Data Parallel and MIMD Architectures

Concepts and Characteristics

Tópico 5

Data Parallelism – Where?

- The idea behind data parallel architectures is *conceptually* simple, but can be complex in realisation.
- Basically it is the execution of a single instruction operating on various data elements.
- Even under the von Neumann style, while the execution of instructions takes place *sequentially*, each of the instructions may operate on a number of different data elements *in parallel*.
- Data parallelism exists at various levels! (Just depends on the nature or size of the data entity.)

Example Parallel Data Sets

- **Matrices**, which occur in many mathematical problems, highlight one of the principle subtleties of data-parallel computation – the significance of the relationships between the elements.
- **Images** have extremely high degrees of parallelism. Image processing, machine vision and pattern recognition applications again highlight the significance of the spatial relationship between elements.
- The use of computers in the fields of accounting and management far outweigh their use in scientific computing. The largest data structures which occur here are **databases**.

Connectivity

Although highly parallel data sets share many features in common, because they arise in a variety of application fields, both their structure and the operations which are performed on them can differ significantly.

One of the most important aspects of the design of parallel machines is that relating to connectivity between processors.

While there is a wide range of ways parallel computers can be configured (i.e. a variety of interconnection patterns), some have been found to be more suitable than others depending on the application/data set.

Connectivity$_{(cont)}$

Connectivity arrangements arise primarily from data communication patterns of applications, e.g. near neighbour connectivity might be used to map spatially related data.

The design space for connectivity in parallel computers include:

- buses, crossbar, multistage and reconfigurable networks;
- trees (graphs), pyramids, hypercubes and near-neighbour (or mesh) networks.

Architectural Classes

Various network configurations are reflected in a number of alternative classes of data-parallel architectures:

- SIMD systems usually arrange their processing elements into a connectivity which physically matches the configuration of their target data sets.
- Systolic and pipelined machines take advantage of the repetitive nature of simple calculations on data.
- Vectorising computers utilise a similar method, but incorporate both enhanced programmability and parallelism within the data set.
- Associative and neural computers exploit parallelism within databases in executing parallel comparisons.

Principle Characteristics

Property	*SIMD*	*Systolic*	*Pipeline*	*Vector*	*Neural*	*Associative*
Programmability	Good	Fixed	Fixed	Good	Poor	Good
Availability	Good	Poor	Poor	Good	Poor	Poor
Scalability	Good	Fixed	Fixed	Fixed	Fixed	Good
Applicability	Wide	Narrow	Narrow	Wide	Narrow	Wide

There appears to be two groups: one which is programmable and therefore of wide applicability; and the other which consists of fixed function machines with defined performance and therefore of limited application.

MIMD Architectures

Thread and process-level parallel architectures are typically realised by MIMD computers. This class of parallel computers is the most general since it permits autonomous operations on a set of data by a set of processors without any architectural restrictions.

MIMD architectures represents a natural generalisation of the uniprocessor von Neumann machine, in which a single processor and memory is extended to architectures with multiple processors and memory modules.

But how are all these processors and memories connected?

Basic MIMD Structures

1. Replicate processor-memory pairs (called *nodes* or processing elements) and connect them via an interconnection network.

   - Nodes work more or less independently of one another, but send messages whenever interaction among nodes is necessary.

   - No node can access the memory of another node directly.

   - *Distributed Memory MIMD Architectures* or *multicomputers*.

2. Create a set of processors and a set of memory modules where any processor can directly access any memory via an interconnection network.

   - The set of memories represent the global address space which is shared among the processors.

   - *Shared Memory MIMD Architectures* or *multiprocessors*.

As with SIMD, one of the main design considerations is how to construct the interconnection network so as to reduce "message" traffic and data access latency.

Interconnection Networks

- Network topology is an important property which significantly influences the *latency* of the network.

- According to their topology, networks can be classified as *static* or *dynamic*.

- Multicomputers are typically based on static networks, while dynamic networks are typically employed in multiprocessors.

Distributed Memory MIMD Architectures

- Since processors work with their attached local memory most of the time, the *contention problem* is not so severe as in shared memory systems. Thus multicomputers are more scalable and good architectural candidates for building massively parallel computers.

- Processes cannot communicate though shared data structures and hence sophisticated synchronisation techniques are not required. Message passing solves not only communication but synchronisation as well.

Distributed Memory MIMD Architectures$_{(cont)}$

Most of the problems come from programming:

- In order to achieve high performance, attention has to be paid to *load balancing*.
- Message-passing based communication and synchronisation can lead to deadlock situations.
- Although there is no architectural bottleneck in multicomputers, message passing requires data structures to be physically copied between processes. Intensive data copying can result in significant performance degradation.

Shared Memory MIMD Architectures

Most of the advantages come from programming:

- There is no need to partition the code or the data, therefore uniprocessor programming techniques can easily be adapted.

- There is no need to phyically move data when two or more processes communicate. The consumer process can access the data from the same place as the producer stored it. As a result, communication between processes is very efficent.

Shared Memory MIMD Architectures$_{(cont)}$

Unfortunately, there are drawbacks too:

- Although programming shared memory systems is generally easier than for multicomputers, synchronised access to shared data structures requires special constructs such as semaphores, critical regions, monitors etc. The use of these constructs results in nondeterministic program behaviour which can lead to programming errors that are difficult to discover.
- But the main disadvantage is the lack of scalability due to the contention problem.

Shared Memory MIMD$_{(cont)}$

However, there are several ways to overcome the problem of low scalability:

- Use a high throughput, low latency interconnection network;
- Processors can be extended with local cache memories. However, this assumes that shared memory programs generally provide good locality of reference. (Unfortunately this is not always the case and also there is the *cache coherence problem.*)
- Logically shared memory can be physically implemented as a collection of local memories (i.e. *Distributed Shared Memory Architectures*).

Distributed Shared Memory Systems

Can be divided into three classes based on how the local memories are accessed:

1. Non-uniform memory access (NUMA) machines (e.g. Cray T3D);
2. Cache-coherent non-uniform memory access (CC-NUMA) machines (e.g. Stanford DASH and MIT Alewife); and
3. Cache-only memory access (COMA) machines (e.g. KSR-1).

NUMA Machines

- Each processor has a local memory which is also attached to the interconnection network.
- Local memory access is *much faster* than a remote memory access.
- Careful program and data distribution is required in order to exploit the potentially high performance of these machines.
- Have drawbacks similar to distributed memory machines.
- However, the main difference between them being programming style.

COMA and CC-NUMA Machines

These machines employ coherent caches in order to avoid the problems of NUMA machines. Together, the single address space and coherent caches significantly ease the problem of data partitioning and dynamic load balancing, proving better support for multiprogramming and parallelising compilers.

The machines differ in the extent to which the caches are applied. In COMA machines, where every local memory is a cache, data dynamically and continuously migrate to the local caches of those processor's where they are needed, according to the applied cache coherency scheme.

COMA and CC-NUMA Machines$_{(cont)}$

CC-NUMA machines represent a compromise between NUMA and and COMA machines. Like NUMA machines, the shared memory is contructed as a set of local memory blocks. However, in order to reduce network traffic each processor has a cache as well. Although inital data distribution is static as in NUMA machines, dynamically redistribution is achieved by the cache coherence protocols as in COMA machines

Problems of Scalable Computers

There are two fundamental problems to be solved in any scalable computer system:

1. Tolerate and hide the latency of remote loads.

2. Tolerate and hide idling due to synchronisation among parallel processes.

Remote loads are unavoidable in scalable parallel systems that use some form of distributed memory.

Problems of Scalable Computers$_{(cont)}$

Several hardware/software solutions have been proposed:

- Application of cache memory
- Prefetching
- Introduction of threads and fast context switching mechanisms.

Design Issues in Scalable Computers

The main design issues in scalable parallel computers:

- Processor design;
- Interconnection network design;
- Memory system design; and
- I/O system design.