A. V. Gerbessiotis
CIS 485/Spring 1999
Handout 1
Introduction to Parallel Computing

Parallel Computing

A *parallel algorithm* is an algorithm designed for a parallel computer.

0.1 What is a parallel computer?

A *parallel computer* is a collection of processors that cooperatively solve computationally intensive problems faster faster than other computers. Parallel algorithms allow the efficient programming of parallel computers. This way the waste of computational resources can be avoided.

The term supercomputer refers to a general-purpose computer that can solve computational intensive problems faster than traditional computers. A supercomputer may or may not be a parallel computer.

The cost of building a parallel computer has dropped significantly in the past years. One is able to build a parallel computer using off-the-shelf commercial components. This way workstations can be arranged together in a cluster/network and interconnected with a high-speed network (eg Myrinet siwtches, 100Mb/sec or Gigabit ethernet switches) and form a high-performance parallel machine.

The construction of a parallel computer is more complicated than that of a sequential machine. One needs to take into consideration not only those issues related with building a sequential machine but also issues related with the interconnection of multiple processors and their efficient coordination. A complete network where each processor can directly communicate with every other processors is not currently technologically feasible if the parallel machine consists of more than a few processors. Methods thus need to be devised that allow the efficient dispatch of information from one processor to another (routing protocols and algorithms). Problems like congestion (bottlenecks) may occur if someone is not very careful in designing such methods.

0.2 Course outline

This course deals with the design, analysis and implementation of parallel algorithms and study of their performance. An efficient sequential algorithm does not necessarily translate into an efficient parallel algorithm. Implementation of an inappropriate algorithm in parallel may waste resources (eg computational power, memory). A sequential algorithm needs perhaps to be split into pieces so that each piece is executed by a processor of the parallel machine. Each piece, however, may need to interact with the piece held at another processor (say because it wants to access a memory location held there). If a sequential algorithm utilizes M memory locations and solves a problem instance within time T, one expects a parallel algorithm solving the same problem instance to utilize as many resources as its sequential counterpart. If the algorithm is run on a p processor machine it is thus expected to utilize memory of overall size (over all p processors) that is comparable to M and the sum of execution time over all p processors to be around T. The best one can hope for is for the parallel algorithm to use M/p memory per processor and its parallel time to be T/p. In order to achieve such a running time (and it is not clear that this is possible, even if it is feasible) an algorithm designer/programmer needs to take into consideration the architecture of the parallel machine (eg how processors are interconnected). This optimal speedup of a parallel program is more the exception rather than the rule in practice. The reason is that *interprocessor communication* and *synchronization* are expensive operations and unavoidable in a parallel environment.

1 Users of parallel computers

One usually associates parallel computer users with the defense and aerospace industry where there is a constant need for computational power in order to solve weapon simulation problems, or to study the airflow around an aircraft (be it a fighter or a passegner jet). There are, however, many science problems that can only be solved and studied by using simulations. Such problems require extraordinary powerful computers and cannot be solved in reasonable amount of time today. A collection of problems that can be solved by a parallel computer are the following ones. Sometimes such problems are called *grand challenge problems*.

- Quantum chemistry, statistical mechanics, cosmology and astrophysics.
- Material sciences (eg superconductivity).
- Biology, biochemistry, gene sequencing.

- Medicine and human organ modeling (eg. human heart, to study the effects and dynamics of a heart attack).
- Environmental modeling and global weather prediction.
- Visualizaton (eg movie industry).
- Data Mining (find preferences of customers and then use more efficient direct marketing methods).
- Computational-Fluid Dynamics (CFD) for aircraft and automotive vehicle design.

For global weather prediction, the earth atmosphere is broken into cubic cells each one mile wide from the surface to an altitude of 10 miles. There are $5 \cdot 10^9$ cells overall each one requiring around 200 floating operations for each iteration of a weather prediction program. Each iteration gives the state of the atmosphere for a 10 minute interval and therefore approximately 1000 iterations are required for a 10 day forecast for a total computation count of 10^{15} flops. A commercial processor rated at 100 megaflops would require 100 days *to predict the weather 3 months ago!*

N-body simulation involves the calculation of forces and therefore of the position of N bodies in three-dimensional space. There are approximately 10^{11} stars in our galaxy. A single iteration of such a program would require several divisions and multiplications. On a uniprocessor machine it would take a year to complete a single iteration of the simulation. In order to predict whether a meteorite is going to hit earth in the distant future one would need to sacrifice precision over running time or the other way around.

In the 1980's a Cray supercomputer was 20-100 times faster than other computers (mainframes, minicomputers) in use at that time. Because of this, there was an immediate pay-off in investing on a supercomputer (a tenfold price increase was worth 20-100 times improvement in peformance). In the 1990's a "Cray"-like CPU is on the average twice - four times as fast as (and sometimes, slower than) a microprocessor. Paying 100 times more to buy a supercomputer does not make much sense. The solution to the need for computational power is a *massively parallel* computer, where tens to hundreds of commercial off-the-shelf processors are used to build a machine whose performance is much greater than that of a single processor.

Various questions arise when such a combining of processor power takes place.

- How does one combine processors efficiently?
- Do processors work independently?
- Do they cooperate? If they cooperate how do they interact with each other?
- How are the processors interconnected?
- How does one program such machines so that programs run efficiently and do not waste resources?

Answers to many of these questions will be provided in this course.

2 Parallel Computer Taxonomy

2.1 Flynn's Taxonomy of computer architectures

Depending on the execution and data streams computer architectures can be distinguished into the following groups.

(1) *SISD (Single Instruction Single Data)* : This is a sequential computer.

(2) *SIMD (Single Instruction Multiple Data)* : This is a parallel machine like the TM CM-200.

(3) *MISD (Multiple Instructions Single Data)* : Some consider a systolic array a member of this group.

(4) *MIMD (Multiple Instructions Multiple Data)* : All other parallel machines. A MIMD architecture can be an MPMD or an SPMD. In a Multiple Program Multiple Data organization, each processor executes its own program as opposed to a single program that is executed by all processors on a Single Program Multiple Data architecture.

2.2 Taxonomy based on memory distribution

(1) A *shared memory* machine has a single address space shared by all processors (UMA). Examples: SGI Power Challenge, SMP machines.

(2) In a *distributed memory machine* each processor has its own memory. Each processor can access its own memory faster than it can access the memory of a remote processor (NUMA). Examples: Cray T3D/T3E, IBM SP1/SP2.

(3) A *distributed shared memory* system is a hybrid between the two previous ones. A global address space is shared among the processors but is distributed among them. Example: SGI Origin 2000.

2.3 Taxonomy based on processor granularity

The granularity sometimes refers to the power of individual processors. Sometimes is also used to denote the degree of parallelism.

(1) A *coarse-grained* architecture consists of (usually few) powerful processors.

(2) a *fine-grained* architecture consists of (usually many inexpensive) processors (eg TM CM-200).

Process Granularity refers to the amount of computation assigned to a particular processor of a parallel machine for a given parallel program. It also refers, within a single program, to the amount of computation performed before communication is issued. If the amount of computation is small a process is *fine-grained*. Otherwise granularity is *coarse*.

2.4 Taxonomy based on processor synchronization

(1) In a *fully synchronous* system a global clock is used to synchronize all operations performed by the processors.

(2) An *asynchronous* system lacks any synchronization facilities. Processor synchronization needs to be explicit in a user's program.

(3) A *bulk-synchronous* system comes in between a fully synhronous and an asynchronous system. Synchronization of processors is required only at certain parts of the execution of a parallel program.

3 Performance Characteristics of a Parallel Algorithm

The performance of a parallel algorithm A that solves some problem instance can be measured in terms of various measures.

- *Parallel Time* T gives the execution time of the parallel algorithm and $T = max_i T_i$, where T_i is the running time of algorithm A on processor i.
- *Processor size* P is the number of processors assigned to solve a particular problem instance.
- *Work* is the product $P \cdot T$. Sometimes the actual work $\sum_{j=1}^{T} P_j \leq W$ is used instead, where P_j is the number of processors that are active in step j.
- *Speedup* given by $s = T_s/T$ is the ratio of the execution time T_s of the most efficient sequential algorithm that solves a particular problem instance to parallel time T of A on the same instance.
- *Scaled speedup* is the ratio T_1/T_p, where T_i here denotes the parallel time T of A when i processors are used to solve that particular problem instance.
- *Efficiency* $e = s/p$, which is sometimes expressed as a percentage. It is also the case that $e = T_s/W$, where $W = TP$.
- *Scaled Efficiency*, which is derived similarly from scaled speedup.

4 Contents of this course

The course can be divided into three parts.

4.1 Simple Parallel Algorithms on a shared memory system

Parallel computing is introduced in the context of the Parallel Random Access Machine (PRAM in short) model. A PRAM consists of a collection of processors that can synchronously access a global shared memory in unit time. The advantage of the PRAM is its simplicity in capturing parallelism and describing parallel programs. Its disadvantage is that it ignores synchronization and communication issues which are quite important in currently built parallel machines.

4.2 Algorithms on fixed connection networks.

Fixed connection networks are processor interconnection networks whose links are fixed in time. A variety of parallel algorithms that are network specific will be described in this part of the course.

4.3 Realistic parallel computer models

Hardware designers sometimes ignore theory thus creating parallel machines that are very hard to program and therefore, programmers need to know the intricate details of the architecture to be able to effectively use such machines. In the past, there were no parallel computing models that would abstract parallel hardware in such a way that would allow parallel programs to be written that are *portable* (work equally well on shared and distributed memory machines of various manufacturers), *scalable* and *efficient*. Algorithms that are both portable and scalable will be called *transportable*.

Network algorithms are architecture dependent and an algorithm that works well on a given network may not work well on another one. Many of the network and PRAM algorithms that will be examined assume that *unlimited parallelism* is available (eg in order to multiply two $n \times n$ matrices, a machine with $O(n^2)$ or $O(n^3)$processors would be utilized). In real life, large problem sizes (say, $n = 10000$) are solved on machines with few processors (up to 1024 in most cases).

Realistic parallel computer models such as the BSP and LogP are subsequently introduced. Design and analysis of parallel algorithms in an aritecture independent way that result in algorithms that are parallel efficient (transportable and efficient) and also practical (implementable in real parallel machines) is then undertaken.

5 Fixed Connection Networks

In this course we examine fixed processor interconnection networks, that is, networks whose links do not change (fail) during the course of a computation. One can view an interconnection network as a graph whose nodes correspond to processors and its edges to links connecting neighboring processors. The properties of these interconnection networks can be described in terms of a number of criteria.

(1) *Set of processor nodes* V. The cardinality of V is the number of processors p (also denoted by n).

(2) *Set of edges* E linking the processors. An edge $e = (u, v)$ is represented by a pair (u, v) of nodes. If the graph $G = (V, E)$ is directed, this means that there is a unidirectional link from processor u to v. If the graph is undirected, the link is bidirectional. In almost all networks that will be considered in this course communication links will be bidirectional. The exceptions will be clearly distinguished.

(3) The *degree* d_u of node u is the number of links containing u as an endpoint. If graph G is directed we distinguish between the out-degree of u (number of pairs $(u, v) \in E$, for any $v \in V$) and similarly, the in-degree of u. The *degree* d *of graph* G is the maximum of the degrees of its nodes i.e. $d = max_u d_u$.

(4) The *diameter* D of graph G is the maximum of the lengths of the shortest paths linking any two nodes of G. A shortest path between u and v is the path of minimal length linking u and v. We denote the length of this shortest path by d_{uv}. Then, $D = max_{u,v} d_{uv}$. The diameter of a graph G denotes the maximum delay (in terms of number of links traversed) that will be incurred when a packet is transmitted from one node to the other of the pair that contributes to D (i.e. from u to v or the other way around, if $D = d_{uv}$). Of course such a delay would hold if messages follow shortest paths (which is the case for most routing algorithms).

(5) *latency* is the total time to send a message including software overhead. Message latency is the time to send a zero-length message.

(6) *bandwidth* is the number of bits transmitted in unit time.

(7) *bisection width* is the number of links that need to be removed from G to split the nodes into two sets of about the same size (± 1).

Below we present the network characteristics for some of the most popular networks that we shall be examining in this course. We refer to the textbook for more detailed definitions of these networks.

1. Linear array

In a linear array with N processors, processors are ordered in a single dimension so that each processor is connected to its two neighbors (except for the first and last processor). For such a network $|V| = N$, $|E| = N-1$, $d = 2$, $D = N-1$, $bw = 1$ (bisection width).

2. 2d-array or 2d-mesh or mesh

The processors are ordered to form a 2-dimensional structure (square) so that each processor is connected to its four neighbor (north, south, east, west) except perhaps for the processors of the boundary. For $|V| = N$, we have a $\sqrt{N} \times \sqrt{N}$ mesh structure, with $|E| \leq 2N = O(N)$, $d = 4$, $D = 2\sqrt{N} - 2$, $bw = \sqrt{N}$.

3. 3d-mesh

A generalization of a 2d-mesh in three dimensions. *Exercise*: Find the characteristics of this network and its generalization in k dimensions ($k > 2$).

4. Complete Binary Tree (CBT) on $N = 2^n$ leaves

For a complete binary tree on N leaves, we define the level of a node to be its distance from the root. The root is of level 0 and the number of nodes of level i is 2^i. Then, $|V| = 2N - 1$, $|E| \leq 2N - 2 = O(N)$, $d = 3$, $D = 2 \lg N$, $bw = 1$

5. 2d-Mesh of Trees

The 2d-Mesh of Trees (2d-MOT) combines the advantages of 2d-meshes and binary trees. A 2d-mesh has large bisection width but large diameter ($\sqrt{N}$). On the other hand a binary tree on N leaves has small bisection width but small diameter. The 2d-MOT has small diameter and large bisection width.

An N^2-leaf 2d-MOT consists of N^2 nodes ordered as in a 2d-array $N \times N$ (but without the links). The N rows and N columns of the 2d-MOT form N row CBT and N column CBTs respectively. For such a network, $|V| = N^2 + 2N(N-1)$, $|E| = O(N^2)$, $d = 3$, $D = 4 \lg N$, $bw = N$ The 2d-MOT posseses an interesting decomposition property. If the $2N$ roots of the CBT's are removed we get 4 $N/2 \times N/2$ CBTs.

A 3d-MOT can be defined similarly.

6. Hypercube

The hypercube is the major representative of a class of networks that are called hypercubic networks. Other such networks is the butterfly, the shuffle-exchange graph, de-Bruijn graph, Cube-connected cycles etc.

Each vertex of an n-dimensional hypercube is represented by a binary string of length n. Therefore there are $|V| = 2^n = N$ vertices in such a hypercube. Two vertices are connected by an edge if their strings differ in *exactly one* bit position. Let $u = u_1 u_2 \ldots u_i \ldots u_n$. An edge is a dimension i edge if it links two nodes that differ in the i-th bit position. This way vertex u is connected to vertex $u^i = u_1 u_2 \ldots \bar{u}_i \ldots u_n$ with a dimension i edge. Therefore $|E| = N \lg N/2$ and $d = \lg N = n$. The hypercube is the first network examined so far that has degree that is not a constant but a very slowly growing function of N. The diameter of the hypercube is $D = \lg N$. A path from node u to node v can be determined by correcting the bits of u to agree with those of v starting from dimension 1 in a "left-to-right" fashion. The bisection width of the hypercube is $bw = N$. This is a result of the following property of the hypercube. If all edges of dimension i are removed from an n dimensional hypercube, we get two hypercubes each one of dimension $n - 1$.

7. Butterfly.

The set of vertices of a butterfly are represented by (w, i), where w is a binary string of length n and $0 \leq i \leq n$. Therefore $|V| = (n+1)2^n = (\lg N + 1)N$. Two vertices (w, i) and (w', i') are connected by an edge if $i' = i + 1$ and either (a) $w = w'$ or (b) w and w' differ in the i' bit. As a result $|E| = O(N \lg N)$, $d = 4$, $D = 2 \lg N = 2n$, and $bw = N$. Nodes with $i = j$ are called level-j nodes. If we remove the nodes of level 0 we get two butterflies of size $N/2$. If we collapse all levels of an n dimensional butterfly into one, we get a hypercube.

6 Parallel vs Sequential Computing

Theorem 6.1 (Amdahl's Law) *Let f, $0 \leq f \leq 1$, be the fraction of a computation that is inherently sequential. Then the maximum obtainable speedup S on p processors is*

$$S \leq \frac{1}{f + (1-f)/p}$$

Proof. Let T be the sequential running time for the named computation. fT is the time spent on the inherently sequential part of the program. On p processors the remaining computation, if fully paralelizable, would achieve a running time of at most $(1-f)T/p$. This way the running time of the parallel program on p processors is the sum of the execution time of the sequential and parallel components that is, $fT + (1-f)T/p$. The maximum allowable speedup is therefore

$$S \leq T/(fT + (1-f)T/p)$$

and the result is proven. ■

Amdahl used this observation to advocate the building of even more powerful sequential machines as one cannot gain much by using parallel machines. For example if $f = 10\%$, then $S \leq 10$ as $p \rightarrow \infty$. The underlying assumption in Amdahl's Law is that the sequential component of a program is a constant fraction of the whole program. In many instances as problem size increases the fraction of computation that is inherently sequential decreases with time. In many cases even a speedup of 10 is quite significant by itself.

In addition Amdahl's law is based on the concept that parallel computing always tries to minimize parallel time. In some cases a parallel computer is used to increase the problem size that can be solved in a fixed amount of time. For example in weather prediction this would increase the accuracy of say a three-day forecast or would allow a more accurate five-day forecast.

Theorem 6.2 (Gustafson's Law) *Let the execution time of a parallel algorithm consist of a sequential segment fT and a parallel segment $(1-f)T$ and the sequential segment is constant. The scaled speedup of the algorithm is then.*

$$S = \frac{fT + (1-f)Tp}{fT + (1-f)T} = f + (1-f)p$$

For $f = 0.05$, we get $S = 19.05$, whereas Amdahl's law gives an $S \leq 10.26$. Amdahl's law assumes that problem size is fixed, while Gustafson's law assumes that running time is fixed.

7 Terminology

Parallel Processing is the concurrent manipulation of data distributed on one or more processors solving a single computationally intensive problem.

A parallel computer is a multiple processor computer capable of parallel processing.

Supercomputer is a general-purpose computer capable of solving problems at high computational speeds and faster than traditional computers.

Process Granularity refers to the amount of computation assigned to a particular processor of a parallel program. It also refers, within a single program, to the amount of computation performed before communication is issued. If the amount of computation is small a process is *fine-grained*. Otherwise granularity is *coarse*.

Throughput is the number of results produced in one time unit (usually, one second).

UMA (Uniform Memory Access). It refers to that processor organization where the cost of accessing a memory location is the same for any processor. In a machine supporting UMA all processors work through a centralized switch to reach a shared memory. Examples of such machines are Encore Multimax, Sequent Symmetry and SGI Power Challenge.

NUMA (Non-Uniform Memory Access). It refers to that processor organization where the cost of accessing a memory location is not uniform. Parallel machines with multiple memory hierarchies fall into this category. Example of such machines is the BBN TC2000 butterfly supporting up to 128 processors, IBM SP1/SP2, SGI/Cray T3D/T3E and SGI Origin 2000.

Multiprocessor system. A collection of processors sharing a global shared memory. An SMP (Symmetric MultiProcessor) is an instance of a UMA multiprocessor system.

Multicomputer system. A collection of computers (each one consisting of at least one processor with its own memory). Examples of such machines are the Intel Paragon, Thinking Machines CM-5, Cray T3D/T3E, IBM SP1/SP2, etc.

Message Passing is the method of interprocessor communication where each processor sends/receives messages. A store-and-forward protocol (used in nCUBE/10, T800 transputers) requires that the message be copied into the memory of the receiving processor before it is dispatched away. In circuit-switched message passing (eg ATM, nCUBE 2) no intermediate processor (other than source and destination) stores the message. A communication "pipe" is opened between source and destination and then a communication is initiated.

One way one can increase the level of parallelism is to use the technique of *pipelining* where a computation is split into stages so that the output of one stage becomes the input of the following one. This way after some initial delay different data are processed at different stages/levels of the pipeline. An alternative to pipelining is *data parallelism* where multiple units operate on different pieces of data. Note that the level of parallelism for pipelining is fixed (number of stages) where as for data parallelism it is not fixed. In this course we deal a lot with data parallelism.

The term *scalability* refers to both algorithm scalability and architectural scalability. Algorithmic scalability means that the level of parallelism increases at least linearly with problem size. For architectural scalability, it means that the same performance per processor is maintained if number of processors increases.