Chapter 7  HIGH PERFORMANCE COMPUTING

7.1  Vector Computers

These computers can be used for the majority of continuum-model problems, as well as for many particle-model problems.

Continuum model; a model of physical systems in which continuous quantities are modeled at discrete points and physical interactions are modeled as interactions among neighboring mesh points.

Particle model; a computational process in which physical behavior is modeled through the simulation of discrete particles acted upon by physical forces produced remotely.

The vector computer has emerge as the most important high-performance architecture for numerical problems. It has the two key qualities of efficiency and wide applicability.

7.2  Parallel Computers

There are many ways in which parallel computers can be constructed. These computers differ along various dimensions such as control mechanism, address-space organization, interconnection network, and granularity of processors.

Control mechanism; processing units in parallel computers either operate under the centralized control of a single control unit or work independently.

Address-space organization; Solving a problem on an ensemble of processors requires interaction among processors. The message-passing and shred-address-space architectures provide two different means of processor interaction.

Classification of computers

A single-instruction, multiple-data-stream machine (SIMD).
A multiple-instruction, multiple-data-stream machines (MIMDs).

What is Parallel Computing?

One way to solve a problem faster is to break the problem into pieces, and arrange for all the pieces to be solved simultaneously.
Fig. 7.1 Cost vs performance curve and its evolution over the decades.

The more pieces, the faster the job goes-up to a point where the pieces become too small to make the effort of breaking-up and distributing worth the bother.

A "parallel program" is a program that uses the breaking-up and handing-out approach to solve large or difficult problems.

Parallelism is one way to solve problems fast. Throughout human problem-solving history, complex engineering and organisational problems have been attacked and mastered by using parallelism.

Parallelism is a proven way to run programs fast. Everybody who has an interest in running his programs fast has an interest in parallelism whether he knows it or not.

Parallelism will become, in the not too distant future, an essential part of every programmer’s repertoire.

Parallelism is a proven way to run programs fast. Everybody who has an interest in running his programs fast has an interest in parallelism whether he knows it or not.

Parallel computers are classified by two qualifiers: the relationship of the memory to the processors and the number of instruction streams available to the system.

There are two general relationships between memory and processors: distribute memory and shared memory. In distributed-memory systems, each processor has its own private memory. Shared-memory systems have a single pool of memory to which all processors have access.
Single Instruction Systems

Multiple Instruction Systems

Fig. 7.2 MIMD vs SIMD operation.

Memory Schemes

Fig 7.3 Distributed-memory systems.
Communication

The other terms that describe distinguishing characteristics of parallel computers are MIMD (Multiple Instruction, Multiple Data) and SIMD (Single Instruction, Multiple Data), called mim-dee and sim-dee. MIMD systems allow processors to work on separate instruction streams, or tasks, at the same time, while the processors in SIMD systems all operate on a single instruction stream simultaneously.

Synchronous behaviour is automatic in SIMD machines; MIMD systems require that synchronisation be programmed in. A MIMD machine can simulate a SIMD machine, but not vice-versa. MIMD machines can make use of either shared or distributed memory, but only distributed memory is used in SIMD machines.

MIMD systems are made up of a set of nodes, each of which consists of a main processor, memory, and interface to the network. Each node has the computing power of a stand-alone workstation. Nodes process information independent of one another and communicate by sending and receiving messages. This independence gives these systems what is called a loosely-coupled architecture.

A loosely-coupled parallel system provides three major advantages:
It is flexible. Nodes with different types of processors can be used to adapt the system to specialised problems.

It is cost-effective. Parallel processing can significantly reduce the processing time for large computation-intensive problems. While some systems use specially designed components, others further reduce costs by using existing technology wherever possible.

It is easily scalable. Loosely-coupled systems are easy to expand, and you can also run several smaller problems concurrently.
A Programming Model for the Loosely-Coupled Parallel System

The process of producing a parallel program.
There are three major steps:

1. Decomposition—the division of the application into a set of parallel processes and data.
2. Mapping—the way processes and data are distributed among the nodes.
3. Tuning—alteration of the working application to improve performance.

Inherent in this process are certain principles that are guides to designing an efficient program. These principles help you determine each of the steps of the process:
Balance the computational load.
Minimize the communication to computation ratio.
Reduce sequential bottlenecks.
Make the program scalable (independent of the actual number of available nodes).

**Domain Decomposition: Examples and Techniques**

The main characteristic of many applications is regularity in the domain or data structures. Domain decomposition was developed to make problems of this kind easy to translate to a parallel system.

Second to perfectly parallel decomposition, domain decomposition is, in fact, the most straightforward of the decomposition techniques.

The general approach to the domain decomposition of an application is as follows:
1. Distribute the domain.
2. Restrict the computation so that each process updates its own data.
3. Put in the communication.

**Application of FEM**

The FEM is an active application area of massively parallel computing. FEM is a computational tool for deriving approximate numerical solutions to partial differential equations over a discretized domain.

The FEM is in itself a type of domain decomposition. Decomposition and mapping are the two steps that allow you to implement algorithms on parallel computers.

The basic idea is to break the problem up (decomposition) and assign the pieces to the nodes (mapping).

![Fig. 7.9 The region “R” subdivided into elements.](image)
Fig. 7.10 Sequential assembly algorithm.

Fig. 7.11 Mapping the elements to sixteen processors.

Fig. 7.12 The parallel assembly algorithm
Program 7.1 A serial sparse Gaussian elimination algorithm and the corresponding modify (modify_gauss) and divide (divide_gauss) operations.

Program 7.2 The modify (modify_chol) and divide (divide_chol) operations for use with a sparse row-oriented Cholesky factorization.
A Parallel Implementation of Sparse Gaussian Elimination

There are three levels of parallelism available in sparse factorization:

1. **Fine-grain** parallelism at the level of individual scalar floating-point operations.
2. **Medium-grain** parallelism at the level of performing floating-point operations over nonzero elements of entire rows or columns of the coefficient matrix (such as divide and modify operations).
3. **Coarse-grain** parallelism at the level of updating groups of rows or columns that can be solved independently of other such groups. If the factorization process is viewed as a collection of subtasks whose partial ordering is defined by an elimination tree, then coarse-grain parallelism refers to processing entire subtrees of the elimination tree.

```pseudocode
1. procedure PARALLEL_SPARSE_GAUSS (my_id, A[my_id,*])
2. begin
3. for i := 0 to my_id - 1 do
4. if i ∈ Smym_id then
5. begin
6. receive (Ai[*]) from the processor labeled i;
7. modify_gauss (my_id, Ai[*]);
8. endif; /*Line 4*/
9. divide_gauss (my_id);
10. for all j such that ((j > my_id ) and (Aj, my_id] ≠ 0)) do
11. send (A[my_id, *]) to the processor labeled j;
12. end PARALLEL_SPARSE_GAUSS

Program 7.3 A parallel sparse Gaussian elimination algorithm for the case in which each processor stores one row of the matrix of coefficients. Data is mapped such that the processor labeled my_id stores row number my_id. The algorithm assumes that Smym_id has been generated during symbolic factorization, and is available before numerical factorization starts.

Reference books