Chapter 3

Data Parallel Programming

Connection Machine systems are designed to operate on large amounts of data. These data sets may be richly interconnected or totally autonomous. A scientific simulation data set, such as a finite-element grid, is highly interconnected, with every node value connected to several element values and vice versa. Disparate values are continually being brought together, computed on, and redispersed. A document database, on the other hand, may be totally autonomous. The search of any one document proceeds entirely without reference to any of the others. There is no need to repeatedly combine information from multiple documents in a single computation.

The Connection Machine system is made up of large numbers of processors, each with its own local memory. From the programming perspective, it is possible to think of the memory in either of two ways. When computing on interconnected data sets, it is easiest to think of the memory as a single multigigabyte data space. When computing on autonomous data, it is easiest to think of it as many local memories.

Efficient Connection Machine algorithms invariably combine both points of view. When gathering data, one regards it as global. When computing on the gathered data, one thinks of it as local data, and of the computations themselves as being carried out in multiple local memories.

3.1 Data Sets and Distributed Memory

Data parallel programs can be expressed in terms of the same data structures used in serial programs. Emphasis is on the use of large, uniform data structures, such as arrays, whose elements can be processed all at once. A statement such as `A = B + C`, which in a serial language adds a single number `B` to a single num-

Copyright © 1993 Thinking Machines Corporation

ber **C** and stores the result in **A**, can equally well indicate thousands of simultaneous addition operations if **A**, **B**, and **C** are declared to be arrays.

In fact, the basic unit of data in a Connection Machine system is the array, or some other form of parallel variable. Arrays are spread across the distributed memory of the CM so that each element is in the memory of a separate processor. If the number of elements in the array matches the number of physical processors, then each local memory receives one element. If the number of elements in the array exceeds the number of physical processors, then several elements are placed in the memory of each processor. The elements remain distinct. Each is considered to have its own "virtual processor" and is handled accordingly.

The choice of parallel data structures is perhaps the most important aspect of data parallel programming. Once data has been properly allocated, executable code follows naturally. It is not necessary to use different operation names for different cases. Parallel code can look just like serial code, in the same way that floating-point arithmetic looks like integer arithmetic. A conventional compiler examines the declarations of variables **B** and **C** to determine whether the expression **B + C** requires an integer or floating-point add instruction. In the same way, a compiler for a data parallel language determines whether **B + C** requires a single addition operation or thousands.

Array Layout

A user program runs within a partition of a CM-5. Defined by the administrator, a partition may represent part or all of the CM-5 system. In order to allow a program compiled with a CM compiler to run on a partition of any size, the precise mapping of data elements to processors occurs at run time; the run-time system lays out the array for best efficiency. Compiler directives in each language allow programmers to request that the mapping be optimized for particular purposes.

Local Computation

Unless the programmer has specified otherwise, arrays of equal size and shape will have identical layouts. Thus, identical elements of each such array will share the memory of a particular processor. When a computational statement such as **C = A + B** is executed, each processor locates and stores the needed data in its own memory; no interprocessor data movement is required, and the operation proceeds very quickly.

November 1993
Copyright © 1993 Thinking Machines Corporation

3.2 Interconnected Data Structures

The inherent structure of most data sets links each data element to some, but not all, other elements. Often the linkages are to neighboring elements, in which case the structure is said to be localized.

A matrix, for example, is generally thought of as having row and column structure. Elements that share one subscript are used in a connected way. If the matrix is used as part of a finite-difference calculation, then the horizontal and vertical neighbors are continually brought together for computation. If a data structure is converted from the spatial domain to the frequency domain, then a butterfly pattern may be required during the course of a Fast Fourier Transform (FFT).

It is not possible to arrange interconnected data so that all the pieces of data will reside in the processors that need to use them, because the same piece of data may be used in more than one part of the computation, by more than one processor. Interprocessor communication is required. Computations on data structures have a definite rhythm: first data elements are brought together, then computations are performed. Once the data elements have been brought together, the computations are local. Even on very complex data structures, it is possible to have most of the interacting elements located in the same processor memory. Typically, only a few need to be brought in from another processor's memory.

Establishing Linkages among Data Elements

Data parallel languages use pointers or array subscripts to establish connections between processors and hence between their data elements. If the required patterns are regular and local, such as processors sharing data with their nearest neighbors, then each processor can easily calculate the address of its neighbors as needed. For irregular arrays, an array of pointers, itself a parallel data structure, establishes an arbitrary pattern of intercommunication.

3.3 Interprocessor Communications

There are four important categories of interprocessor communications:

- replication
- reduction

Copyright © 1993 Thinking Machines Corporation

- permutation
- parallel prefix

Each of these four types of data transfer can be applied to regular or irregular data sets: to vectors, matrices, multidimensional arrays, variable-length vectors, linked lists, and completely irregular patterns. All these combinations are supported by data parallel software within the CM-5. In addition, the most common or otherwise important cases are supported directly by special hardware built into the Control Network. In all cases, the CM-5's high performance is a result of having all the processors act cooperatively to achieve the needed data transfers.

Replication

Replication consists of taking some data values and making a larger number of data values by copying them. (See Figure 7.) A single value, for example, may be *broadcast* to all processors for use in a computation. A vector may be copied into each column of a matrix, or into each row. (The general case of making many copies of an array to fill a higher-dimensional array is called *spreading*.) A less regular pattern is the division of a collection into arbitrary subsets of varying size, and one may wish to broadcast a different value within each subset. If the subsets are ordered and not interleaved, one may regard them as a collection of vectors of various sizes; this common case can be implemented more efficiently than the general case.

Most data parallel programming languages support broadcasting implicitly; if **`A`** and **`B`** are arrays and **`X`** is a scalar quantity, the statement **`A = B + X`** implicitly broadcasts **`X`** to all processors so that the value of **`X`** can be added to every element of **`B`**. The general case of replication is typically supported through parallel array indexing, that is, indexing the same array with many index values. If some of the index values are the same, then the same array element will be copied to many places. Intrinsic functions (such as **`SPREAD`** in Fortran) cover important special cases.

November 1993
Copyright © 1993 Thinking Machines Corporation

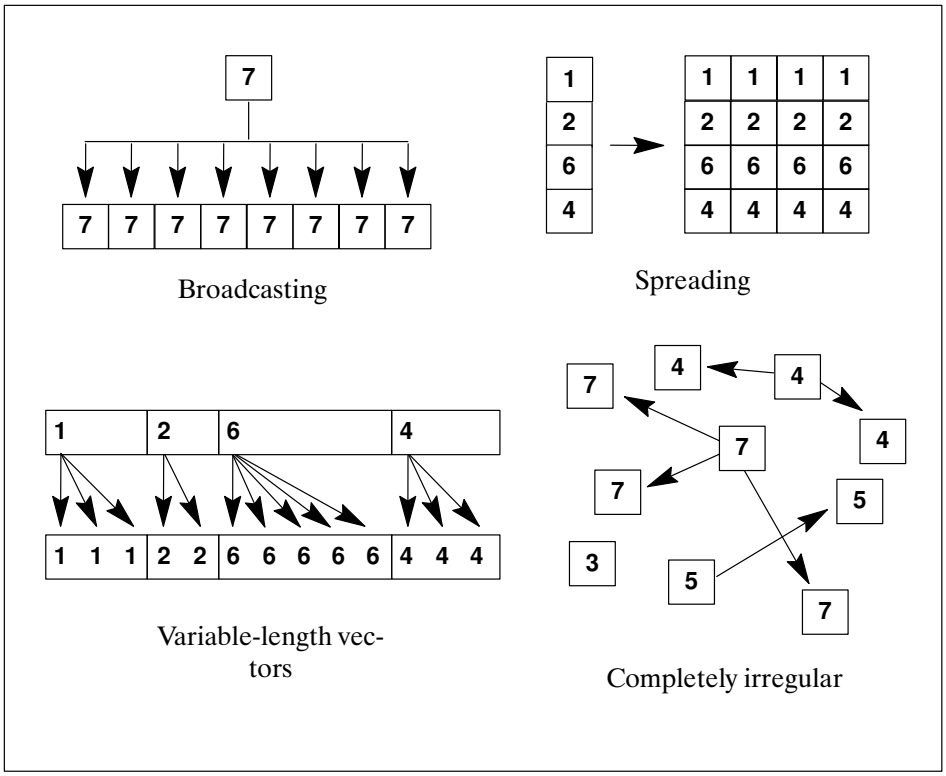

Figure 7. Replication.

Reduction

Reduction is the opposite of replication: Reduction consists of taking some data values and making a smaller number of data values by combining them. (See Figure 8. Note that it is similar to Figure 7 except that the arrows all point the other way.) A single value, for example, may be produced by computing the *sum* of a set of values; here the combining operation is addition. Other important reduction operations include taking largest or smallest value (maximum or minimum), logical AND (are all results true?), and logical OR (is any result true?). All these start with a large collection of values and reduce them to a single result.

More complex patterns of reduction mirror related patterns of replication. The rows of a matrix may be summed to produce the elements of a column-vector result; this is the opposite of a **SPREAD** operation. A collection of variable-length vectors may be reduced, producing a separate sum for each vector. Completely general patterns may be specified by index values or pointers.

Copyright © 1993 Thinking Machines Corporation

Most data parallel languages provide a collection of operators or intrinsic functions for expressing various patterns of reduction. For example, the Fortran statement **X = SUM(A)** sums all the elements of the array **A** and places the scalar result in **X**. The same computation can be expressed in C* as

```
x = (+= a);
```

and if the old value of **x** is to be included in the sum one may simply write

```
x += a;
```

(which says that every element of **a** is to be added into **x**).

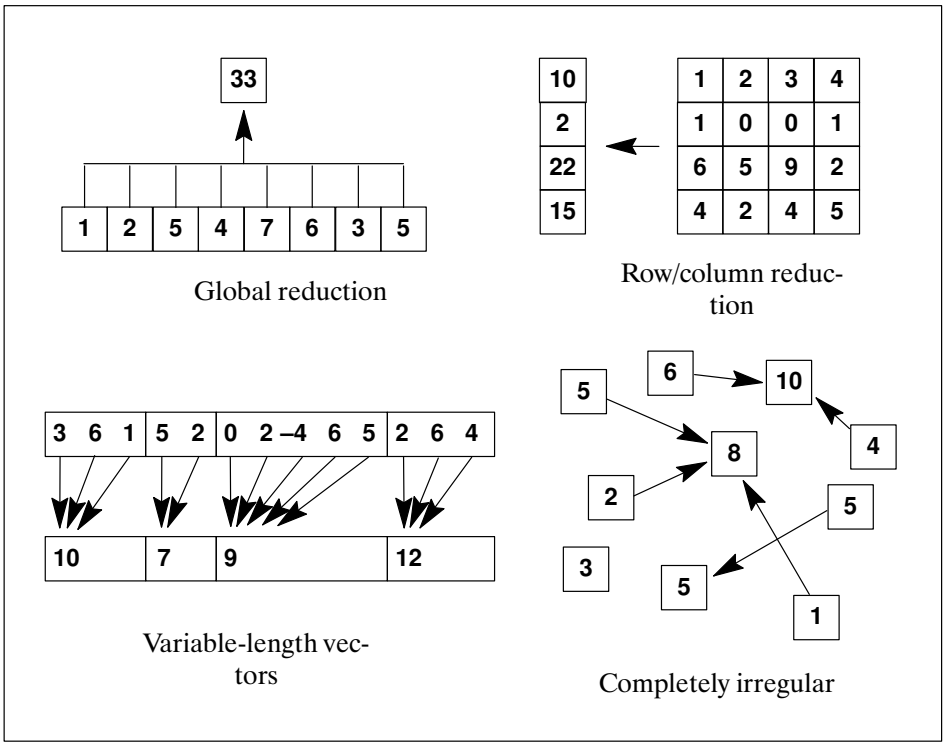

Figure 8. Reduction.

Permutation

Permutation rearranges its inputs to produce the same number of results; every data value comes from one place and goes to one place. (See Figure 9.) Trans-

November 1993
Copyright © 1993 Thinking Machines Corporation

posing a matrix, reversing a vector, shifting a multidimensional grid, and FFT butterfly patterns are all examples of permutation.

Data parallel languages usually express permutation through parallel array indexing and special-purpose intrinsic functions. A typical example of use might be a finite-difference grid used in the discretization of Laplace's Equation, in which the average of four nearest neighbors is iteratively computed:

```
 C = 0.25 * ( CSHIFT(A,1,+1)
&           + CSHIFT(A,1,-1)
&           + CSHIFT(A,2,+1)
&           + CSHIFT(A,2,-1) )
```

Here **CSHIFT** is a Fortran intrinsic that shifts (or rotates) an array with periodic boundary conditions. Elements shifted off one edge are circularly shifted into the opposite edge; thus no elements are lost in this operation. In contrast, **EOSHIFT** performs an end-off shift that discards shifted-out elements and introduces a pad value, usually zero, into vacated positions; this operation is thus technically a hybrid of permutation (of array elements) and replication (of the pad value).

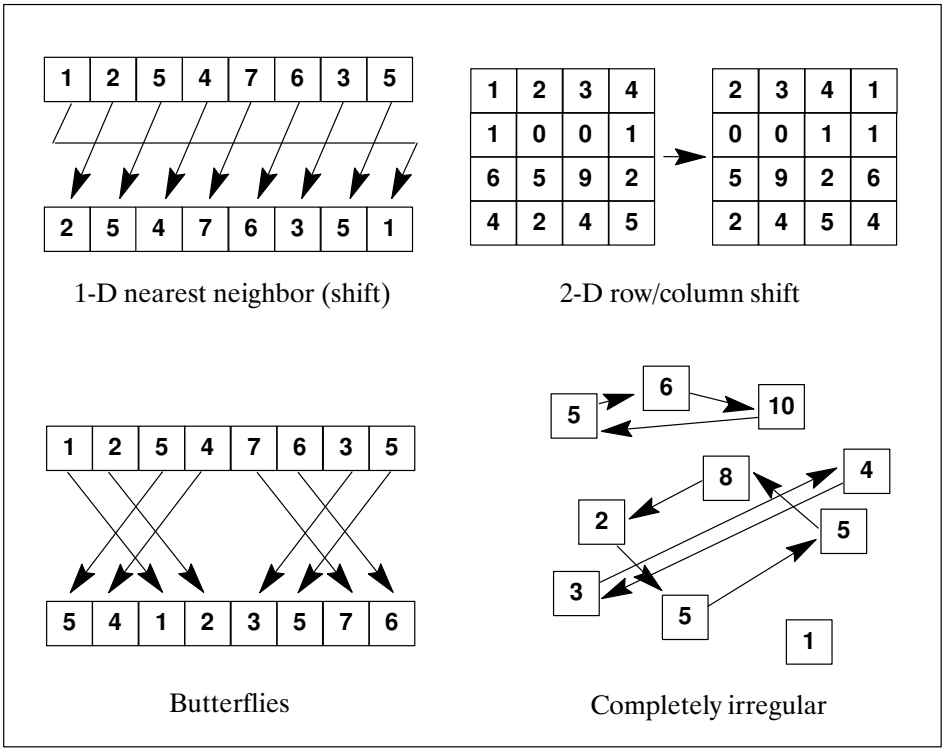

Figure 9. Permutation.

Copyright © 1993 Thinking Machines Corporation

Parallel Prefix

A parallel prefix operation is a very specific compound operation; it produces as many results as inputs, but each result may be a reduction of many inputs, and each input may contribute to many results. There happens to be a rapid and efficient parallel method for performing this complex compound operation; the CM-5 supports it with a combination of hardware and library software. It is of particular use in parallel computations because it permits rapid parallel execution of operations that at first glance appear to be inherently sequential.

The simplest example of a parallel prefix operation is computing the running totals of a list of numbers. The *k*th result is the sum of the first *k* inputs. (See Figure 10.) There is a simple sequential implementation of such a computation:

```
RUNNING_TOTAL = 0.0
DO J = 1,1000
  RUNNING_TOTAL = RUNNING_TOTAL + B(J)
  A(J) = RUNNING_TOTAL
END DO
```

This would appear to be an inherently sequential process, scanning the array **B** from one end to the other, but by bringing many processors to bear in parallel, one can perform this computation in 10 steps instead of 1000 steps (10 is approximately the base-2 logarithm of 1000).

November 1993
Copyright © 1993 Thinking Machines Corporation

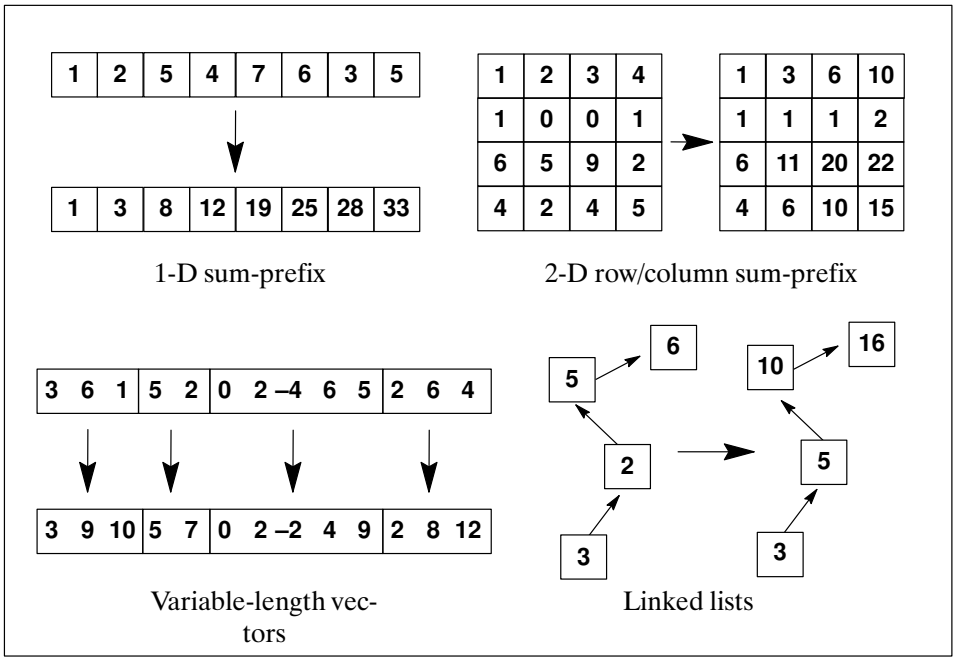

Figure 10. Parallel prefix.

3.4 Conditionals

Conditional operations are an essential part of data parallel programming, as of serial programming. Some of the control constructs (**`IF`**, **`CASE`**) are identical; others (**`WHERE`**, **`FORALL`**) are specific to parallel usage.

Data parallel programs implement conditionals by limiting the impact of operations to a certain subset of the data elements of a parallel data structure. A conditional operation first tests a specified condition in all elements of a parallel data structure. The specified operation is then performed only on elements for which the condition is true, while either an alternate operation, or no operation, is performed on the other elements. As in serial programs, conditionals may be nested in very general ways.

Copyright © 1993 Thinking Machines Corporation

3.5 In Summary

The data parallel model of computation makes it easy to program massively parallel computers. The model is also suitable for use on sequential computers, including vector processors, and on shared-memory parallel computers. High-level data parallel languages support the data parallel style. The CM-5 architecture is specifically designed for efficient execution of data parallel programs on large data sets.

Data parallel programming provides a practical framework for organizing interprocessor communication. An analogy may be drawn with the way "structured programming" has provided a practical framework for organizing control flow in sequential programs. Each model begins with primitive computations and uses a fixed set of standard combining forms to impose structure on the program.

Structured programming begins with simple assignment statements and observes that most patterns of control flow can be expressed in terms of sequencing (**BEGIN–END**), conditional branching (**IF–THEN–ELSE**), and looping (**WHILE–DO**). If these structures are conventionally used wherever appropriate, then use of a low-level construct such as a **GOTO** is a strong indication, and a useful one, that something unusual is going on; maintenance programmers should pay special attention, and language designers should ask whether the situation represents a class of problems that could be addressed more generally. Conventional syntax has evolved for certain frequently used compound patterns, such as **CASE** statements and **DO** loops.

Similarly, data parallel programming begins with local computations and observes that most patterns of interprocessor communication can be expressed in terms of replication, reduction, permutation, and parallel prefix. If these structures are conventionally used wherever appropriate, then use of a low-level construct such as explicit message-passing is a strong indication, and a useful one, that something unusual is going on; maintenance programmers should pay special attention, and language designers should ask whether the situation represents a class of problems that could be addressed more generally. Conventional syntax has evolved for certain frequently used compound patterns, such as shifting of regular grids, sorting, and fast transforms such as FFT.

As Figure 11 suggests, the data parallel model simplifies the programmer's job by providing for parallel programs the conventional structure and discipline that structured programming provides for sequential programs. Indeed, the data parallel model is the only programming methodology yet put forward that provides a coherent global organization for structuring programs that operate on thousands of processors.

November 1993
Copyright © 1993 Thinking Machines Corporation

	Structured Programming	Data Parallel Programming
Primitive Computations	assignment statements	local computations
Basic Patterns	BEGIN . . . END IF . . . THEN . . . ELSE . . . WHILE . . . DO . . .	replication reduction permutation parallel prefix
Common Compound Patterns	DO loops CASE statements REPEAT . . . UNTIL . . .	regular grids, stencils fetch-with-add sorting, fast transforms
Low-Level Mechanisms	GOTO	message passing

Figure 11. Structuring programs.

3.6 More Information to Come

This introduction barely begins to present the features and capabilities of the CM-5. The remainder of this book presents them in somewhat more detail (although still at a summary level). Part II discusses the software that supports applications programming on the CM-5. Part III discusses the various aspects of the system's architecture.

For information beyond this, you can turn to technical reports on Connection Machine programming, and to the CM-2–CM-200 and CM-5 documentation sets. Especially recommended for new users are the manuals *Getting Started in C** and *Getting Started in CM Fortran*.

Copyright © 1993 Thinking Machines Corporation