CS411
Models of Parallel Computation

So far in class we've primarily been limiting ourselves to the RAM (random access) machine. In this machine there is one processor with M memory locations that can each be individually accessed in O(1) time.

Now let's consider the **PRAM**, or Parallel Random Access Machine. This is the most widely utilized parallel model of computation. When it was developed, the hope was that it would do for parallel computing what the RAM model did for sequential computing – become a platform upon which people could design theoretical algorithms that would behave as predicted by the asymptotic analysis on real computers.

Unfortunately, this has not quite been the case. The problem is that the PRAM ignores communication issues. This is good in the sense that it allows the user to focus on the potential parallelism available. However it is bad in the sense that it can often be difficult to map an actual physical machine to a particular PRAM algorithm. This is not the case for the RAM model, where it is easy to map a physical machine to a RAM algorithm.

The PRAM model has the following characteristics:

Processors: There are n processors, P_1, P_2, up to P_n where each is identical to a RAM processor.

Memory: There is a common, global memory available. If processors wish to communicate, they do so via common memory – there is no special communications channel between processors. Sometimes this is called a blackboard. It is typically assumed there are $m > n$ memory locations.

Memory access unit: The memory access unit of the PRAM is similar to the RAM in that it takes O(1) time to access any individual memory location for any processor.

The next question that should arise is how memory data access conflicts are resolved. We will probably have difficulties if multiple processors try to simultaneously write to the same memory location.

Two basic models exist for handling read conflicts:

1) Exclusive Read (ER). Only one processor is allowed to read from a given memory location during a cycle. That is, it is considered an illegal instruction if at any point in the execution, two or more processors attempt to read from the same memory location.

2) Concurrent Read (CR): Multiple processors are allowed to read from the same memory location during a clock cycle.

Write conflicts are more complex, and a variety of options exist:

1) Exclusive Write (EW): Only one processor is allowed to write to a given memory location during a clock cycle. It is considered an error if two or more processors attempt to write to the same memory location at once.

2) Concurrent Write (CW): Multiple processors are allowed to write to the same memory location during the same clock cycle. How should one resolve conflicts? A variety of schemes have been proposed;

    a. Priority CW : Some priority is given to each processor in advance (e.g. its processor ID number) and that with the highest priority succeeds. Note there is no feedback to the processors regarding success or failure of its write.
    b. Common CW: This model assumes that all processors attempting to write to the same memory location are writing the same value.
    c. Arbitrary CW: Some processor will succeed in writing, but it is arbitrary which is successful.
    d. Combining CW: When processors are writing, somehow the result of all of the writes is combined using some function (e.g.. SUM, AND, MIN, etc.)

Popular PRAM models are:

1. CREW (Concurrent Read Exclusive Write). This is one of the most popular because it maps to physical architectures well.

2. EREW (Exclusive Read Exclusive Write). This is the most restrictive model, but usually it is desired to allow concurrent reads.

3. CRCW (Concurrent Read Concurrent Write). When this is used, the details of the concurrent write must be specified.

Some Simple PRAM Algorithms

Perhaps the simplest algorithm is to broadcast a piece of information. While this sounds trivial, we need some semi-sophisticated methods to broadcast efficiently on some architectures. In the broadcast problem, some processor contains a piece of information in one of its registers or its cache and it wants to send this information to all other processors.

Let's start with a CR PRAM machine. A simple algorithm to broadcast is:

CR PRAM broadcast

1. Processor P[I] contains the data, *d*, to broadcast. It writes *d* from register R[I,J] to shared memory location X
2. In parallel, all processors read *d* from X

Step 1 requires O(1) time while step 2 also requires O(1) time because the model has the concurrent read property. This takes a total of O(1) time regardless of the number of processors.

Now consider the same problem on a ER PRAM. We need a new strategy since all processors can't read the same memory location.

ER PRAM Broadcast

1. Processor P[I] contains the data *d*, and writes it from register R[I,J] to memory location M[1]
2. For I = 1 to $\log_2 n$ do
   In parallel, processors P_j where $j \in \{1, 2, 4, \ldots. 2^{i-1}\}$ do
   Read d from M[J]
   If $j + 2^{i-1} <= n$ then write *d* to $M[j+2^{i-1}]$

This algorithm is an example of a *recursive doubling procedure*, in which during each step of the algorithm, the number of copies of the initial data item has doubled. Initially, there is one copy of the data at M[1]. This is read by processor 1. Processor 1 then writes the data to M[2]. In the next iteration, processor 1 and processor 2 read the data simultaneously. They then copy the data to M[3] and M[4] – the number of copies have been doubled. In the next iteration, processors 1-4 will copy the data for processors 5-8.

Since each step of reading and writing takes O(1) time, regardless of the number of processors, this algorithm with n processors can perform the broadcast in log time, O(lgn).

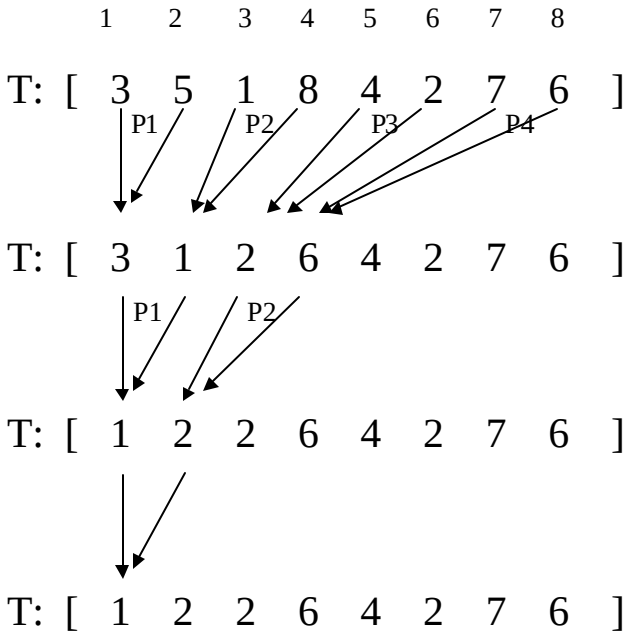
Next let's look at a PRAM algorithm to calculate the minimum of an array of values. The array is of size n, and we have at least n processors.

PRAM Min Algorithm

1. Copy array to a temporary array, T
2. For I = 1 to $\log_2$ n do
   In parallel, Processor P_j where $j \in \{1, 2, 4, \ldots. 2^{\log_2 n-i}\}$ do
   a. Read T[2j-1] and T[2j]
   b. Write min(T[2j-1],T[2j]) to T[j]
3. If desired, broadcast T[1] as the min

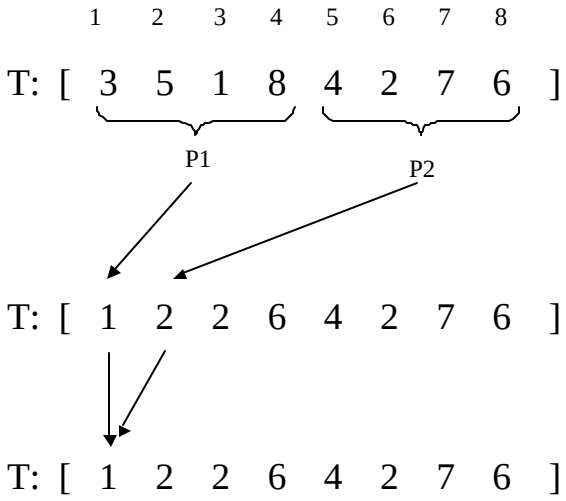
This process is more easily seen on an example. We end up with essentially a bottom-up tree, reducing the number of candidates in half each step of the way:

The final answer, the min, is stored in T[1].



Step 1 of the algorithm takes constant time since each processor can copy a unique element in time O(1). If you didn't care about preserving the input, then of course this step could be skipped. Step 2 requires O(lgn) time to perform the bottom-up tree operations. Each level of the tree is performed in parallel. The entire algorithm then takes O(lgn) time.

What if we didn't have n processors available for the previous problem? (Actually we could get by with n/2). For example, consider a case where the array is of size P where P > n. The most common solution is to break the problem up so that each processor works on a section of the array of size P/n and finds the min in its section. We have now reduced the problem to a size of n and can run the prior algorithm to find the min (using n/2 processors).



The runtime in this more arbitrary case is now limited by the RAM-based sequential search for the min and takes time O(P/n), where P is the size of the array and n is the number of processors. Note that since the number of processors is typically constant, this is just O(P) time – usually expressed as O(n) where n would be the size of the array (not to be confused with the number of processors!) So in terms of Big-O, there is really no asymptotic gain with the parallel algorithm version a RAM algorithm, unless we have enough processors.

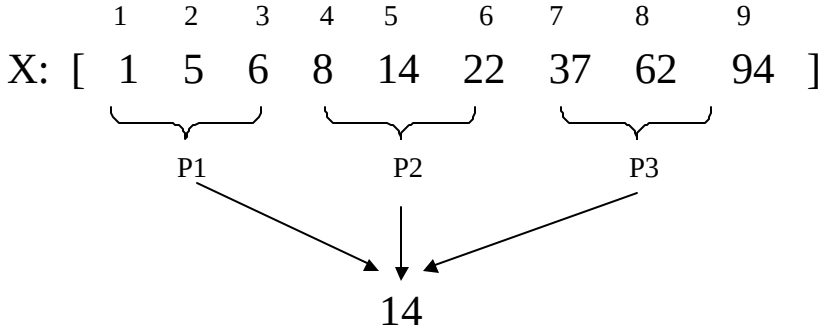
We have a similar situation if we follow the same strategy to create a PRAM to search an ordered array using binary search:

CRCW PRAM for an Ordered Array
N processors, combining operator of Minimum
Input: ordered array X, where X ranges from 1 .. m and a search value S

1. In parallel, every processor initializes success = infinity
2. In parallel, every processor I conducts a binary search on
   Lowindex = ((I-1)*m / n) + 1
   Highindex = (I*m / n)
   If binary search is successful, success is set to the index the value was found
3. In parallel, every processor saves success to memory location Z

Since we are combining using MIN, if the value was found, a non-infinity is stored in Z holding the index of the first occurrence of S in X. If the value was not found, infinity is stored in Z.

For example: Given m=9 and n=3
P1 gets index 1 to 3
P2 gets index 4 to 6
P3 gets index 7 to 9



For the runtime, each processor does binary search on a list of size m/n. The resulting runtime is then O(lg(m/n)). Once again, if n is a constant for the number of processors, we are running in time O(lgm) which is the same asymptotic runtime as a RAM machine.