CS-273 Foundations of Parallel and Network Computing
Professor Satish Rao — February 1, 2001
Scribe notes by Mark Pilloff — mdp1@uclink4.berkeley.edu

1 Load Balancing PRAM Simulations

Last time we considered simulating a v-processor PRAM on a p-processor Module PRAM and obtained the following bounds on the running time (all with high probability):

- $v/p = \Omega(\log p) \Rightarrow O(v/p)$–time simulation (constant factor efficiency)
- $v/p = \omega(\log p) \Rightarrow v/p + o(v/p)$–time simulation ($1-\epsilon$ factor efficiency)
- $v/p = 1 \Rightarrow O(\log p/\log\log p)$–time simulation

The first of these was proved in the last lecture– for concreteness we now prove the second item. First we need the following lemma which we state without proof:

Lemma 1 *(Chernoff/Hoeffding) Given n independent $0-1$ random variables $x_1, \ldots, x_n$ each with mean μ,*

$$Pr[\sum_i x_i > (\mu + \alpha)n] \leq \exp(-\alpha^2 n/3\mu).$$

We now use this to prove the following:

Theorem 1 *For a sequence of memory requests $M = \{m_0, \ldots, m_{v-1}\}$ assigned via hash function h to p memory modules uniformly at random where $v/p = \omega(\log p)$, the number of memory requests assigned to module b is*

$$C_b(M) = v/p + o(v/p)$$

with probability at least $1 - 1/p^c$.

Proof: Define random variables $x_1, \ldots, x_v$ where $x_i = 1$ iff $h(m_i) = b$. Then $C_b(M) = \sum_i x_i$ and each x_i has mean $\mu = Pr[x_i = 1] = 1/p$. By lemma 1 it follows that

$$Pr[\sum x_i > (1/p + \alpha)v] \leq \exp(-\alpha^2 vp/3).$$

Choose $\alpha = \sqrt{3c \log p/vp}$ so that the righthand side of the previous inequality becomes $1/p^c$. It then follows that

$$Pr[C_b(M) < v/p + \alpha v] \geq 1 - \frac{1}{p^c}.$$

All that remains is to show that $\alpha v = o(v/p)$ but this follows since

$$\alpha v = \frac{v}{p}\sqrt{3c\frac{\log p}{v/p}} = \frac{v}{p}\ o(1).$$

The last equality is a consequence of $v/p = \omega(\log p)$. Finally we note that $o(v/p) \equiv v/p\ o(1)$, thereby completing the proof. ■

Note that this applies only to a single memory module but when extended to all p banks, the probability of failure increases by at most a factor of p. Furthermore, this applies only to a single time step and we must bound the total running time by a polynomial in p if we want to ensure that it holds for all time. In general though, we will only be concerned with average load, so this may not be important.

Can we do better than the initially stated bounds by using a more clever simulation? Yes! We use a variation on the multiple hash function technique seen previously. First note that we cannot simply use two hash functions and always read from or write to the least loaded of the two choices. Since the loads change as the program executes, a write to a relatively unloaded module will force us to read that data back from the same module, even though the module may be heavily loaded by the time the read takes place. The solution to this problem is to use *three* hash functions and always read from or write to the *two* least loaded modules. By always choosing two modules, the read and the write will always have at least one module in common. (It is also necessary to attach a time stamp to the data as it is written so that a subsequent read can determine the most recently written data.) We claim without proof that using such a scheme guarantees a maximum load of $O(\log \log p)$ with high probability for $v/p = 1$.

2 Vector Computers

Vector computers support component-wise operations on whole vectors of data. A typical implementation connects p processors to b memory modules through a medium cooled with liquid Nitrogen and provides very high bandwidth (relative to latency). Such machines have gather/scatter operations which allow for fast reading and writing of data which may be spread over large areas of memory. For an array M (representing memory), an index vector I, and a temporary vector R, gather(M, I, R, numItems) fills the vector R according to $R[i] = M[I[i]]$ and scatter(M, I, R, numItems) writes to M according to $M[I[i]] = R[i]$.

NESL by Blelloch used this architecture to obtain good performance in linear relaxation problems (which use sparse matrices). There, each node is given a random index. Since the bandwidth of the machine is so high, there is no need to put neighboring nodes on the same processor. This is in stark contrast to the example highlighted in lecture 1 where we used locality to save on bandwidth.

3 Packet Routing Networks

Definition 1 *A packet routing network is a directed graph $G = (V, E)$ with inputs $I \subseteq V$ and outputs $O \subseteq V$.*

Definition 2 *A packet routing problem is a set of packets $\{p_i\}$ where $p_i = (i_i, o_i) \in I \times O$.*

Definition 3 *A solution to a packet routing problem consists of paths for each packet p_i connecting i_i to o_i and a schedule such that any edge is used no more than once in each time step and packets use the edges in their paths in order.*

Definition 4 *Given a set of paths, the congestion of an edge is the number of paths that contain it, the congestion of the set of packets (denoted C) is the maximum congestion of all edges, and the maximum dilation (denoted D) is the maximum length of any path.*

We note that minimum routing time is bounded from below by $\max(C, D)$ and from above by $C \cdot D$. The latter bound follows from considering a greedy schedule where every packet traverses an edge as soon as possible and ties are broken arbitrarily. Then no packet has to wait at a vertex for more than C time steps before traversing the next edge of its path, and no packets has to traverse more than D edges total, whereupon the bound follows.

Now we will consider the specific case of a permutation routing on an N-input butterfly network. First consider the worst-case permutation. In particular, consider an edge at level i (recall that there are $n \equiv \log N$ levels of edges in an N-input butterfly network). Such an edge is reachable from 2^i input nodes and leads to 2^{n-i-1} output nodes. Since routings in a butterfly network are unique for a given input and output, this edge could be in the paths of $\min(2^i, 2^{n-i-1})$ packets. A bit stream reversal is an explicit realization of such a case. For an edge at level $n/2$ this expression is approximately $2^{n/2} = \sqrt{N}$. Hence the worst-case routing time, regardless of what routing solution is chosen, is at least $C = \sqrt{N}$ as well.

Now consider random mappings where each input i_i chooses a random output o_i. The pair (i_i, o_i) specifies a unique path in the butterfly network so we need only specify a schedule to obtain a full solution. We will choose a greedy, first-in first-out (FIFO) algorithm. Each packet will traverse the next edge in its path as soon as it can, with ties broken based upon order of arrival (and arbitrarily for packets arriving simultaneously).

Theorem 2 *The maximum congestion in the above problem using the greedy FIFO algorithm is $O(\log N/\log\log N)$ with high probability.*

Before proving this, we note the following lemma.

Lemma 2 *(Chernoff/Hoeffding) Let $x_0, \ldots, x_{n-1}$ be independent $0-1$ variables with mean $\sum_i x_i = \mu$. Then*

$$Pr[\sum_i x_i \geq m] \leq \left(\frac{\mu e}{m}\right)^m.$$

Lecture ended before the theorem was proved.