In this lecture we review what has been covered in the course. We will attempt to emphasize some of the important points that might have been missed. We will also discuss how the various topics relate to each other i.e. there was some method to the madness.

The goals of the course were twofold. We tried to understand the basic issues in designing parallel algorithms, and also in designing the interconnection network between the computers.

# 1 Models and algorithms

We studied a number of parallel computer models on which to design algorithms.

1. Network models. In this there were various submodels, e.g. 2d arrays, Complete binary trees, General trees, Hypercubes, Hypercubic networks (Butterfly, deBruijn)

2. Shared memory/PRAM. Various varieties were discussed, e.g. EREW, CREW, CRCW of various write resolution rules.

We also studied the Semisystolic model and retiming. This is really not a model of computers, but a convenient device for designing algorithms.

We also discussed many algorithms. Each algorithm seemed to have a preferred model; though we sometimes also discussed how the algorithm could be executed in other models.

1. Prefix. We saw that this could be done in time $O(diameter)$ on all networks. However, a tree is what is really needed for this.

2. Dense matrix operations (multiplication, solving linear systems, all source shortest paths). We saw how these could be done on 2d arrays.

3. Normal algorithms. We saw how these could be done on hypercubes. We then saw how these could be done on hypercubic networks, and also on arrays.

4. Sorting. We studied this as a normal algorithm and also otherwise. Perhaps we did not study the best possible general algorithm. This was discussed in some of the exercises. The idea is: take a random sample and figure out how the keys are distributed. Then break the entire range into subranges, one for each processor. The sampling will help you ensure that the number of keys in each range is nearly the same. Send the keys in a subrange to the corresponding processor. Now do a local sort on each processor.

5. List ranking. We discussed this only for PRAMs. To run this on networks, we need to use simulation.

6. Packet routing. We considered specific problems such as permutation routing, Random destination routing.

   We saw that permutation routing or random destination routing can be completed in time O(diameter) on arrays, hypercubes, and hypercubic networks w.h.p. This result is important for purposes of simulation.

To argue about the goodness of our algorithms, we used 3 primary lower bound techniques.

1. Speedup bound. Time $\geq$ Best seq time/$P$, where $P$ is the number of processors.
2. Diameter bound. Time = $\Omega$(diameter of the network). This applies whenever some output depends upon information read in every processor.
3. Bisection bound. For some problems such as sorting, it is possible in the worst case that nearly all data has to cross the bisection of the graph. Thus we have time = $\Omega$("Data to be moved"/Bisection width).

# 2 Simulation/Graph Embedding

We also saw a number of graph embedding/simulation results. The motivation for these was to understand the relative power of different networks, and also to "automatically" port algorithms developed for one network onto another network. Some of these results are given below. In these, by "simulate well" we mean simulate with $O(1)$ slowdown.

1. N node arrays cannot simulate N node CBTrees well.
2. N node CBTrees can simulate N node 1d arrays, but not 2d arrays
3. Hypercubes can simulate arrays and trees well but not vice versa
4. Hypercubes can simulate CCC and Butterflies well but not vice versa.
5. N node Hypercubic networks can simulate N node arrays but not vice versa. The proof of this is a bit involved and we did not see it.
6. PRAMs can simulate any network well.
7. Arrays, Hypercubes, Hypercubic networks can simulate PRAMs with O(diameter) slowdown. Follows from packet routing results.
8. Any bounded degree network can be simulated on 2d array, hypercube, hypercubic network with O(diameter) slowdown. Follows from the packet routing results.

An important variation concerns simulating $N$ node guest networks on $P$ node host networks. Here are cases where the constant slowdown simulation is possible. We have considered some of these, and you should be able to prove all.

1. 2d array on 2d array.
2. CBT on CBT
3. Hypercube on hypercube.
4. Butterfly on butterfly.
5. 2d array on hypercubic networks. You should be able to prove this for the case $N = P \log^2 P$. Hint, simulate $N/P \times N/P$ squares on each processor of the host.
6. Network on PRAM

A key question in all this is: How should you break up the network into smaller pieces and give one or more piece to each host processor. The natural answer is locality preserving partitioning: the individual pieces should have as few outgoing edges as possible so as to minimize communication overhead. This means: break an array into small squares, butterfly into small butterflies, etc.

Note that once this principle is understood, simulating a network with many processors on a smaller network is much easier and more efficient than one in which both have the same number of processor.

The notion of graph embedding applies to certain kinds of *irregular* computations too. For example, (sparse) matrix vector multiplication, $Ax = b$, can be thought of as happening on a graph represented by the matrix: there is an edge from vertex $i$ to vertex $j$ if $a_{ij} \neq 0$. The vector element $x_i$ is associated with vertex $i$, and in multiplication it must be sent to those vertices $j$ s.t. $a_{ij} \neq 0$, where the product can be computed. Many computations are repeated matrix vector multiplications, and hence they would go fast if we can embed the graph nicely.

# 3 Thoughts on programming real computers

A real computer will usually have far fewer processors than the problem size. Processors in a real computer are connected to a shared memory or as a complete network. However, the complete network will often have low communication capability, i.e. getting one word of communication will take much longer than one computation.

If you are called upon to solve a problem you already know, e.g. matrix multiplication, here is what you should try.

1. Pick the model in which you had the best possible speedup for the problem when we studied it. For example, for dense matrix multiplication/inversion/... use mesh. For FFT: use butterfly. For sorting use sampling based algorithm, but you may already have a library function which does it well.
2. Simulate the chosen model/algorithm on your computer: use locality preserving partitioning. This way communication will be minimized on networks. On shared memory computers this will improve the cache hit ratio.

If you are called upon to design a parallel algorithm to solve a new problem, here is what you could do. First, understand the sequential algorithm well. Break it into steps. See how to parallelize individual steps. Individual steps may appear inherently sequential. See if something like prefix or list ranking will work – remember that prefix and list ranking appeared to be inherently sequential, and yet could be parallelized. If there is recursion, especially multiway recursion, the recursive calls may be inherently parallel.

## 3.1 Irregular parallelism

It may be, that your problem has parallel parts, however, the time taken for each part is difficult to predict. In this case, it is difficult to distribute the work among the available processors.

There are some natural solutions. You could maintain a central queue from which processors take away jobs. Alternatively, you have a queue for each processor. When a job is created, it is sent to a random queue. Hopefully, randomness will ensure each processor gets same amount of work.

A variation on this is work stealing: each processor asks a random processor for work when it has run out of work in its own queue.

Obviously, this is tricky in a distributed memory setting: sending work means sending data too.

## 3.2 Synchronization

Your program may have phases. The data generated in phase $i$ may be needed in phase $i+1$, and hence it may be necessary that a phase end completely before work in the next phase can begin. If a processor finishes its work for phase $i$, it must wait until all other processors also finish. This is called synchronization.

Synchronization can be achieved simply by keeping a shared counter, which all processors increment when they finish a phase. A processor increments the counter and waits for it to equal the number of processors. The processors must access the counter through some kind of mutual exclusion. The total time to update the counter will be proportional to the number of processors. Potentially during this period all processors could remain idle. This can introduce an inefficiency.

Alternatively, all processors could organize into a tree. The leaf processors send a signal to the parents when they finish. In general, a receipt of a signal means that the subtree beneath the sender has finished. Thus a node can send a signal to its parent when it finishes and it has received a signal from all its children.

We did not study synchronization first because it was not needed in network algorithms: the processors did not depend on other processors besides neighbours. On PRAMs we did not study synchronization because our PRAMs are synchronous. Synchronization is necessary with asynchronous shared memory (which is what you get in most real computers). The programming model will usually provide some built in mechanism for this, which will usually be adequate.

# 4 On the role of randomness

Randomness plays an even more important role in parallel computation, than sequential computation. This is because randomness, (coin tosses!) can be effectively used to resolve contention between processors. Randomness can also be used to create batches, as was the case in the levelled directed network routing algorithm. There have been other uses too, related to these basic ideas, in designing algorithms (that we did not study).

We did not use terribly complex statistical ideas. Our basic tool was: estimate the expected value of a random variable (say by expressing it as the sum of 0-1 variables). Then see if it takes that value w.h.p. If this is not the case, we can see how much larger does the bound needs to be (e.g. $logn$ times larger?) than the expectation, in order that it hold w.h.p.