## Chapter 1

# Supercomputing and Parallelism

The Connection Machine system CM-5 provides high performance plus ease of use for large, complex, data-intensive applications. Its architecture is designed to scale to teraflops or teraops performance for terabyte-size problems. It features

- independent scalability of processing, communication, and I/O
- extremely high floating-point and integer execution rates
- high processor-memory bandwidth
- efficient execution of high-level languages
- multiple job execution, both timeshared and partitioned
- multi-user network access
- security between users
- flexible high-bandwidth I/O
- balanced scalar and parallel execution
- balanced I/O, processing, and memory
- high reliability and high availability

The CM-5 continues and extends support for the parallel programming model that has proved so successful in the CM-2 and CM-200. To achieve its goals, the CM-5 takes advantage of the latest developments in high-speed VLSI, new compiling technologies, RISC microprocessors, operating systems, and networking. It combines the best features of existing parallel architectures — including fine- and coarse-grained concurrence, MIMD and SIMD control, and fault tolerance — in a single, integrated, "universal" architecture.


*Copyright © 1993 Thinking Machines Corporation*

## 1.1 Parallelism

One of the most notable advances in computing technology over the past decade has been in the use of parallelism, or concurrent processing, in high-performance computing. Of the many types of parallelism, two are most frequently cited as important to modern programming:

- *control parallelism*, which allows two or more operations to be performed simultaneously. (Two well-known types of control parallelism are *pipelining,* in which different processors, or groups of processors, operate simultaneously on consecutive stages of a program, and *functional parallelism*, in which different functions are handled simultaneously by different parts of the computer. One part of the system, for example, may execute an I/O instruction while another does computation, or separate addition and multiplication units may operate concurrently. Functional parallelism frequently is handled in the hardware; programmers need take no special actions to invoke it.)

- *data parallelism*, in which more or less the same operation is performed on many data elements by many processors simultaneously.

While both control and data parallelism can be used to advantage, in practice the greatest rewards have come from data parallelism. There are two reasons for this.

First, data parallelism offers the highest potential for concurrency. Each type of parallelism is limited by the number of items that allow concurrent handling: the number of steps that can be pipelined before dependencies come into play, the number of different functions to be performed, the number of data items to be handled. Since in practice the last of these three limits is almost inevitably the highest (being frequently in the thousands, millions, or more), and since data parallelism exploits parallelism in proportion to the quantity of data involved, the largest performance gains can be achieved by this technique.

Second, data parallel code is easier to write, understand, and debug than control parallel code.

The reasons for this are straightforward. Data parallel languages (such as the Connection Machine system's CM Fortran, C*, and *Lisp) are nearly identical to standard serial programming languages. Each provides some method for defining parallel data structures: CM Fortran uses the Fortran 90 array features, while the other two languages add a new data type. Once the data sets (arrays, matrices, structures, etc.) are defined, a single sequence of instructions, as in serial code, causes operations to be performed concurrently either on the full data sets or on selected sections thereof. Very little new syntax is added: the power


November 1993
Copyright © 1993 Thinking Machines Corporation

of parallelism arises simply from extending the meaning of existing program syntax when applied to parallel data.

The flow of control in a data parallel language is also nearly identical to that of its serial counterpart. Since this control flow, rather than processor speed, determines the order of execution, race conditions and deadlock cannot develop. The programmer does not have to add extra code to ensure synchronization within a program; the compilers and other system software maintain synchronization automatically. Moreover, the order of events, being essentially identical to that in a serial program, is always known to the programmer, which eases debugging and program analysis considerably.

## 1.2 Parallel Programming

Prior to the CM-5, the most successful implementation of the data parallel programming model was the so-called SIMD (Single Instruction, Multiple Data) architecture. As implemented on the Connection Machine models CM-2 and CM-200, the SIMD architecture has shown itself to be extremely efficient and powerful. Arrays that are hundreds or thousands of elements in size are laid out across hundreds or thousands of processors, one element per processor, in a format whose logical structure matches that of the data set itself and the operations to be performed on it. (See Figure 1.) When there are more array elements than processors, the processors subdivide themselves into "virtual processors" and give each element its own virtual processor. Instructions are then executed upon each element simultaneously. For example, given three 400 x 400 arrays, **A**, **B**, and **C**, the statement **C = A + B** is a single statement — and is executed as such — in data parallel programming.

But "data parallel" and "SIMD" are not necessarily synonymous terms. Consider, for example, finite difference codes. Boundary elements in these codes usually require special treatment, which means conditional branching. In data parallel languages, such branching is frequently coded along the lines of

```
where (boundary_elements)
    do_a
elsewhere
    do_b
end where
```


Copyright © 1993 Thinking Machines Corporation

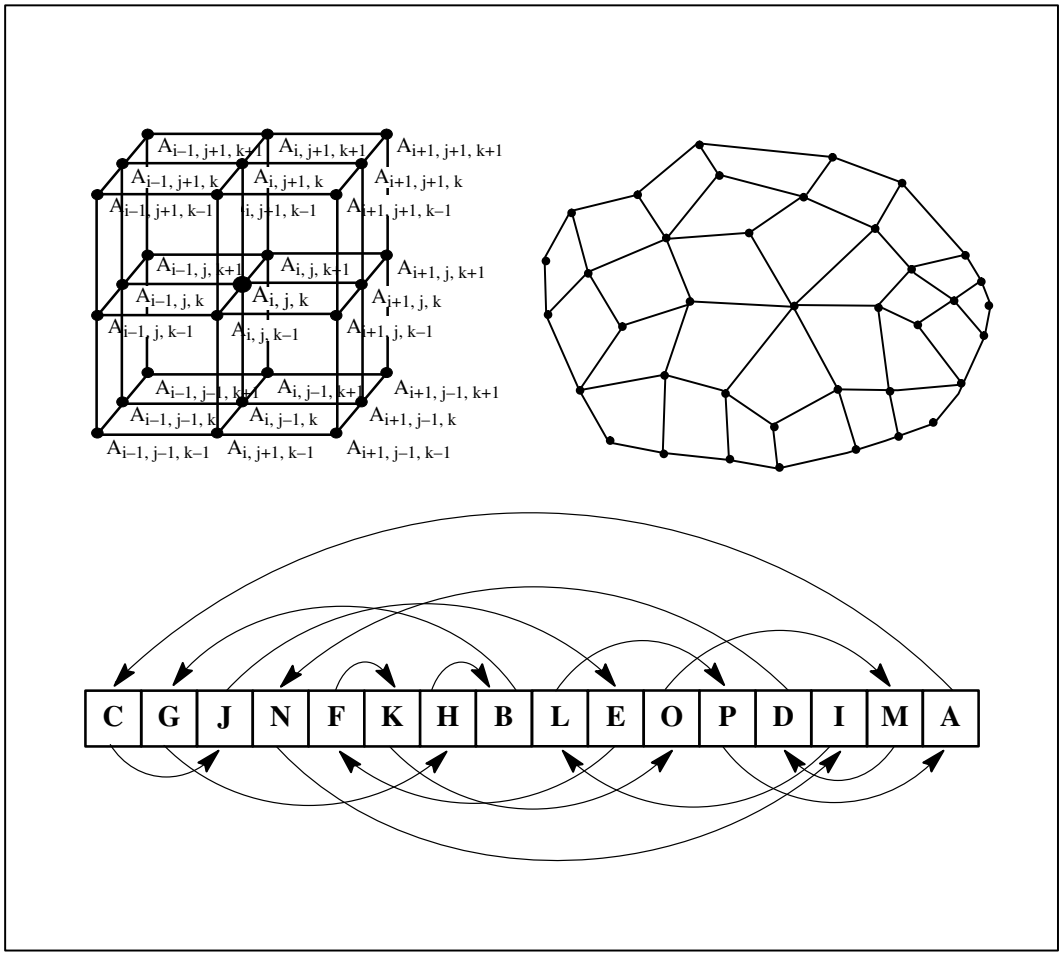


**Figure 1. Examples of data sets.**

**Some problems involve data sets organized as multidimensional grids. The calculation for each data point relies on the values of neighboring data pieces. The pattern of interaction is both local and regular. Finite difference methods are typical of this category.**

**Other problems, exemplified by finite element methods, operate on data that is less rigidly structured. The calculation for each data point again relies on the values of nearby data points, but the pattern of interaction is irregular. In some cases the pattern of interaction may change over time, as dictated by the content of the data (for example, to make the mesh finer in regions of interest).**

**For tasks such as sorting, the manner in which data points interact depends greatly on the data values; the pattern of communication will be both nonlocal and irregular.**

**The communications networks of the CM-5 are designed to support both regular and irregular patterns of communication. Patterns that are predominantly local are rewarded with higher throughput.**


*November 1993*
*Copyright © 1993 Thinking Machines Corporation*

A pure SIMD implementation of such code will execute the **where** branch for all boundary elements, and then execute the **elsewhere** branch for all interior elements. A MIMD (Multiple Instructions, Multiple Data) implementation will execute both branches simultaneously, with each processor making its own decision whether to fetch and execute instructions for the **where** branch or for the **elsewhere** branch for each element. When all processors have finished execution, the program will proceed to the next statement. Note that both implementations use the same code; both are undoubtedly data parallel programming. The only externally visible difference will be in performance; the second implementation, by using functional parallelism in support of data parallelism, can run faster than the first.

Note that the order of events in either case is identical to the order that would obtain for serial code. Even if the **where** branch takes several times as long as the **elsewhere** branch to execute, no processors will proceed to the subsequent statement until all have finished executing the **where** block. System software implements this control; the programmer does not have to worry about it. Only where events have no dependencies on each other, so that their order does not matter, will the order be unknown. (Figure 2 illustrates the combined independence and synchronization of program execution in this MIMD implementation of the data parallel programming model.)

## Extensions to the Data Parallel Model

Although data parallel programming provides the biggest gains among known techniques of parallelism, it may sometimes be usefully extended by mixing in other parallel techniques. For example, some applications may perform best when divided into sections, each section making use of data parallel programming and all sections together acting as a pipeline. Thus, one process might gather data and do some preliminary selecting or compacting; it would then pass its results to a second process, which would do more intense computing on the smaller data set; and that process would then pass its results to a third process, which would perform some visualization or reporting function. On the CM-5, all three processes can run in parallel, either timesharing on a single partition or perhaps each having exclusive use of a separate partition. In the latter case, each process has its own physical computing resources; I/O for the first process and computation for the second occur simultaneously, with no impact on each other or on the third process.


Copyright © 1993 Thinking Machines Corporation

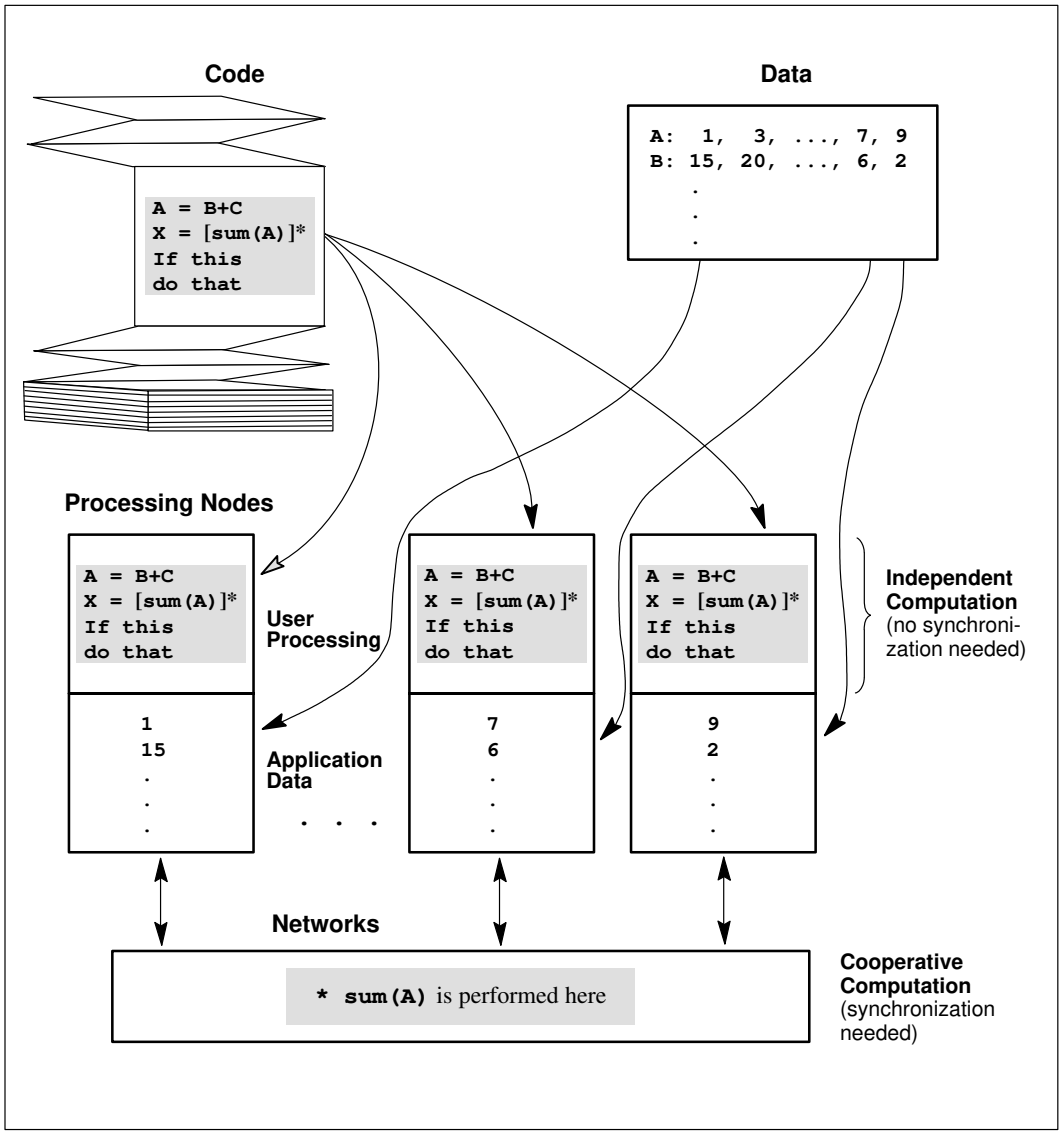


**Figure 2. Code running on a CM-5.**

**A partition manager loads identical code onto every processing node in a partition. Data is distributed across the nodes: Given an array of *m* values and a partition of *n* nodes, each node handles *m*/*n* values.**

**Each node executes its program independently, branching according to its own data values. As long as computation remains local, no synchronization or communication is needed.**

**When data needs to be transferred among processors — for example, when processors must each contribute values to a global sum — the communications networks carry the data and enforce the necessary synchronization. (For global combining operations such as `sum`, the Control Network performs the reduction.)**


November 1993
Copyright © 1993 Thinking Machines Corporation

The CM-5 thus extends the data parallel programming model developed for the CM-2 and CM-200 to incorporate an even broader and more widely useful mix of parallel techniques. Optimized for data parallelism, the CM-5 nonetheless supports other forms of parallelism that can either enhance data parallelism or allow the porting of programs from other architectures. This extended model, which we may call *coordinated parallelism*, represents the best that is known about parallel programming today.

## 1.3 Advantages of a Universal Architecture

In the past, programmers of supercomputers were forced to choose between MIMD machines, which were good at independent branching but bad at synchronization and communication, and SIMD machines, which were good at synchronization and communication but poor at branching. The CM-5 supports the full data parallel model by providing high performance for branching and synchronization alike — and, indeed, for all aspects of both SIMD-style and MIMD-style architectures.

### Scalable Computing

The CM-5 is the first architecture to offer truly scalable computing. It does this by combining its universal architecture with completely scalable hardware and scalable programming models. An application that runs on a small CM-5 can be run without change on a larger CM-5, and will see its performance increase accordingly. That same application may also be run on a workstation, mainframe, or shared-memory multiprocessor.

Figure 3 shows some of the ways in which applications originally written for other systems can run on the CM-5. (The illustration is based on the Fortran language, but the CM-5 supports C and Lisp as well.)

- Existing CM-2 and CM-200 Fortran programs can be moved directly onto the CM-5; recompiling is all that is needed.
- In some cases, partial recoding of CM-2 and CM-200 programs can bring better performance by taking advantage of new compiler features.


Copyright © 1993 Thinking Machines Corporation

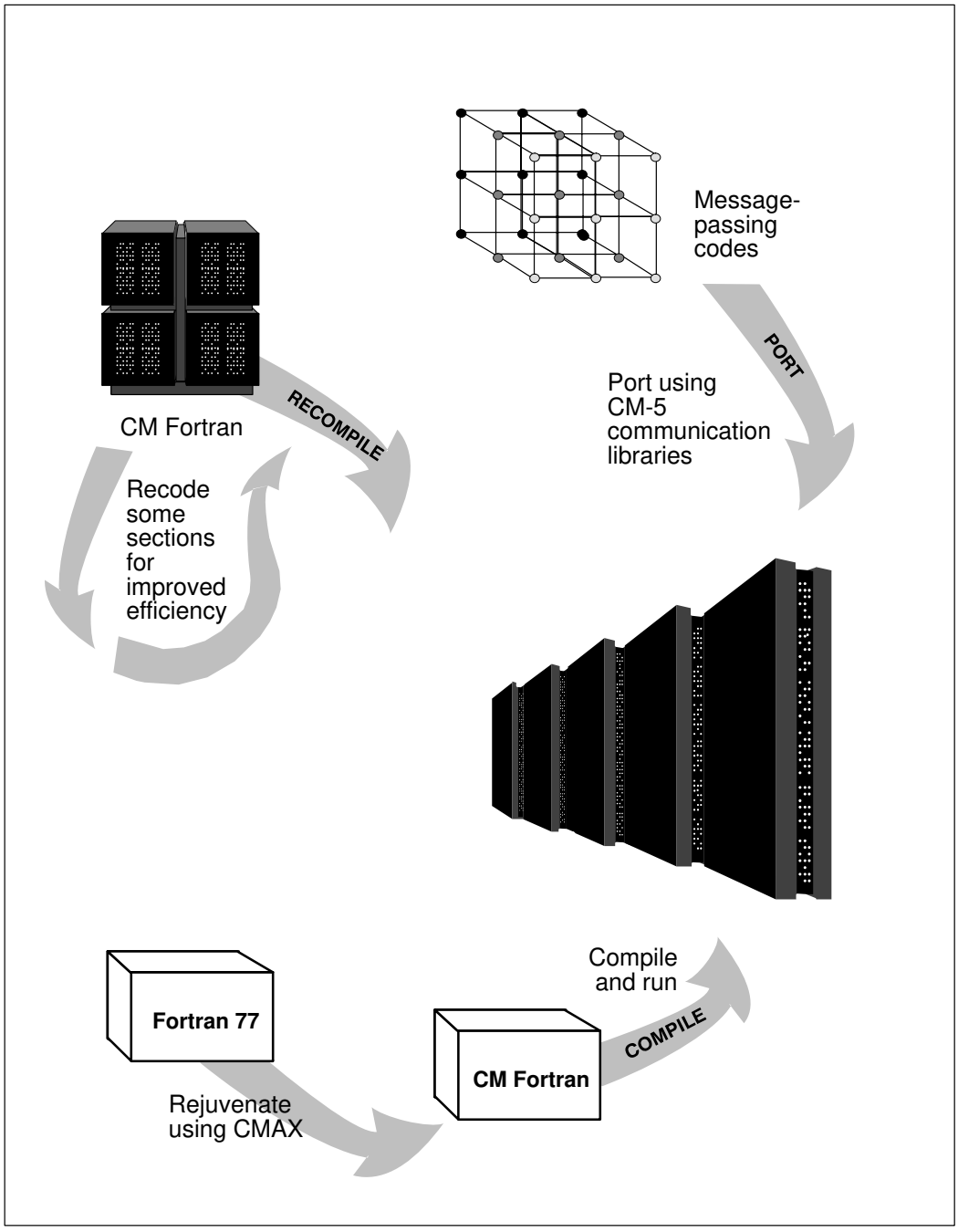


**Figure 3. Transporting programs to the CM-5.**

**CM Fortran programs written for the CM-2 and CM-200, and Fortran 77 programs written for execution on serial computers, and message-passing programs designed to run on MIMD-only architectures are all easily ported to the universal architecture of the CM-5.**




Copyright © 1993 Thinking Machines Corporation

- Applications written using a message-passing programming model for distributed memory computers can run on the CM-5 by substituting calls to a CM-5 message-passing library for the original calls.
- With some additional recoding, message-passing programs can be tuned to take advantage of the superior hardware facilities for cooperative computation offered by the CM-5.
- Existing Fortran 77 codes can be migrated to Fortran 90, using CMAX, and then compiled by the CM Fortran compiler. This allows many widely used codes to function effectively on the CM-5.

## 1.4 Looking Ahead

The next two chapters explain further what coordinated parallelism on the CM-5 offers. Chapter 2 shows how the CM-5 hardware is optimized to support coordinated parallelism, while Chapter 3 provides further explanation of the features to be found in data parallel languages.


Copyright © 1993 Thinking Machines Corporation