like row-major order. It requires $(6n + O(n^{2/3}))\, t_R$ + $(n + O(n_{2/3}))\, t_C$ time. If $t_C \le 2\, t_R$ then $(8n)\, t_R$ is sufficient time for sorting. Sorting with the s^2-way merge algorithm and then rearranging data items into shuffled row-major order takes less than $(11n)\, t_R$ total time. For large sorting problems, it is therefore faster to perform the s^2-way merge sort first and then some rearrangement algorithm to translate to the right indexing scheme.

5.2 Sorting with Shadow Regions

The bitonic sorting algorithms of [Thom77] and [Nass79] assume a square array of mesh-connected processors. Performing the sorting algorithms on the CHiP computers thus needs a square region of $2^{2\lceil \log\sqrt{n} \rceil}$ processing elements for sorting n data items. Without any extra effort, the sorting algorithms can also be executed on a 1:2 rectangular region. The required CHiP region is thus reduced to have $2^{\lceil \log n \rceil}$ area. However, there are still $2^{\lceil \log n \rceil} - n$ more processing elements used than the number of data items to be sorted, where $0 \le 2^{\lceil \log n \rceil} - n \le n - 1$.

On a rigidly mesh-connected computer, allocation of a larger processor array to sorting a smaller number of data items is inevitable. The sorting algorithms must also assume a dummy data item ($-\infty$ or $+\infty$) at each additional processing element. On the CHiP computers, the processor interconnection is flexible and configurable. We therefore propose to take advantage of the programmable switch lattice to resolve the superfluous allocation problem and handle the dummy value requirement.

Suppose the bitonic sorting algorithm of [Thom77] is chosen to sort data items into the shuffled row-major order. We present a technique, called sorting with shadow regions, that requires the allocation of exactly n processing elements. In Figure 5-4, an example of sorting 176 data items with two shadow regions is shown. The shadow regions can be allocated to smaller sorting jobs or solving other smaller problems. Hence the benefit of sorting with shadow regions is to improve the utilization of the processing elements.



1. Each square repesents a 4×4 mesh-connected region.
2. 176 + 4×4 + 8×8 = 256.

Figure 5-4. Sorting 176 data items with 4×4 and 8×8 shadow regions.

Given n data items, a sorting region of n processing elements is allocated according to the shuffled row-major indices from 0 to $n-1$. The region consisting of the other $2^{\lceil \log n \rceil} - n$ processing elements is the shadow region. If the data items are to be sorted in ascending order, the dummy value of $+\infty$ is chosen. The communication between the sorting region and the shadow region is therefore nothing but sending and receiving the dummy value. Using the shadow region merely for the trivial communication is totally wasteful.

The trivial communication can be simulated at those processing elements on the boundary with the shadow region. When those processing elements are requested to read a value from the shadow region, they are given the dummy value; when they are requested to write to the shadow region, they just ignore the request. An incomplete mesh interconnection that connects only the processing elements of indices from 0 to $n-1$ is thus sufficient. There are no connections between the sorting region and the shadow region. The shadow region is free for other use.

The technique of sorting with shadow regions can be applied to allocate CHiP regions for sorting jobs in a more compact fashion. Let n_1 and n_2 be the numbers of the data items of two sorting jobs. Together for the two sorting jobs, a CHiP region of $n = 2^{\lceil \log(n_1+n_2) \rceil}$ is allocated. The region of $(0 \sim n_1-1)$ is dedicated to the first job, and the region of $(n-n_2 \sim n-1)$ is dedicated to the second. They both assume the regions not allocated to themselves as shadow regions. They may both choose the dummy value of $+\infty$. Hence the first sequence is sorted in ascending order and the second is sorted in descending order. Interestingly the whole data sequence in the region of n area becomes a bitonic sequence. The benefit of applying the technique of sorting with shadow regions will be demonstrated further in Chapter 6.

5.3 Improvements on the Data Routing

The bitonic sort with the mesh interconnection requires $O(\log^2 n)$ comparison time and $O(\sqrt{n})$ data routing time. The comparison time is optimal with respect to the bitonic sorting method. The data routing time, however, is due to the restricted communication power of the mesh interconnection.

With the mesh interconnection, data communication between two distant processing elements is achieved by passing data over. To send a data item from a processing element to the other one i locations apart thus requires i routing steps. With the corridor width w and the cross-over capability c, the CHiP computer may provide up to $w*c$ data paths crossing the corridors. The availability of the $w*c$ data paths can be used by the processing elements to communicate with each other at a distance.

Consider a row of $2i$ processing elements and a horizontal corridor dedicated to the data communication among the processing elements. The bitonic sort requires that the i data items at the processing elements in the left half be sent to those in the right half. This needs at least $i/w*c$ time units since the communication bandwidth through the corridor is $w*c$. Hence any improvement in the data routing on the CHiP computers over the mesh interconnection is bounded by the speed-up factor $w*c$.

In additional to performing the passing-over type of communication, the CHiP computers can improve the communication power over the mesh interconnection in the following ways:

1. *Communication with direct connections.* It is feasible to provide direct connections for all the communication requirements of the bitonic sort. A possible cost is $O(\log^2 n)$ reconfiguration steps and $O(\log n)$ switch settings. Moreover, data transmission through paths of significantly different lengths needs careful synchronization. However, cautious application of direct connections, e.g. for short distance communication, can avoid the complication of reconfiguration and synchronization.

2. *Communication with z-location-jumps.* Direct connections for short distance communication also provide short cuts for long distance communication. Direct connections between processing elements z locations apart can be used to communicate processing elements $i*z$ locations apart in i steps of z-location-jump.

On different switch lattices, or different values of w and c, we expect some variations in reaching the optimal improvement on the data routing. We are most interested in the practical values of w and c which are $1 \leq w \leq 8$ and $1 \leq c \leq 4$ (the degree of incident data paths to switches $d \leq 8$). An example of $w = c = 2$ and $d = 8$ which achieves the speed-up factor $w*c$ shall be demonstrated. For other values of w and c in which we are interested, the improvement on the data routing can be done in a similar way.

With the switch lattice of $w = c = 2$ and $d = 8$, we design three interconnection patterns: $I_{1,2}$, I_{4h}, and I_{4v}. Figure 5-5 shows three sub-patterns which are superimposed to form the pattern $I_{1,2}$. The interconnection $I_{1,2}$ provides direct connections required between PEs one or two locations apart, both horizontally and vertically. In other words, $I_{1,2}$ maps the necessary connections for the bitonic stages 1~4 onto the CHiP switch lattice. The interconnection I_{4h} provides direct connections for PEs four horizontal locations apart, and I_{4v} for PEs four vertical locations apart. They can be layout using the lacing technique as in Appendix B. The three interconnection patterns together map the necessary connections for the bitonic stages from 1 to 6. For bitonic merge stages 7, 8, and so on, the interconnection patterns I_{4h} and I_{4v} can be used to speed up the data routing with the 4-location-jumps.

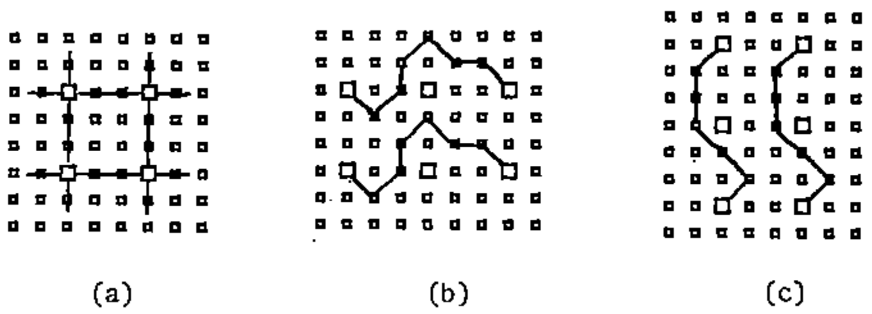


Figure 5-5. The interconnection pattern $I_{1,2}$ composed of three sub-patterns: (a) I_1, (b) I_{2h}, and (c) I_{2v}.

To perform the bitonic sort with the three interconnection patterns, the comparison steps remain the same. The routing steps and the reconfiguration steps are analyzed as follows ($w = c = 2$).

$$\text{routing steps} = \sum_{j=1}^{\log n}\sum_{i=1}^{j}\left\lceil\frac{2^{\lfloor i/2\rfloor}-1}{w*c}\right\rceil \geq O\left(\frac{\sqrt{n}}{w*c}\right)$$

$$\text{reconfiguring steps} = \sum_{i=4}^{\log n} i-3 = O(\log^2 n)$$

When n is large the asymptotic speed-up factor is $w*c$. If we employ only the interconnection pattern $I_{1,2}$ then the reconfiguration steps are reduced to one but the speed-up factor becomes $w*c / 2$.

5.4 K-fold Sorting

External sorting is expensive. The problem of sorting more data items than the number of processing elements is thus important. Knuth addressed that problem in [Knut73, p.241-242]. He pointed out that a sorting network of n data items can be generalized to sort $k*n$ data items if the comparison operation is replaced by a k-way merge operation. This generalization idea was applied to several sorting algorithms by G. Baudet and D. Stevenson in [Baud78].

To sort $k*n$ data items on n processing elements, the data items are initially distributed evenly to each processing element. The data sequence at each processing element is then sorted locally. Now, the sequence $Q = Q_1; Q_2; \ldots; Q_n$ is partially ordered, where Q_i is the sorted sequence of k elements at PE_i. For any sorting algorithm using only the comparison-interchange operation, Baudet and Stevenson proposed that it can be generalized to sort the partially ordered sequence Q by substituting the comparison-interchange operation with a merge-splitting operation. Performing the merge-splitting operation on two sequences Q_i and Q_j is to merge the two sequences and split into halves to produce the new occurrences of Q_i and Q_j.

Assume that m is the local memory size of processing elements on a CHiP computer, that is, each processing element can hold m data items.

The internal memory capacity is computed as $m*n$, provided that a CHiP region of n processing elements is allocated for the sorting. Only when the data items to be sorted exceed the internal memory capacity should we resort to external sorting. However, Baudet and Stevenson's generalization method does not work when $\frac{m}{2} < k \leq m$ since the merge-splitting operation needs at least $2k$ working space[†] at each processing element. We therefore consider two indexing schemes for the bitonic sort to sort as many as $m*n$ data items on n processing elements. The comparison-interchange operation does not have to be replaced by a merge-splitting operation; we simply perform k comparison-interchange steps.

The two indexing schemes are extensions to the shuffled row-major one. The processing elements are still indexed in the shuffled row-major order. Since there are k data items at each processing element, we need to index further those data items at the same processing element. Data items may be indexed in the following ways:

(1). Aggregation scheme – Index those data items at PE_i as $i*k$, $i*k+1$, ..., $i*k+(k-1)$.

(2) Projection scheme – Index those data items at PE_i as i, $n+i$, ..., $(k-1)*n+i$.

[†] Baudet and Stevenson used $3k$ working space at each processing element.

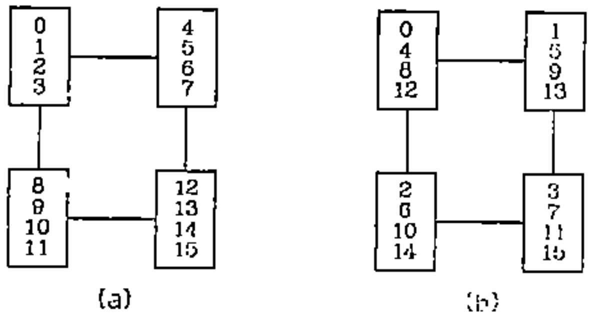


Figure 5-6. Indexing 16 data items on 4 processing elements: (a) Aggregation scheme, and (b) Projection scheme.

Assume that both k and n are powers of 2, $k = 2^p$ and $n = 2^q$. To sort the $k*n$ data items using the bitonic sort, $p+q$ stages of the bitonic merge are required. With the aggregation scheme, the first p stages are to sort locally each sequence Q_i of k elements at PE_i. Then, q more stages are performed to sort the partially ordered sequence $Q = Q_1; Q_2; \ldots; Q_n$. At the $(p+1)$-th, ..., and $(p+q)$-th stages, they all perform a local execution of the first p bitonic stages (see Figure A-1).

The bitonic sort with the aggregation scheme may be modified in two ways. Each execution of the first p bitonic merge stages can be replaced by a faster local sort. To perform k comparison-interchange steps between two processing elements may be improved by some overlapping of read/write and comparison instructions. Let c_0 be the local sorting time of k data items at each processing element, and c_1 be the time saved by integrating the k comparison-interchange steps. If the bitonic sort is directly applied without any modification then $c_0 = \frac{k}{4}(\log^2 k + \log k)$ and $c_1 = 0$.

Define $T(2^i,k)$ to denote the time required to merge the $k*2^i$ data items in the processing elements from 0 to 2^i-1, and $S(2^{2j},k)$ the time to sort the $k*2^{2j}$ items in the processing elements from 0 to $2^{2j}-1$. Notice that $T(1,k) = 0$ and $S(1,k) = c_0*t_C$. We analyze the time complexity of the bitonic sort with the aggregation scheme in the following:

$$\begin{cases} T(1,k) = 0, \\ T(2^i,k) = T(2^{i-1},k) + (k*2^{\lceil i/2 \rceil}-c_1)\, t_R + k\ t_C. \end{cases} \tag{5.1a}$$

$$\begin{cases} S_1(1,k) = c_0*t_C, \\ S_1(2^{2j},k) = S_1(2^{2(j-1)},k) + T(2^{2j-1},k) + T(2^{2j},k). \end{cases} \tag{5.1b}$$

Solve the recurrences 5.1a for the merge time function,

$$T(2^i,k) = \begin{cases} [k(3*2^{(i+1)/2}-4)-i*c_1]\, t_R + i*k\ t_C, \text{ if } i \text{ is odd} \\ [4k(2^{i/2}-1)-i*c_1]\, t_R + i*k\ t_C, \text{ if } i \text{ is even.} \end{cases} \tag{5.2}$$

Substitute the above equation into equation 5.1b,

$$\begin{cases} S_1(1,k) = c_0*t_C, \\ S_1(2^{2j},k) = S_1(2^{2j-1},k) + [7k*2^j - 8k - (4j-1)c_1]\, t_R + (4j-1)k\ t_C. \end{cases}$$

Solve the recurrences for the sorting time function;

$$S_1(n,k) = [14k(\sqrt{n}-1) - \frac{c_1}{2}\log^2 n - \frac{(8k+c_1)}{2}\log n]\, t_R + [\frac{k}{2}\log^2 n + \frac{(c_0+k)}{2}\log n]\, t_C. \tag{5.3}$$

With the projection scheme, the first q stages are equivalent to performing k runs of the bitonic sort of n data items on n processing elements. From another point of view, the first q stages with the projection scheme are the same as with the aggregation scheme except with $c_0 = 0$. The next p

stages are to merge the sequence $Q = Q_1; Q_2; \ldots; Q_k$, where Q_i is a sorted sequence of n data items over the n processing elements. Assume that both p and q are even numbers ($k = 2^p$ and $n = 2^q$). The required merge time at the $(q+1)$-th stage is $\frac{k}{2} t_C + T(n,k)$, and $k\ t_C + T(n,k)$ at the $(q+2)$-th stage. Let ∇S be the time for the p merge stages.

$$\begin{aligned}\nabla S(n,k) &= (1+2+\ldots+\frac{p}{2})(\frac{3}{2}k\ t_C + 2\,T(n,k)) \\ &= (\log^2 k + 2\log k)[k(\sqrt{n}-1) - \frac{c_1}{8}\log n]\ t_R + \\ &\quad (\log^2 k + 2\log k)[\frac{3k}{16} + \frac{k}{8}\log n]\ t_C\end{aligned}$$

The total time for the bitonic sort with the projection scheme is thus

$$S_2(n,k) = S_1(n,k)\,|_{c_0=0} + \nabla S(n,k). \qquad (5.4)$$

Comparing the equations 5.3 and 5.4, one finds that the comparison time might be reduced with the projection scheme, but the data routing time is definitely increased. Data routing time is the dominating factor in the time complexity of the bitonic sort with the mesh interconnection. Notice that when $k = 1$, $c_0 = c_1 = \nabla S(n,k) = 0$. In summary,

$$\begin{aligned}S_1(k*n,1) &= S_2(k*n,1) = O(\sqrt{k*n})\ t_R + O(\log^2 k + \log^2 n)\ t_C \\ S_1(n,k) &= O(k\sqrt{n})\ t_R + O(k\log^2 n + c_0\log n)\ t_C \\ S_2(n,k) &= O(k\log^2 k\ \sqrt{n})\ t_R + O(k\log^2 n + k\log^2 k\log n)\ t_C\end{aligned}$$

We conclude that the k-fold bitonic sort with the aggregation scheme outperforms the projection scheme assuming $c_0 \le O(k\log^2 k)$. The saving factor c_1 does not have any significant effect on the time complexities. The aggregation scheme emphasizes data locality while the projection scheme emphasizes parallelism. The former attempts to reduce the routing steps and the latter attempts to reduce the comparison steps. Only when $c_0 \to O(k^2)$ and k is large may the projection scheme be better than the aggregation scheme. In that situation, the sequential sorting time c_0 cannot be compensated by improving data locality.

CHAPTER 6

QUERY EMBEDDING

Partitioning a large problem into several small and more tractable subproblems, or divide-and-conquer, is a common approach in computing theory and practice. Subproblems are often referred to as basic operations. Existing algorithms for the basic operations are then applicable to solving many large problems. When each subproblem is very efficiently solved by highly parallel hardware, one interesting question is: What is the relative overhead of data movement among the basic operations?

One benefit of the CHiP computer is to imitate the performance efficiency of algorithmically specialized processors on the same devices. Owing to its configurable switch lattice, the CHiP computer is capable of embedding suitable interconnections for performing different algorithms efficiently. To solve a large and computationally intensive problem, several algorithms are usually involved. The configurability of the CHiP computer also provides a potential for composing those algorithms without producing any bottleneck of data movement [Snyd82].

Composing algorithms includes the embedding of interconnections on the switch lattices for individual algorithms and the embedding for harmonious interaction among the algorithms. In [Snyd82], an example of solving a system of linear equations is demonstrated. To solve the problem, one might need an algorithm for the LU-decomposition of the coefficient matrix and a linear recurrence solver to perform the backward substitution. Snyder showed the interconnection embeddings on a switch lattice which put together Kung and Leiserson's LU-decomposition algorithm and a systolic method for the backward substitution [Mead81, Ch.8.3].

To evaluate a database query, several database operations are usually invoked. Many efficient algorithms exist for implementing those database operations. Like solving a system of linear equations, techniques of composing algorithms might also be able to solve query evaluation effectively. However, I/O and data flow in query evaluation are much more complex. It is a multiphased problem that takes multiple relations as operands (possibly at different time) and produces a single relation as result. The problem structure as well as the problem size, moreover, varies for different database queries.

Query embedding is the idea of embedding suitable interconnections in order to process whole queries on the CHiP computer. It involves allocating a CHiP region and providing appropriate interconnections for efficiently inputting relations, solving the multiphased problem, and outputting the result. If query embedding is done in such a way as to embed individual operations separately and then to compose them together, the interconnections for routing results from one operation to the next operation may be far from being realistically embeddable. Fortunately the primitive operation POP-SORT which unifies many database operations gives a hope to avoid this difficulty. Composing algorithms would simply become putting different runs of the same algorithm together. The bitonic POP-SORT is especially suitable for query embedding since it works in a particularly regular manner. Actually, query embedding can be simplified significantly if database operations are implemented by the bitonic POP-SORT.

In this chapter we shall explore techniques of query embedding for processing whole queries in a highly parallel fashion on the CHiP computer. In Section 6.1 we parse algebraic queries into operation trees and discuss a general scheme of embedding the operation trees. Taking advantage of the unification provided by the primitive operation POP-SORT, we then demonstrate how simple query embedding can be done for a restricted type of operation trees in Section 6.2. Section 6.3 summarizes some general strategies for improving query embedding. We also extend the embedding techniques to evaluate all the algebraic operation trees in Section 6.4.

6.1 Embedding of Operation Trees

Query languages for the relational data model are based on two types of abstract languages: relational algebra and relational calculus [Ullm80, Ch.4]. Both abstract languages are equivalent in expressive power; calculus expressions can always be translated into algebraic expressions. It is trivial that an algebraic expression can be parsed into a tree of algebraic operations (Figure 6-1). In the operation trees, internal nodes represent algebraic operations and external nodes represent input relations. At the root node a final operation is performed and the result is produced.

Existing query languages are not necessarily the exact implementations of the abstract ones. They may have certain extensions to the abstract

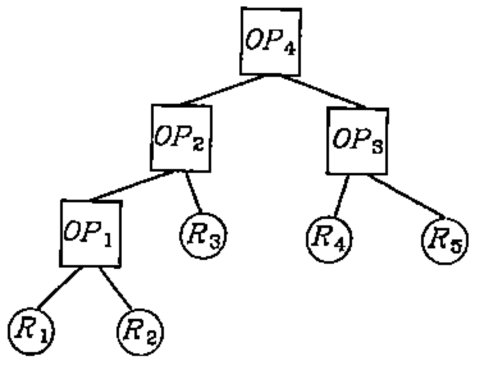


Figure 6-1. An operation tree from parsing a query.

languages, e.g. transitive closure, fixed point, and looping. In the most general case, queries are arbitrary functions on relations. Given any query, one can still represent it as an operation tree, but the operations are no longer restricted to algebraic ones. Nevertheless the abstract languages serve as a benchmark for evaluating existing query languages. Efficient evaluation of algebraic operation trees is thus very important in achieving fast query processing.

Since queries are represented and processed as operation trees, query embedding on the CHiP computer is reduced to the embedding of operation trees. To evaluate whole queries by embedding operation trees, a wide spectrum of parallelism is possible. We may have inter-operation and intra-operation parallelism in evaluating an operation tree. We may also have inter-query parallelism if the CHiP computer is big enough to host several queries. Furthermore, the I/O overhead can be minimized when whole queries are evaluated on the CHiP computer. Intermediate results tend to be kept in the CHiP processor, and therefore the data swapping between the CHiP processor and its external storage may be eliminated. The ideal case occurs when no I/O request is issued besides loading input relations onto the

CHiP processor and outputting the result relation.

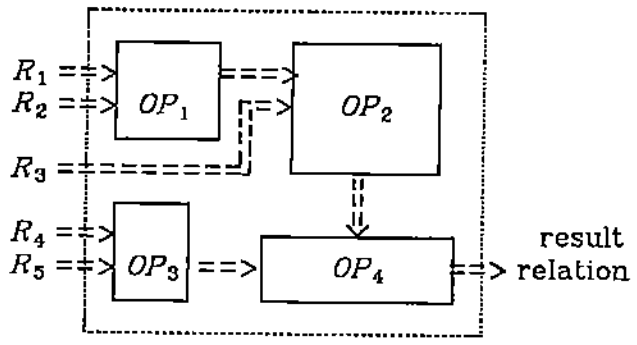


Figure 6-2. A general scheme of composing algorithms (operations) for query embedding.

To embed and execute an operation tree, a contiguous CHiP region is allocated. Within the region, interconnections are to be provided to perform the whole operation tree efficiently. A general scheme of doing this is as follows (Figure 6-2).

- First, allocate regions for embedding algorithmically specialized interconnections to perform individual operations.
- Secondly, tailor those regions as compactly as possible according to the I/O requirements and the communication requirements among the operations.

The CHiP region allocated for the whole operation tree, called the *query region*, is thus partitioned into three type of regions: *operation regions*, *connection regions*, and *I/O regions*. Operation regions are those allocated to embedding suitable interconnections for running efficient algorithms of the operations. Connection regions are those allocated to providing data paths from operations to operations. I/O regions connect some of the operation regions to the CHiP perimeter where the CHiP processor is connected to its external storage devices.

Two obvious optimization objectives for query embedding are to reduce the query region and to minimize the total time for evaluating the whole operation trees. To minimize the query region, operation regions should be kept as small as possible and they should be packed in such a way that the needed I/O regions and connection regions are also small. To minimize the total time, we want the query region to be large enough to provide interconnections for performing efficient algorithms and putting them together. The two objectives may not be achievable together. As space-time tradeoff is a common phenomenon in computing world, we may also find the trade-off between the two objectives.

The general scheme of embedding operation trees, as shown in Figure 6-2, provides a basic strategy of query embedding. Only when there is no better way would we resort to the general embedding scheme, since the general scheme is exposed to the following problems:

- The size of the result relation after preforming an operation depends on the operation itself and the distribution of data values. The amount of significant data items shrinks and swells during the query processing. It is nontrivial to allocate CHiP regions for the later operations.

- Large I/O regions are sometimes necessary. For example, a wide bandwidth is needed in order that OP_2 can read in R_3 fast, and the data paths may be long if OP_2 is buried far away from the CHiP perimeter.

- Optimal algorithms to pack operation regions are extremely difficult to find. Heuristics may result in requesting large connection and I/O regions.

6.2 Buddy System Allocation

Due to the dynamic character of query processing in which the amount of significant data items varies, the allocation of operation regions is subject to dynamic strategies. Unlike memory allocation, dynamic allocation of CHiP regions entails dynamic control of processing elements and dynamic provision of interconnections.. Moreover, problems like deadlock prevention, communication blockade between parent and child regions must be solved. To avoid the dynamic complication we therefore present area-effective static allocation policies in this section.

The bitonic POP-SORT which is an efficient primitive for many database operations also yields a nice solution to query embedding. In this section we demonstrate how well the bitonic POP-SORT can simplify query embedding on the CHiP computer. A restricted type of operation tree is considered. In Section 6.4, the embedding techniques are then extended to evaluate all algebraic operation trees.

Operation trees considered here may contain some or all of the following algebraic operations: restriction, projection, duplicate-removal, union, intersection, difference, and join. Cartesian product and quotient are two useful algebraic operations being left out. These two operations are extremely difficult in nature. They are not in the scope directly implementable by the primitive operation POP-SORT. Fortunately, quotients are not often executed in query processing and Cartesian products can often be replaced by joins [Wong76]. Hence the restricted type of operation trees still covers quite a portion of database queries.

Any POP-SORT has the following two important features:

- It employs marking functions to mark off all the unwanted data items.
- It works well even with some marked and unwanted data items in the input.

Suppose that there are parent and child operations which are all implemented by POP-SORT. The child operations can send the whole chunk of data possibly consisting of unwanted and marked items to the parent. The parent operation can then carry on without worrying about the marked-off items. These operations thus can be allocated CHiP regions by some static strategies.

Based on the two features of POP-SORT, another two valuable observations are:

Observation 1. Restrictions would just produce more marked items, and projections would reduce the tuple length. They can be combined with other operations that precede or follow them.

Observation 2. Remove-duplicates are already combined with union, intersection, and difference due to the versatility of POP-SORT. Thus the duplicate-removal before or after these three operations is redundant.

By merging the internal nodes according to the above observations, operation trees are shrunk to have only external nodes and those internal ones for operations excluding restriction and projection (and maybe duplicate-removal.) The allocation of operation regions now becomes the allocation for a smaller number of internal nodes.

The bitonic POP-SORT, in particular, works in a very regular manner. As for query embedding, mesh interconnection is chosen for the primitive operation in order to keep the operation regions small. Data items are assumed to be sorted into shuffled row-major order. For joins, data items are then rearranged into snake-like row-major order with a relatively insignificant overhead (Chapter 5.1). In addition to the two features mentioned before, the bitonic POP-SORT has another very important one:

- Assuming mesh interconnection and shuffled row-major indexing, the bitonic POP-SORT works well on a square region or a 1:2 rectangular region.

It is this feature that makes algorithms composition very simple. More observations implied by this feature are as follows:

Observation 3. The CHiP region for the parent operation can be overlaid with its child operation regions. No connection regions are necessary because data items are always in positions ready for next operation.

Observation 4. The parent operation may only need to execute a stage of the bitonic merge instead of the whole sorting procedure since the child operations would have sorted the data items in their regions.

As an analogy to the buddy system for dynamic storage allocation [Knut73, I, Chapter 2.5], we present a buddy system for static allocation of operation regions. Each operation node is allocated a CHiP region of size a power of 2. The two child operation regions are buddies. We do not insist that buddies be equal in size, but buddies must be located together. Merging two buddies becomes a larger region and the larger region is for the parent operation.

Algorithm 6-1: Buddy system allocation.

A.1. Compute area for each internal leaf node; $n_i + n_j \rightarrow 2^{\lceil \log (n_i+n_j) \rceil}$, where n_i and n_j are sizes of input relations.

A.2. Compute areas for remaining input relations; $n_i \rightarrow 2^{\lceil \log (n_i) \rceil}$.

A.3. Compute areas for parent nodes from areas of child nodes (buddies): $2^i + 2^j \rightarrow 2^{max(i,j)+1}$, where 2^i and 2^j are areas of buddies.

B.1. Allocate a query region for the root node.

B.2. Allocate one half of the parent region to each of its child regions.

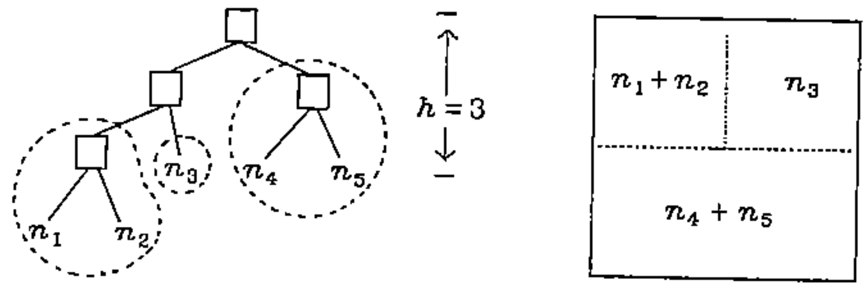


Figure 6-3. An example of buddy system allocation.

Algorithm 6-1 for buddy system allocation is composed of two phases. First, compute the areas of operation regions from the operation tree's bottom up. Secondly, allocate operation regions from the top down. In Figure 6-3 we show an example of buddy system allocation.

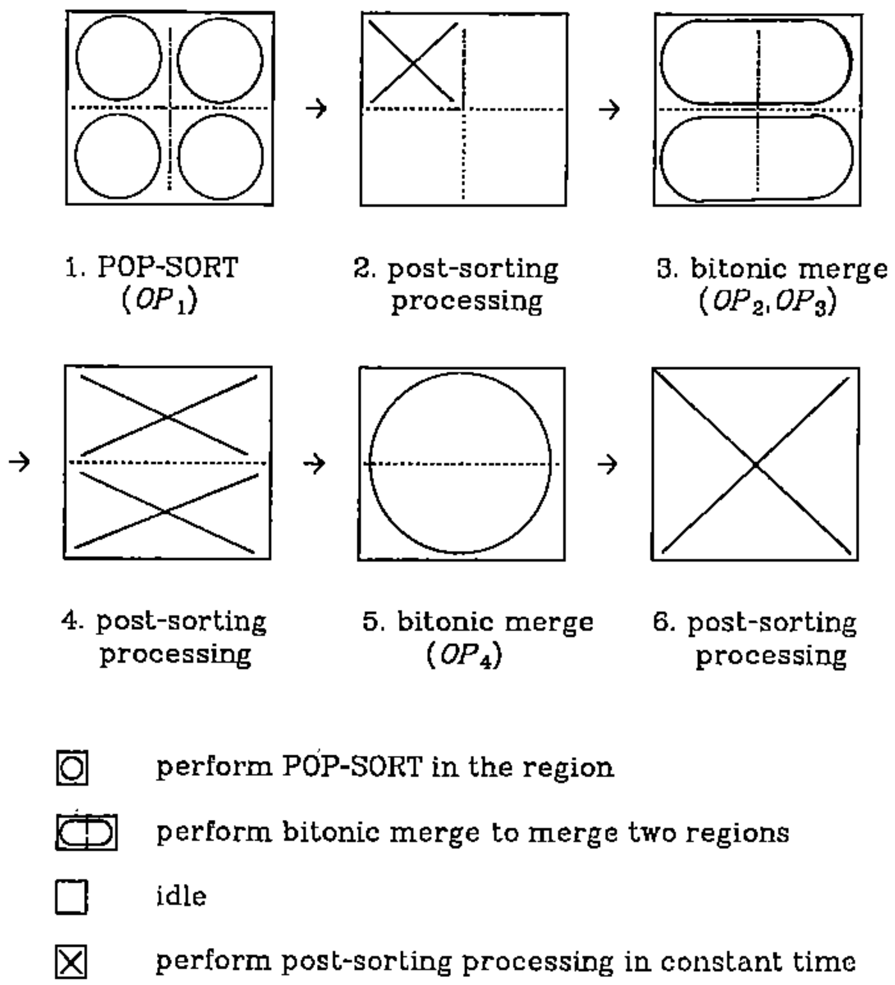

1. POP-SORT (OP_1)
2. post-sorting processing
3. bitonic merge (OP_2, OP_3)
4. post-sorting processing
5. bitonic merge (OP_4)
6. post-sorting processing
perform POP-SORT in the region
perform bitonic merge to merge two regions
idle
perform post-sorting processing in constant time

Figure 6-4. An example of processing a class of queries using the bitonic POP-SORT.

Processing the whole example query is partitioned into six phases in figure 6-4. Phases 1 and 2 would complete the execution of OP_1, phases 3 and 4 would complete OP_2 and OP_3 concurrently, and so on. In phase 1, the bitonic POP-SORT is performed in each "circled" region. In phase 2, the post-sorting processing is executed in the "crossed" region, and the rest region does nothing but wait. In phase 3, one stage of the bitonic merge is sufficient since the data items in the circled regions are already ordered as bitonic sequences.

It is surprising that the multiphased query processing can be viewed as a big sorting job interleaved with other processes. The post-sorting processing for union, intersection, or difference requires only constant time (see Chapter 3.1). For queries involving only these operations and restriction, projection, and duplicate-removal, the multiphased query processing thus works exactly like a big sorting job, except with some constant time processing. It guarantees a total processing time of $O(\sqrt{Q_{area}})$, where Q_{area} is the area of the query region.

Notice that the bitonic POP-SORT is also very helpful for some of the equi-join or natural operations. The post-sorting processing for those joins would involve (1) reordering from shuffled row-major index to snake-like row-major index, (2) performing easy-catch process, (3) running the sprinkle algorithm to resolve the hot spots problem, and (4) restoring the order of data items by running the bitonic POP-SORT again. For quite a few practical cases of joins, the post-sorting processing can be done in $O(\sqrt{n})$ time, where n is the area of the region on which the join operation is performed. Thus the total processing time is still $O(\sqrt{Q_{area}})$. However, not all join operations work well this way. We shall address this problem in Section 6.4.

The allocation algorithm 6-1 does not attempt to pack input relations in a compact fashion. It packs better when buddies are about of the same size. However there is a worst case anomaly:

Let $n_1 = 2^k$, and $n_2 = n_3 = n_4 = n_5 = 1$.

==> Total input size is $I_{siz} = 2^k + 4$.

Query region has area $Q_{area} = 2^{k+3}$ ($h = 3$).

==> Nearly $\frac{7}{8}$ of the query region is wasted!

In general, the area of a query region allocated by Algorithm 6-1 is within a range as follows:

$$I_{siz} \le Q_{area} < 2^h * I_{siz}.$$

To resolve the anomaly, we propose two approaches. One is to parse queries into "good" trees which would lead to more compact allocations. This is discussed in Section 6.3 – query amelioration. The other approach is to modify the buddy system allocation to pack input relations. A modified version of the buddy system allocation is shown in the following as Algorithm 6-2. This improved algorithm represents a packing technique based on the feature that the bitonic POP-SORT works well with shadow regions (see Chapter 5.2)

Algorithm 6-2: Modified buddy system allocation.

A.1. Transform the binary operation tree into a quaternary tree as shown in Figure 6-5.

B.1. Compute area for each internal leaf node; $n_{l1} + n_{l2} + n_{r1} + n_{r2} \rightarrow 2^{\lceil \log(n_{l1}+n_{l2}+n_{r1}+n_{r2}) \rceil}$, where n_{l1}, n_{l2} are sizes of left input relations, and n_{r1}, n_{r2} are sizes of right input relations.

B.2. Compute areas for remaining input relations; $n_i \rightarrow 2^{\lceil \log(n_i) \rceil}$.

B.3. Let 2^{l1}, 2^{l2}, 2^{r1}, 2^{r2} be areas for child nodes (buddies). Compute areas for parent nodes from areas of child nodes; $2^{l1} + 2^{l2} + 2^{r1} + 2^{r2} \rightarrow 2^i$, where 2^i is the smallest area that is large enough for all the four buddies.

C.1. Allocate a query region for the root node.

C.2. Let $0 \sim 2^i-1$ be a parent region. Allocate the regions 2^{l1}, 2^{l2} subsequently from 0 to 2^i-1, if $l1 \geq l2$. Allocate the regions 2^{r1}, 2^{r2} subsequently from 2^i-1 to 0, if $r1 \geq r2$.

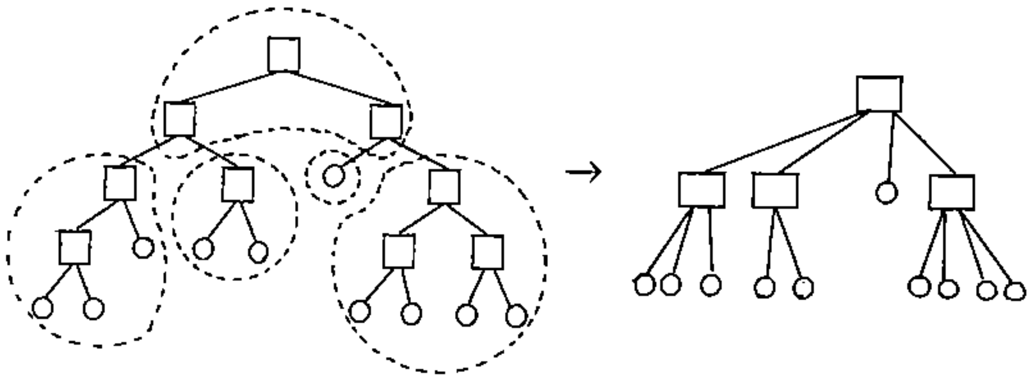


Figure 6-5. Transforming an operation tree into a quaternary tree for more compact allocation.

The modified allocation algorithm tries to pack input relations by grouping four buddies instead of just two. The success of the packing technique again relies on the relative sizes of buddies. However, the upper bound of Q_{area} is already improved significantly. In general,

$$I_{siz} \leq Q_{area} < 2^{h/2} * I_{siz},$$

since the height of a quaternary tree is reduces to $h/2$. Two child operations might be allocated regions of different areas by the packing attempt. The synchronization between two child operations therefore cannot count on the allocation of regions of the same area any more. The operation in a smaller region would need to wait for the other to finish.

Theoretically speaking, sorting with shadow regions can be exploited to a very complicated extent. It is then possible to compact input relations further. Nevertheless, the more compactly input relations are packed, the more complicated the required synchronization tends to be. It might not be worthwhile to pursue more compact packing since the operation trees considered are more likely small, i.e. the value of h is small. Moreover, we may turn to query amelioration for more compact allocations.

6.3 Query Amelioration

The total CHiP region and the total processing time are the two objects to be minimized for query embedding on the CHiP computer. Although "query optimization" is commonly used, query evaluation is not necessarily optimized over all the possible inputs. The term "query amelioration" would be more appropriate [Ullm80], especially when there are two interacting "optimization" objectives. Our query amelioration philosophy is to reduce the total time while still keeping the query region small. In this section, we shall summarize some general strategies for query amelioration. These strategies are from two sources: some by re-phrasing the algebraic expressions, the others based on other implementation considerations.

Queries may take a long time to execute, and the conventional execution time could be reduced greatly if the queries are rephrased according to some optimization criteria [Ullm80, Ch.6]. As a rule of thumb, the general strategies for optimization in [Ullm80, Ch.6] are also valuable on the CHiP computer. In particular, we summarize four rules for rephrasing algebraic expressions in order to reduce the CHiP region or the total processing time.

1. *Perform restrictions as early as possible.* Restrictions tend to make significant data items sparse so that more join operations can be performed by using POP-SORT. (See also Strategy 5.)

2. *Perform projections as early as possible.* Projections tend to reduce the tuple length, therefore reduce the amount of data flow in CHiP processor. (See also Strategy 5.)

3. *Cascade restrictions and projections.* A sequence of these operations can be performed all at a once.

4. *Combine certain restrictions with their prior Cartesian products into joins.* This helps controlling the size of intermediate results. Hopefully some allocation of large CHiP regions can be avoided.

Among the equivalent expressions there are some which usually take longer time than the others. The goal of rephrasing an expression is to avoid those more time-consuming ones. The first two strategies are feasible by commuting restriction with other operations, or by commuting projection with a Cartesian product, join, union, or intersection (but not difference.) Strategy 4 is in a sense a special example of Strategy 1.

The way in which a particular expression is evaluated also affects the query processing time. We summarize more strategies based on the implementation considerations in the following.

5. *Perform restrictions and projections on the input relations on the mass storage level.* To reduce the input size and the amount of data flow in the CHiP processor, the restriction and projection on an input relation is better performed on the mass storage level using the approaches as in the conventional database machines.

6. *Combine restrictions and projections with other operations that follow or precede them.* It can be done by loading the restriction predicates and projection attributes on the processing elements. This simplify the allocation of CHiP regions.

7. *Delete redundant duplicate-removal.* Multiset operands are allowed to union, intersection, and difference. Remove-duplicates before these operations is thus redundant.

8. *Combine a sequence of unions.* A single run of POP-SORT on all the operand relations would complete a sequence of unions.

9. *Parse queries into operation trees in a weight-balanced fashion (or process small relations first.)* The buddy system allocation algorithms work especially well when buddies are about of the same sizes.

10. *Load input relations as an ensemble.* Input relations are loaded together onto CHiP processor according to the allocation pattern. I/O time of $O(\sqrt{Q_{area}})$ is thus guaranteed.

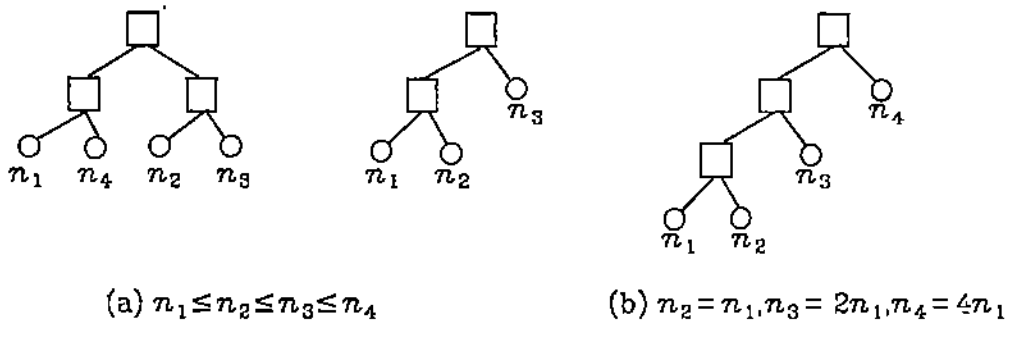

n_1
n_4
n_2
n_3
n_1
n_2
n_3
n_1
n_2
n_3
n_4
(a) $n_1 \le n_2 \le n_3 \le n_4$
(b) $n_2 = n_1, n_3 = 2n_1, n_4 = 4n_1$

Figure 6-6. Weight-balanced trees.

Commutative laws and associative laws for unions, intersections, joins, or Cartesian products are the weapons that we may use to parse queries in a weight-balanced fashion. Strategy 8 presents an even better amelioration method on performing a sequence of unions. It is feasible because of the versatility of POP-SORT to perform union on multisets. For a sequence of intersections or a sequence of joins (Cartesian products), Strategy 9 can be applied to parse the operation sequence into a weight-balanced tree. Examples are shown in Figure 6-6. For a sequence of differences, Strategy 9 is also useful because of the equivalent law: For any i, $1 \leq i < k$, $R_1 \Delta R_2 \Delta \cdots \Delta R_k \equiv R_1 \Delta \cdots \Delta R_i \Delta (\bigcup_{j=i+1}^{k} R_j)$, where Δ denotes the multiset operation difference with left-to-right precedence and $\bigcup_{j=i+1}^{k} R_j$ denotes the union of multisets. Assume that the examples in Figure 6-6 show the weight-balanced parsing of a sequence of differences. It is interesting to note that the operation nodes on the path from the external node n_1 to the root all perform differences and the rest all perform unions.

6.4 Extensions

Although the operation trees considered in Section 6.2 may contain join operations, not all the join operations work well using POP-SORT with mesh interconnection on the CHiP computer. Equi-join and natural join are more likely to work than other join operations. However, even for equi-join or natural join, performance may degrade due to the hot spots problem.

In this section we shall present a method of performing Cartesian product that generates the result relation in a square CHiP region. Join operations can be implemented, at worst, as Cartesian products. Quotient can also be implemented by Cartesian product, difference, and projection. Adding Cartesian product to the restricted type of operation trees, we therefore extend the query embedding techniques presented in section 2 to evaluate all algebraic operation trees. Similarly, we may extend further to include operations other than algebraic ones.

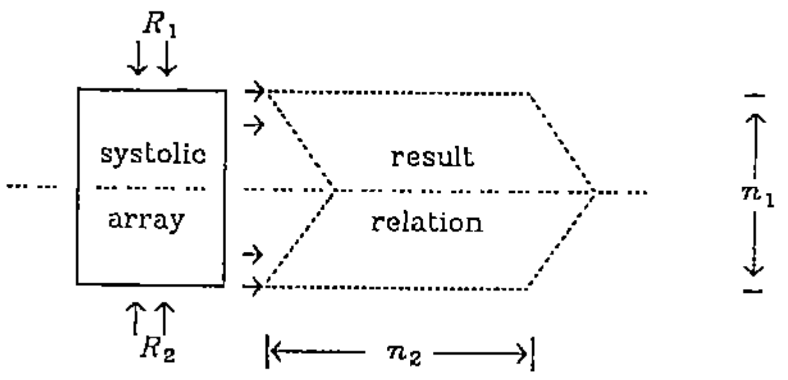


Figure 6-7. Systolic method of Cartesian product.

The systolic method of Cartesian product in [Kung80] produces a result relation in a -- -shaped region (see Figure 6-7). In order to simplify the composition of Cartesian product with other operations implemented by POP-SORT, the result relation needs to be in a square region (or a 1:2 rectangular region.) A simple modification of the systolic method can shape the result relations in square regions. However, the I/O bandwidth of a CHiP processor is assumed proportional to its perimeter. We shall seek a faster algorithm which takes advantage of the I/O ports on the perimeter processing elements.

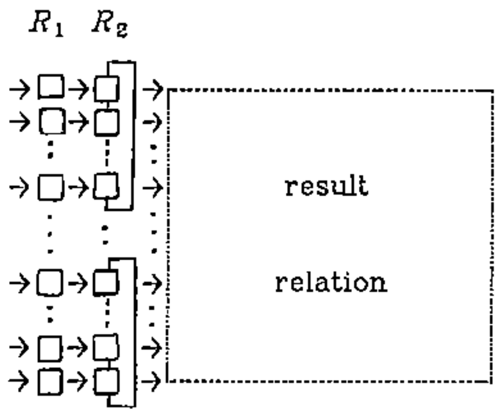


Figure 6-8. Cartesian product in a square region.

Let R_1 and R_2 be the two relations, $|R_1| = n_1$, $|R_2| = n_2$, and $n_1 \leq n_2$. The square region to hold the result relation requires a size of n_r^2, where $n_r = \lceil \sqrt{n_1 * n_2} \rceil$. Notice that $n_1 \leq n_r \leq n_2$. Assume that $n_r = k * n_1$ for simplicity. First, allocate a query region of area $(n_r + 2) * n_r$. The first two columns, called the processing columns, are used to produce result tuples. The rest of the region performs only left-to-right shift and is used to store the result relation. The following algorithm would generate the Cartesian product of R_1 and R_2 in a square region (see Figure 6-8) in $O(n_r)$ time.

Algorithm 6-3: Cartesian product.

1. Load k copies of R_1 on the second processing column.
2. If no more R_2 tuples then stop. Otherwise, load another column of R_2 tuples on the first processing column.
3. Rotate n_1 steps each copy of R_1 and produce n_1 columns of result tuples. Go to step 2.

Given any algebraic expression, we may proceed to do the following to evaluate the expression. First, rephrase it according to the query amelioration Strategies 1~4 summarized in Section 6.3. The rephrased expression is then parsed into an operation tree possibily containing some harder-than-sorting operations like Cartesian products, differences, and some joins. Those operations, not belonging to the restricted type, would partition the operation tree into single operation nodes and subtrees. Each subtree is then of the restricted type. The query amelioration strategies in Section 6.3 and the query embedding techniques in Section 6.2 are thus applicable to each subtree. Single operation nodes can be implemented by the method of producing a Cartesian product in a square region. For example, join operations that are not suitable for the easy-catch implementation can be implemented this way. Significant data items can then be "cornered" to a smaller square region by performing the bitonic POP-SORT.

To process any algebraic query, the composition of algorithms is no longer automatically done by the buddy system allocation. Composing algorithms, in a most general sense, thus becomes a three level approach. Ranked in the order of preference, they are: (1) buddy system allocation, (2) the general scheme shown in Section 6.1, and (3) off-CHiP processing, the last resort.

CHAPTER 7

SUMMARY AND CONCLUSIONS

This thesis applies highly parallel database machines to improve relational database processing. The database machines are dedicated computers enhanced with highly parallel processing capability. Regularity and uniformity are mandatory for achieving high-performance and cost-effectiveness. This work first deals with unifying several relational operations on a regular sorting algorithm.

Given a highly parallel processor, we have shown that any sorting algorithm designed to execute on the processor can be easily modified to become a primitive operation POP-SORT. The primitive operation can efficiently perform sorting, duplicate-removal, union, intersection, and difference. It can also be used to perform join operations requiring no more than linear post-sorting processing time.

This thesis then applies an instance of POP-SORT, the bitonic POP-SORT, on the CHiP processors to process whole queries. Query embedding presents a methodology which embeds appropriate interconnections for processing whole queries on the CHiP processors. Due to some interesting characteristics of the bitonic POP-SORT, query embedding is simplified significantly.

In Section 7.1 we summarize the main contributions of this thesis. To apply the results of this work successfully to a complete design of back-end system, several important issues need to be investigated further. Section 7.2 discusses briefly those important issues.

7.1 Main Contributions

The main contributions of this thesis are summarized in the following.

1. The methodology of applying sorting to solve other database operations is presented for highly parallel situations. Two techniques are shown to adapt, with negligible overhead, merge-oriented and other sorting methods to solve several database operations (POP-SORT).

2. The efficiency of POP-SORT is studied. POP-SORT based on an optimal sorting method is proved to be also optimal for performing duplicate-removal, union, intersection, and difference for a reasonable class of homogeneous comparison computation.

3. The join system which employs a halting mechanism can terminate join operations in sublinear time after the argument relations are pre-conditioned by POP-SORT.

4. The algorithm Sprinkle can efficiently redistribute data items such that they are almost evenly distributed over the processing elements.

5. The bitonic POP-SORT which generalizes Batcher's bitonic sort to become a powerful primitive defines the (new) upper time bound for performing each of the five database operations — sorting, duplicate-removal, union, intersection, and difference.

6. The bitonic sort on the mesh-connected computers in [Thom77, Nass79] can be improved by a speed-up factor up to $w*c$ with mesh-like interconnections on the CHiP computers.

7. Efficient algorithms for reordering data items among three major indexing schemes and sorting $k*n$ data items on n PEs are proposed and analyzed.

8. Query embedding is to exploit all the possible parallelism in processing whole queries. With the use of the bitonic POP-SORT, query embedding for a restricted type of queries is simple and straightforward. The algorithm to produce Cartesian products in square regions further extends the restricted query embedding to processing all the queries.

9 The lacing technique is shown to exploit the maximum number of data paths provided by the switch corridors on the CHiP computers.

7.2 Future Research

In this thesis we have concentrated on exploring parallelism in processing relational queries with the use of highly parallel processors. There are other issues needed to be investigated to complete a reliable design of a highly parallel database machine.

The I/O bandwidth between the mass storage and the highly parallel processor is required to be large to prevent the processor from data starvation. The mass storage should be content addressable such that searching and update can be performed on the storage level efficiently. The design of hardware organizations to implement the requirements of large I/O bandwidth and content addressability is thus important. More pressing, a storage model and an I/O model for near future technologies are necessary to measure I/O complexities and design problem decomposition algorithms.

Given a highly parallel processor with n PEs, we addressed the problem of partitioning the processor to perform several small jobs. We also presented algorithms to allow the total size of argument relations to be $k*n$ if PEs have local memory space k. However, problems with sizes larger than $k*n$ must be decomposed into several small ones. Fortunately the decomposition problem is reduced to an external sorting problem for a family of queries.

The back-end is dedicated to perform database management functions. With new hardware technologies and architectures the traditional designs of database management need to be reconsidered. Programming the highly parallel processor is another important issue. With the unification proposed in this research work the programming difficulty should be reduced.

LIST OF REFERENCES

LIST OF REFERENCES

Astr76	M. M. Astrahan et al., "System R: Relational Approach to Database Management," *ACM Trans. on Database Systems*, **1:2**, June 1976, 97-137.
Babb79	E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," *ACM Trans. on Database Systems*, **4**:1, March 1979, 1-29.
Bane78	J. Banerjee, and D. K. Hsiao, "Concepts and Capabilities of a Database Computer," *ACM Trans. on Database Systems*, **3:4**, December 1978.
Bane79	J. Banerjee, D. K. Hsiao, and K. Kannan, "DBC - A database computer for very large databases" *IEEE Trans. on Computers*, **C-28:6**, June 1979.
Barn68	G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes, "The ILLIAC IV computer," *IEEE Trans. on Computers*, **C-17:8**, August 1968, 746-757.
Batc68	K. E. Batcher, "Sorting networks and their applications," *Proc. 1968 National Computer Conference*, AFIPS, 307-313.
Batc80	K. E. Batcher, "Design of a Massive Parallel Processor," *IEEE Trans. on Computers*, **C-29:9**, September 1980, 836-840.
Baud78	G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," *IEEE Trans. on Computers*, **C-27**:1, January 1978, 84-87.

Bent79 J. L. Bentley and H. T. Kung, "A Tree Machine for Searching Problems," *IEEE Intl. Conf. on Parallel Processing*, August 1979.

Berr79 P. B. Berra and E. Oliver, "The Role of Associative Array Processors in Data Base Machine Architecture," *IEEE Computer*, March 1979.

Bila81 G. Bilarde, M. Pracchi, and F. P. Preparata, "A Critique and an Appraisal of VLSI Models of Computation," *Carnegie-Mellon Conf. on VLSI Systems and Computations*, October 1981.

Bora82 H. Boral and D. J. DeWitt, "Applying Data Flow Techniques to Data Base Machines," *IEEE Computer*, **15:8**, August 1982, 57-63.

Boro82 A. Borodin and J. E. Hopcroft, "Routing, Merging and Sorting on Parallel Models of Computation," *ACM 14th Annual Sym. on Theory of Computing*, 1982, 338-344.

Brow80 S. A. Browning, "The Tree Machine: A Highly Concurrent Computing Environment," Ph.d. thesis, California Institute of Technology, Computer Science Dept., January 1980.

Cana74 R. H. Canaday et al., "A back-end computer for database management," *Comm. ACM*, **17:10**, October 1974, 575-582.

Codd70 E.F. Codd, "A relatioal model of data for large shared data banks," *Comm. ACM*, June 1970, 377-387.

Codd82 E.F. Codd, "Relational Database: A Practical Foundation for Productivity," *Comm. ACM*, February 1982, 109-117.

DeWi79 David J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," *IEEE Trans. on Computers*, **C-28:6**, June 1979, 395-406.

DeWi82 David J. DeWitt, "Applying Data Flow Techniques to Data Base Machines," *IEEE Computer*, **15:8**, August 1982, 57-64.

Fort78 S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," *Pro. 10th Annu. ACM Sym. on Theory of Computing*, 1978, 114-118.

Fost80	M. J. Foster and H. T. Kung, "The Design of Special-purpose VLSI Chips," *IEEE Computer*, **13**:1, Jan. 1980, 26-40.
Gold78	L. Goldschlager, "A Unified Approach to Models of Synchronous Parallel Machines," *Pro. 10th Annu. ACM Sym. on Theory of Computing*, 1978, 89-94.
Hayn82	Leonard S. Haynes, "Highly Parallel Computing: Guest Editor's Introduction," *IEEE Computer*, **15:1**, (A special issue on Highly Parallel Computing,) January 1982.
Hirs78	D.S. Hirschberg, "Fast Parallel Sorting Algorithms," *Comm. ACM*, **21:8**, August 1978, 657-661.
HsiC81	Ching C. Hsiao and Lawrence Snyder, "VLSI Algorithms for Relational Database Operations," *Technical Report CSD-TR-375*, Departments of Computer Sciences, Purdue University, October 1981.
HsiD79	David Hsiao and M. J. Menon, "The Post Processing Functions of a Database Computer," Technical Report TR-79-6, Computer and Information Science Research Center, The Ohio State University, July 1979.
Hoey80	D. Hoey and C. E. Leiserson, "A Layout for the Shuffle-Exchange Network," *Pro. 1980 Intern. Conf. on Parallel Processing*.
Klei81	D. K. Kleitman, T. Leighton, M. Lepley and G. Miller, "New Layouts for the Shuffle-Exchange Graph," *Pro. of 13th Annu. ACM Sym. on Theory of Computing*, May 1981, 278-292.
Knut73	Donald E. Knuth, *The art of computer programming*, Addison Wesley, Vol. 1 & 3.
Kung79	H. T. Kung, "Let's Design Algorithms for VLSI Systems," *Proc. Conf. VLSI: Architecture, Design, Fabrication*, California Institute of Technology, Jan. 1979, 65-90.
Kung80	H. T. Kung and Philip L. Lehman, "Systolic (VLSI) Arrays for Relational Database Operations," *ACM SIGMOD International conference* 1980.

Kung82 H. T. Kung, "Why Systolic Architecture," *IEEE Computer*, **15**:1, January 1982, 37-46.

Lang78 G. G. Langdon Jr., "A Note on Associative Processors for Data Management," *ACM Trans. on Database Systems*, **3**:2, June 1978, 148-158.

Lehm81 P. L. Lehman, "A Systolic (VLSI) Array for Processing Simple Relational Queries," *1981 CMU Conference on VLSI Systems and Computations*, 285-295.

Lev81 G. Lev, N. Pipenger, and L. G. Valiant, "A Fast Parallel Algorithm for Routing in Permutation Networks," *IEEE Trans. on Computers*, 1981.

Lin76 C.S. Lin, D.C.P. Smith, and J.M. Smith, "The Design of a Rotating Associative Memory for Relational Database Applications," *ACM Trans. on Database Systems*, **1**:1, March 1976, 53-65.

Lint81 Bernard Lint and Tilak Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," *IEEE Trans. on Software Engineering*, March 1981, 174-188.

Lipt81 Richard J. Lipton and Robert Sedgewick, "Lower Bounds for VLSI," *Proc. of 13th Annu. ACM Sym. on Theory of Computing*, May 1981, 300-307.

Mead80 Carver Mead and Lynn Conway, *Introduction to VLSI systems*, Addison-Wesley, 1980

Mull75 David E. Muller and Franco. P. Preparata, "Bounds to Complexities of Networks for Sorting and for Switching," *J. ACM*, **22**:2, April 1975, 195-201.

Nass79 David Nassimi and Sartaj Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Trans. on Computers*, **C-27**:1, January 1979.

Ozka75 E.A. Ozkarahan, S.A. Schuster, and K.C. Smith, "RAP - An associative processor for data base management," *Proc. 1975 National Computer Conference*, AFIPS, 379-387.

Pate81	M. S. Paterson, W. L. Ruzzo, and L. Snyder, "Bounds on Minimax Edge Length for Complete Binary Trees," *Pro. of 13th Annu. ACM Sym. on Theory of Computing*, May 1981, 293-399.
Prep78	Franco. P. Preparata, "New Parallel-Sorting Schemes," *IEEE Trans. on Computers*, **C-27:7**, July 1978, 669-673.
Prep81	Franco P. Preparata and Jean Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Comm. ACM*, **24:5**, May 1981, 300-309.
Schw80	J. T. Schwartz, "Ultracomputer," *ACM Trans. on Programming Languages*, **2:4**, October 1980, 484-521.
Schu79	S. A. Schuster, H. B. Nguyen, E. A. Ozkarahan, and K. C. Smith, "RAP.2 - An Associative Processor for Databases and its Applications," *IEEE Trans. on Computers*, **C-28:6**, June 1979, 446-457.
Slot70	D. L. Slotnick, "Logic per track divices," *Advances in Computers*, **10**, J. Tou, Ed., Academic Press, 1970, 291-296.
Snyd81	Lawrence Snyder, "Programming Processor Interconnection Structures," *Technical Report CSD-TR-381*, Dep. of Computer Sciences, Purdue University, October 1981.
Snyd82	Lawrence Snyder, "Introduction to the Configurable, Highly Parallel Computers," *IEEE Computer*, **15:1**, 47-56.
Song80	S. W. Song, "A Highly Concurrent Tree Machine for Database Applications" *IEEE Intl. Conf. on Parallel Processing*, 1980.
Song81	S. W. Song, "On a High-Performance VLSI Solution to Database Problems," Ph. D. Thesis, Computer Science Dept., Carnegie-Mellon Univ., August 1981.
Ston71	Harold S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. on Computers*, **C-20:2**, 1971.
StoB76	M. R. Stonebraker et al., "The design and implementation of INGRES," *ACM Trans. on Database Systems*, **1:3**, September 1976, 189-222.

Su75	S.Y.W. Su, and G.J. Lipovski,. "CASSM: A cellular system for very large databases," *Proc. ACM Intl. Conf. on Very Large Databases*, 1975, 456-472.
Su79	S.Y.W. Su, L.H.Nguyen, A. Eman, and G.J. Lipovski, "The Architectural Features and Implementation Techniques of the Multicell CASSM," *IEEE Trans. on Computers*, **C-28:6**, June 1979, 430-445.
Thom77	C. D. Thompson and H. T. Kung, "Sorting on a Mesh-connected Parallel Computer," *Comm. ACM*, **20:4**, 1977.
Thom80	C. D. Thompson, *A Complexity Theory for VLSI*, Ph.D. Thesis, Carnegie-Mellon University, Computer Science Dept., August 1980.
Ullm80	J. D. Ullman, *Principle of Database Systems*, Computer Science Press, 1980, Chapters 4 & 6.
Vali75	Leslie G. valiant, "Parallelism in Comparison Problems," *SIAM Journal of Computing*, **4:3**, Setember 1975, 348-355.
Wong76	E. Wong and K. Youssefi, "Decomposition - a strategy for query processing," *ACM Trans. on Database Systems*, **1:3**, September 1976, 223-241.

APPENDICES

APPENDIX A

BATCHER'S BITONIC SORT

A.1 The Bitonic Merge

Batcher's bitonic sort [Batc68] is based on his bitonic merge algorithm which sorts bitonic sequences. A sequence $x_0, x_1, \dots, x_{n-1}$ is said to be bitonic if either

(1) there is an index i, $0 \le i \le n-1$, such that $x_0 \le x_1 \le .. .. \le x_i \ge x_{i+1} \ge .. \ge x_{n-1}$ or

(2) the sequence can be shifted cyclically so that condition 1 is satisfied [Ston71].

The bitonic merge algorithm applies $\log n$ parallel comparison steps. Each step partitions a bitonic sequence into low and high bitonic sequences such that every item in the low sequence is no larger than any one in the high sequence. The correctness of the bitonic merge algorithm was proved in [Batc68] and described as the following theorem in [Ston71].

Batcher's Theorem Let the sequence $x_0, x_1, \dots, x_{n-1}$ be bitonic and $a_i = \min(x_i, x_{i+n/2})$, $b_i = \max(x_i, x_{i+n/2})$ for $0 \le i \le n/2$. The two sequences $a_0, a_1, \dots, a_{n/2-1}$ and $b_0, b_1, \dots, b_{n/2-1}$ are both bitonic, and $a_i \le b_j$ for all i and j.

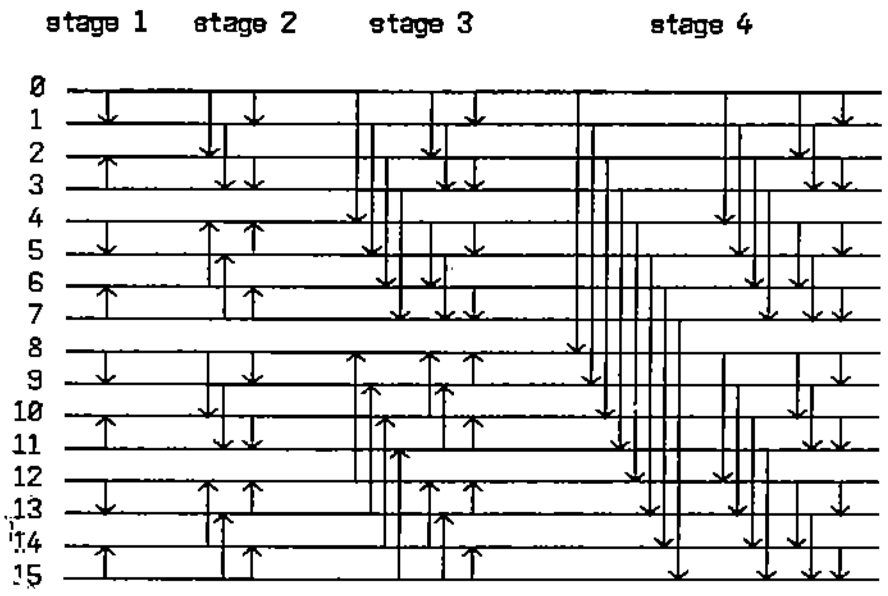


Figure A-1. Sorting network for Batcher's bitonic sort.

Batcher's bitonic sort applies his bitonic merge algorithm for $\log n$ times. It can be described as a sorting network shown in Figure A-1 [Knut73, p.237]. There are $\log n$ bitonic merge stages and $\log i$ comparison steps at the i-th stage. Several adapted versions of the bitonic sort with different processor interconnections are available.

- The bitonic sort can be done in time $(\log^2 n)\, t_R + \frac{1}{2}(\log^2 n + \log n)\, t_C$† with the perfect shuffle interconnection [Ston71].
- On a mesh-connected computer, the bitonic sort can be done in time $(14(\sqrt{n}-1) - 4\log n)\, t_R + \frac{1}{2}(\log^2 n + \log n)\, t_C$ if the sequence is to be sorted into the shuffled row-major order [Thom77].

† The actual time required for one routing step or one comparison step depends on different interconnections or machines.

- On mesh-connected computers, the bitonic sort can be done in time $(14(\sqrt{n}-1)-4\log n)t_R + \frac{1}{2}(\log^2 n + \log n)t_C + (\frac{9}{8}\log^2 n + \frac{3}{4}\log n)t_I$‡ if the sequence is to be sorted into the row-major or the snake-like row-major order [Nass79].

- With the CCC (or the k-Cube) interconnection the bitonic sort can be done in time $O(\log^2 n)(t_R + t_C)$ [Prep81].

A.2 Effects of Propagation Delay

In practice, sending data from a location to another location always takes time. The speed of data propagation in the chip is many orders of magnitude slower than the velocity of light [Mead80]. For circuit performance, the propagation delay in long wires is especially important. However, it is still controversial how the propagation delay affects the VLSI computation time. Depending on assumptions, the propagation delay function $p(x)$ varies widely: $O(\log x) \le p(x) \le O(x^2)$, where x is the wire length [Pate81].

Three interesting propagation delay models discussed in [Pate81, Bila81] are described as follows. It is plausible that R-C circuits define a quadratic propagation delay function. Since resistance and capacitance both grow linearly with the wire length, the time constant of the transistor load is thus a quadratic function of the wire length. However, the propagation delay is about defined by a linear function, provided repeaters are added to the long wires. If special drivers are used to speed capacitance

‡ t_I denotes the time required to interchange the contents of two registers.

charging then the minimum delay, $p(x) = \log x$, is achievable [Mead80].

According to [Bila81], both current and projected silicon technologies fall within the realm of the logarithmic propagation delay function. However, the special drivers cannot be unlimitedly applied to arbitrarily long wires due to the limitations on the current density. Thus the most realistic estimate is perhaps $P(x) = \sqrt{x}$ [Pate81].

In analyzing the asymptotic time complexities, one should be particularly careful about the effect of propagation delay since the maximum wire length may grow with the problem size. Taking the propagation delay into account, we recompute here the complexities for the bitonic sort with different processor interconnections. We consider only the communication time because the computation time is not so susceptible to the propagation delay as the former. The following table summarizes the results of our recomputation.

Table A-1. Effects of propagation delay on the bitonic sort with different interconnections.

$p(x)$	1	$c_1 \log x$	$c_2 \sqrt{x}$
shuffle	$\log^2 n$	$c_1 \log^3 n$	$c_2 \sqrt{n} \log n$
mesh	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$
CHiP	$\frac{1}{s}\sqrt{n}$	$\frac{\alpha}{s}\sqrt{n}$	$\frac{\alpha}{s}\sqrt{n}$

The average length of the edges in a planar embedding of the shuffle-exchange graph is $O(n / \log^2 n)$ [Thom80]. If $p(x) = c_1 \log x$ then the computation time is $c_1 \log(n / \log^2 n) + \log^2 n$ which is approximately $c_1 \log^3 n$. If $p(x) = c_2 \sqrt{x}$ then the computation time is $c_2 \sqrt{n / \log^2 n} + \log^2 n$ which is

$c_2\sqrt{n}\log n$. Other powerful interconnections like the CCC and the k-Cube should be as susceptible to the propagation delay as the shuffle-exchange.

On the CHiP computers, the technique of z-location jumps can be used to improve the data routing on the mesh-connected computers (Section 5.3). The improvement asymptotically achieves a speed-up factor up to z, $z \leq w{*}c$, where w is the corridor width and c the cross-over capability of the switches. Let s denote the speed-up factor, and thus $s \leq z \leq w{*}c$. In the table, we introduce another factor α which denotes the propagation delay required by the z-location jumps. Due to the practical consideration of high utilization of components, $w{*}c$ is bounded by a small constant, say 32. The propagation delay of the z-location jumps is thus relatively constant. Assuming that 1-location jumps take unit time, the factor α should be small, $\alpha \rightarrow 1$. Hence we do not distinguish two α's for the two non-constant propagation delay functions. The CHiP computers may also embed the shuffle-exchange interconnection, but the effect of propagation delay will be more severe than that on the shuffle-exchange as shown in the table.

APPENDIX B

SPRINKLE ALGORITHM

Given n processing elements $PE_0, PE_1, \ldots, PE_{n-1}$ and a sequence of data items distributed over the processing elements. The quantity of data items is not evenly distributed; there are x_i items at PE_i for all i in $[0, n-1]$. The Sprinkle Algorithm is designed to redistribute the data items so that the sequence is approximately equally distributed over the processing elements. That is, $\left\lfloor \frac{1}{n}\sum_{i=0}^{n-1} x_i \right\rfloor \le \hat{x}_i \le \left\lceil \frac{1}{n}\sum_{i=0}^{n-1} x_i \right\rceil$, where $\hat{x}_i$ is the number of data items at PE_i after the redistribution.

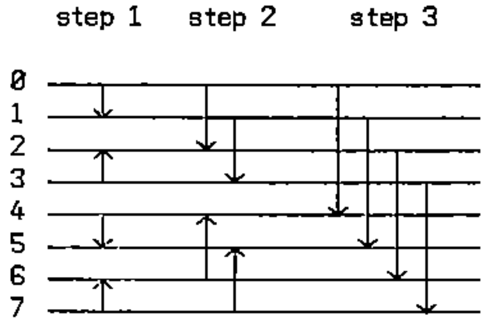
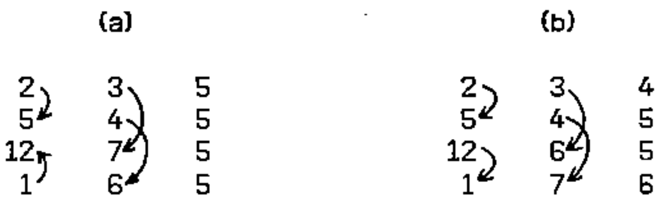


Figure B-1. The communication scheme of $\log n$ steps applied in the Sprinkle Algorithm for $n = 8$.

The redistribution problem is more difficult than the problem of finding the average of the number sequence $\{x_i\}$ which can be best done in $O(\log n)$ steps. However, a communication scheme of $O(\log n)$ steps as shown in

Figure B-1 is still sufficient for the redistribution problem. In Figure B-1, each arrow denotes an operation that redistributes the data items at the two PEs. After the operation, the two PEs both have the same number of data items, or the one pointed by the arrow head has one more data item. In the following example we show how important the directions of the arrows are.

Example. To redistribute data items among four PEs, the following figures shows (a) the communication scheme leading to the correct result, and (b) a communication scheme possibly leading to wrong results.



The Sprinkle Algorithm repeatedly compares the numbers of data items between pairs of PEs and then ships data items to redistribute them approximately evenly between the pairs of PEs. It involves $\log n$ computation steps to determine the redistribution strategies. In addition, the data shipment requires more communication time which depends on the original distribution $\{x_i\}$ and the available processor interconnections. Assume that $0 \le x_i \le k$, where k is a small constant. The Sprinkle Algorithm needs at most $(\frac{k}{2}+1) * \log n \ t_R$ communication time if the appropriate interconnection is provided. With the mesh interconnection, the communication time required is no more than $(\frac{k}{2}+1) * 4\sqrt{n} \ t_R$.

The communication scheme in Figure B-1 is similar to and is actually a portion of that needed in the bitonic sort (Appendix A). Programming the CHiP computers to perform the Sprinkle Algorithm thus does not demand much additional effort.

APPENDIX C

THE LACING TECHNIQUE

On the CHiP computers, the maximum number of data paths allowed to cross the corridors is bounded by $w*c$, with the corridor width w and cross-over capability c of the switches. If $d > 2c$, where d is the degree of incident data paths to the switches, then the maximum bandwidth $w*c$ is feasible with an embedding technique called lacing. The technique is to embed straight data paths as well as zig-zag ones that exploit the maximum bandwidth. Here we show the lacing technique by an example of embedding the perfect shuffle interconnection on a switch lattice.

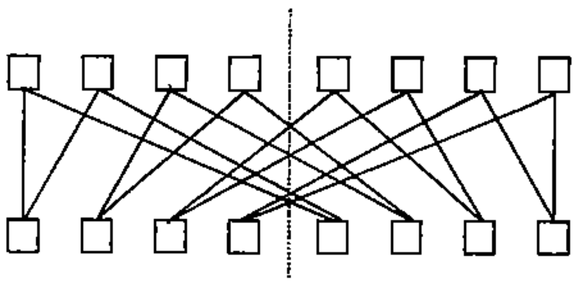


Figure C-1. The schematic perfect shuffle of n data items between two rows of $n/2$ processors, $n = 16$.

Figure C-1 shows the n perfect shuffle connections between two rows of $n/2$ processing elements for $n = 16$. Notice that there are $n/2$ connections passing through the dotted bisection line. To embed the interconnection on the CHiP computers, $\frac{n/2}{w*c}$ horizontal corridors are needed. If $n = 32$ then

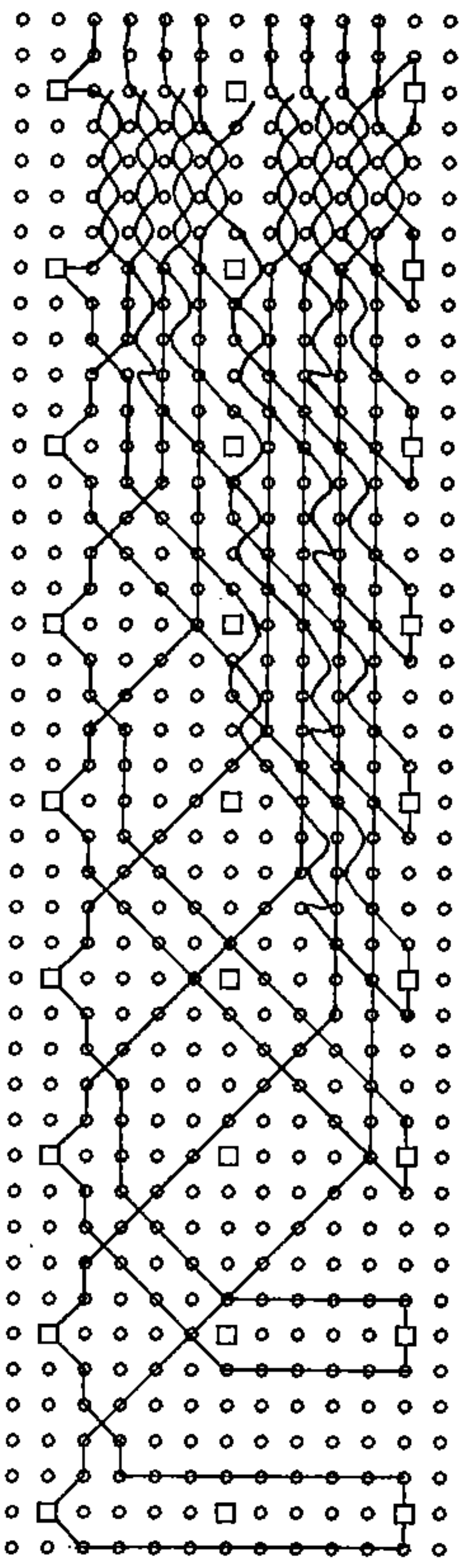


Figure C-2. An embedding of the perfect shuffle for n = 32 on a CHiP switch lattice two corridors are needed to host the interconnection on the switch lattice of $w = 4$, $c = 2$, and $d = 8$. Figure C-2 shows the embedding of the perfect shuffle interconnection for $n = 32$. Figure C-3 depicts some basic components which construct the embedding in Figure C-2.

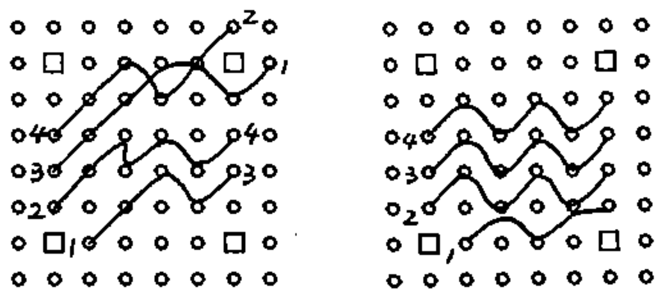


Figure C-3. Some basic components constructing the embedding in Figure C-2.

The four little pieces shown in Figure C-3(b) exploit the $n/2$ possible data paths passing through the bisection line in $\frac{n/2}{w*c}$ horizontal corridors. This lacing technique can be generalized to embed the perfect shuffle for larger values of n and on other switch lattices. The two data per processing element structure excludes the necessity of exchange edges as in the shuffle-exchange graph. The embedding in Figure C-2 can be extended to build multistage bitonic sorters as in [Batc68] or unistage bitonic sorters as in [Ston71] on the CHiP computers.

VITA

VITA

Ching-Chih Hsiao was born in Taiwan, the Republic of China on May 30, 1953. He graduated with honors from the National Chiao-Tung University in Taiwan with a Bachelor of Science in Computer and Control Engineering in 1975. During the subsequent two years he completed the obligatory military service in the Chinese Army Signal Corps as a second lieutenant in command of a signal platoon. In the fall of 1978, Mr. Hsiao enrolled as a graduate student at Purdue University. The University awarded him a Master of Science in Computer Science in 1980 and a Doctor of Philosophy in 1982. While at Purdue he served as a half-time programmer at CINDAS, a teaching assistant, and a research assistant in the Computer Science Department. Mr. Hsiao is a member of ACM and IEEE Computer Society.

He married Nien-Tsu Shen on June 8, 1980 and has a daughter Elaine Cathleen (Lan-Yin).