COMP4510
Introduction to Parallel Computation

Models of Parallel Computation

Outline

- Introduction and overview
- Parallel Computer Architectures
- Models of Parallel Computation
- Data Dependence
- Techniques for Designing Parallel Algorithms
  - Partitioning, Communication, Agglomeration and Mapping, synchronization and load balancing

Types of Models

- There have been a huge range of models proposed for parallel computation
- These range from the abstract/theoretical…
  - Primarily used for reasoning about algorithm design and asymptotic analysis
- … to the concrete/applied models
  - Used as frameworks for parallel program development
- We will focus on the latter in this course

Abstract Models

- Since parallel machines are divided into those with and without shared memory, it is not surprising that there are theoretical models for each (as well as some others too)
- The primary model for shared memory algorithms is known as PRAM (Parallel Random Access)
- The primary model for distributed memory algorithms is known as the interconnection network (ICNet) model

Abstract Models (cont'd)

- The PRAM model assumes shared memory and ignores issues affecting scalability
- Several variants are used depending on assumptions made about concurrent memory accesses:
  - EREW-PRAM:
    - EREW stands for Exclusive Read, Exclusive Write
    - Only one process/thread is allowed to access a memory location at a time
    - This is the most restrictive (and realistic) model

Abstract Models (cont'd)

  - CREW-PRAM:
    - CREW stands for Concurrent Read, Exclusive Write
    - Only one process/thread is allowed to **write** to a memory location at a time
    - This is a less restrictive (and somewhat realistic) model
  - CRCW-PRAM:
    - CRCW stands for Concurrent Read, Concurrent Write
    - Many processes/threads are allowed to read or write a memory location concurrently
    - Very powerful but unrealistic
      - What is the result of concurrent writes?

Abstract Models (cont'd)

- For machines without shared memory, the abstract "IC Net model" can be used for parallel algorithm design
- This model explicitly links messaging over a network with a specific topology (ring, star, mesh, hypercube, etc.) to the algorithm
  - Can lead to unique and very efficient algorithms but they are typically non-portable
  - More later!

Abstract Models (cont'd)

- Other abstract models are largely beyond the scope of this course
- It is worthwhile to note that such models often try to address complaints about how unrealistic other theoretical models are
  - E.g. The BSP (Bulk Synchronous Parallel) model by Valiant and Culler et al's logP (latency, overhead, gap, Processors) model

Abstract Models (cont'd)

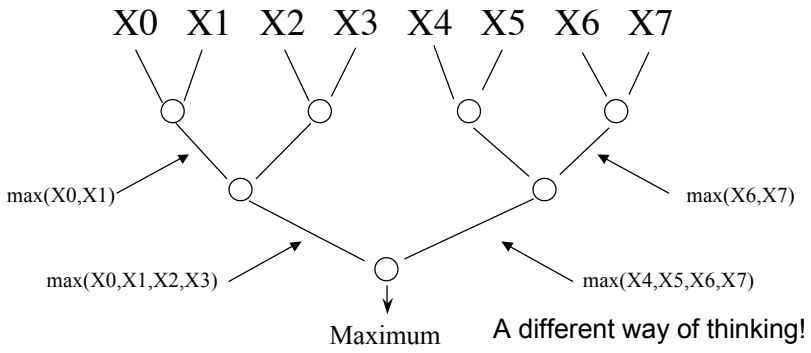
- One might wonder if abstract models are actually useful for much
- The development of parallel algorithms using abstract models for parallel machines is much simpler than addressing all the issues associated with programming a real parallel machine
- Such models are also useful to illustrate some fundamental techniques for solving problems in parallel

Parallel Algorithm Design

- Consider the simple problem of finding the maximum of a set of N numbers
- A straightforward technique for solving this problem in parallel is to use a technique called *binary fan-in*
  - Special case of *parallel prefix*
- The idea is to compare pairs of numbers in parallel on different processors and propagate each larger number into a new set

Parallel Algorithm Design (cont'd)

- This is repeated on the resulting set (of half the size) until a set of size 1 is produced

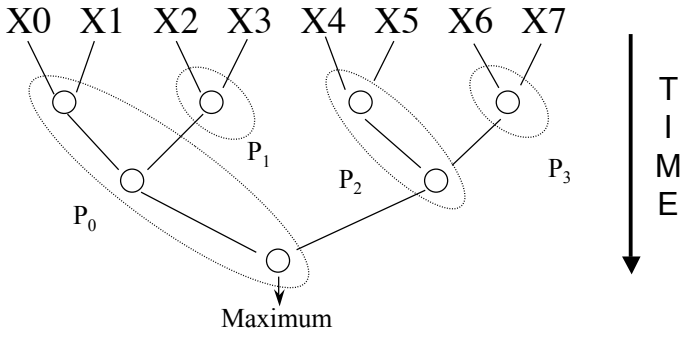


Parallel Algorithm Design (cont'd)

- This algorithm requires N/2 processors and takes time O(logN) (the depth of the tree)
  - compare this with the best serial algorithm which is O(N)
- Only N/2 processors are needed since in any execution cycle (at any level of the tree) at most N/2 pairs of numbers are compared
  - But *where* will each comparison be done???

Parallel Algorithm Design (cont'd)

- Consider the following mapping of comparisons in the binary fan-in algorithm to N/2 processors



Parallel Algorithm Design (cont'd)

- This algorithm can be expressed as follows:
  - For convenience we will use N processors
  - Assume that our X values are initially stored in *shared memory* locations `A[1]` through `A[N]`
  - Assume further that each processor has **local** variables $\texttt{big}_i$, $\texttt{temp}_i$, and $\texttt{incr}_i$.

```
1.FOR i:=1 TO n PARDO
      incr_i:=1
      A[n+i]:= MAXINT
      big_i := A[i]
```

Parallel Algorithm Design (cont'd)

2. REPEAT $\log_2 N$ TIMES
   FOR i:=1 TO N PARDO
   $temp_i$:=A[i+$incr_i$]
   big_i:=max(big_i,$temp_i$)
   A[i]:=big_i
   $incr_i$:=$incr_i$*2

- Upon completion, the result is in memory location A[i]
  - Example on the board - think about why!

Parallel Algorithm Design (cont'd)

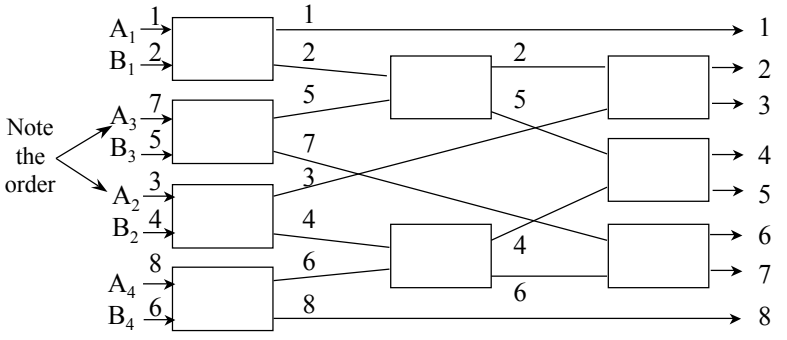
- This algorithm runs successfully without requiring concurrent reads or writes
  - Hence it is EREW-PRAM
- Question: How much better than O(logN) could we do if we added the power of concurrent reading and writing? (CRCW)
- Answer: Significantly better!
  - But that's for a different course
  - And its unrealistic!

Parallel Algorithm Design (cont'd)

- Let's consider the problem of merging two ordered sequences of numbers
- We will use a custom "network" which merges two sequences of sorted numbers
  - also implementable using shared memory
    - We can emulate the network connections in RAM
- This network is known as "Batcher's Odd-Even merge network"

Parallel Algorithm Design (cont'd)

- Consider two sequences: A={1,3,7,8} and B={2,4,5,6}



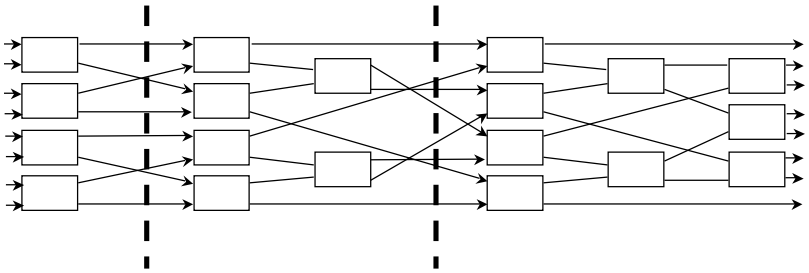
- Each box is a comparator giving the minimum at the top and the maximum at the bottom

A different way of thinking!

Parallel Algorithm Design (cont'd)

- Merging is the basis of sorting and Batcher's Odd-Even merge network can be extended to provide sorting ability
- To sort N elements, we begin by merging N/2 pairs of single elements, then we merge N/4 pairs of two sorted elements, etc. until we have a single sorted list of N elements
- We can use multiple instances of (parts of) Batcher's network to do this

Parallel Algorithm Design (cont'd)



- Of course, we can also convert this to run on a shared memory machine by using temporary memory locations (one for each vertical cross section of edges)

Concrete Models

- Flynn made one of the first attempts to classify parallel machines and his ideas are still useful as more "concrete" models today
- Flynn realized that parallelism could come from two sources: the code or the data
  - What we would now refer to as functional parallelism or data parallelism (more later)
- A parallel machine could support either or both forms of parallelism

Concrete Models (cont'd)

- Flynn's classification:
  - {SI, MI} x {SD, MD} gives:
  - SISD - Single Instruction, Single Data
    - Non-parallel machine
  - SIMD - Single Instruction, Multiple Data
  - MISD - Multiple Instruction, Single Data
    - Doesn't really fit any existing machine
  - MIMD - Multiple Instruction, Multiple Data
- We still refer to SIMD and MIMD models

Concrete Models (cont'd)

- MIMD is clearly the most powerful model
  - Can support the most parallelism
  - Unfortunately, it is also the most complex
    - 500 processors could run 500 different pieces of code in parallel - hard to understand and write
- SIMD has the virtue of simplicity
  - Every processor does the same thing
  - Good for some things (e.g. vector/array operations) but quite restrictive
    - What about conditional operations?

Concrete Models (cont'd)

- Probably the most commonly used parallel programming model is the Single Program Multiple Data (SPMD) model
  - Each processor executes the same program but is not synchronized at the instruction level
    - Thus, different processors can execute different code based on conditionals
    - More flexible and intuitive
    - Less time wasted waiting for other processors
    - But programmer must include synchronization code

Concrete Models (cont'd)

- There are a number of variants on the SPMD model including:
  - Master/Slave model
    - Where one process doles out work to the others and oversees their execution
      - Largely SPMD for the workers
  - Inspector/Executor model
    - If the work that must be done cannot be known a-priori then "inspector" code decides on partitioning and assigns work to the "executor" phase
      - Sort of dynamic Master/Slave

Concrete Models (cont'd)

- The SPMD and Master/Slave models vary a little based on the type of machine (SMP vs. cluster…) you will be using
- You will see and write programs using both models for shared memory…
  - Using OpenMP (and possibly pthreads)
- … and for distributed memory
  - Using MPI