CS 273: Prof. Satish Rao Notes

Scribe David Garmire

1/22/2003

Administrative

- Satish Rao, OH: T 3:30-5, W 1-2
- Christos Papadimitriou, OH: M 1-2, Th 3:30-5

Agenda

- PRAM Algorithms
- Network Problems
- Load Balancing

PRAM

We define a simplified algorithmic model of memory (abstracts the underlying architectural model on which we employ our algorithmic models, any assumptions must be specified explicitly). In sequential land, we use a Random Access Machine (RAM) model (Figure 1) whereby a processor accesses memory (reads and writes) in constant $O(1)$ time. In parallel land, we add more processors, and assume that these processors execute each step of the algorithm at the same time (lock-step) – any synchronicity assumption on contended data must be made explicit (e.g. EREW (exclusive read, exclusive write)).

For example, let us consider `sum(list)` which outputs the sum of a linked list's data elements (see Figure 2). In RAM, the algorithm pseudo-code would be written as

```
IF list = nil
  RETURN 0
ELSE
  RETURN sum(list->next)+list->value
```

and in PRAM, the pseudo-code for a possible iterative algorithm at a given processor (assume it is initially assigned to work on node i in the list, and a node's `sum` (partial sum) is initialized to its `value` and a node's `partialNext` is initialized to its `next` pointer) is

```
IF i->partialNext != nil
  i->sum = i->sum+i->partialNext->sum
  i->partialNext = i->partialNext->partialNext.
```

So iteration t will hold the relationship (assume WLOG pointers allocated sequentially in memory)

$$\texttt{i->sum} = \sum_{j=i}^{i+2^t-1} \texttt{j->value}. \tag{1}$$

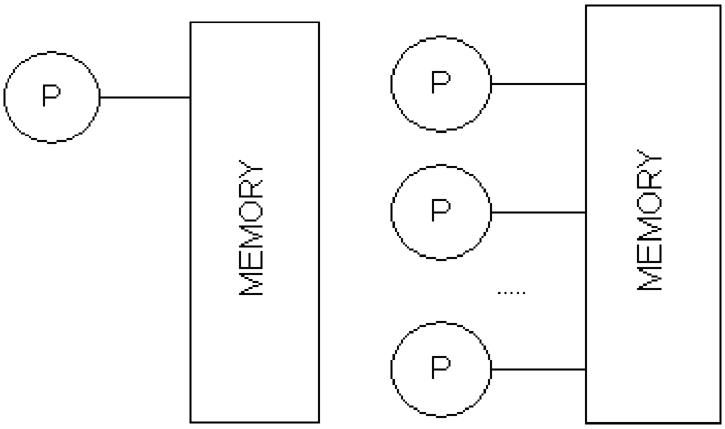

Figure 1: RAM and PRAM model

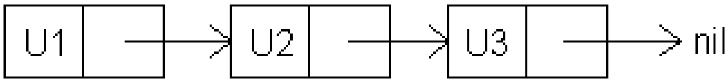

Figure 2: Linked List Representation

Assume there are as many processors p as nodes n. The total time for this program to finish (so the first node has the total sum) assuming the initialization was completed beforehand is $O(\log n)$. If there were fewer processors than nodes, we could have the code iterate for all of the nodes assigned to a given processor at the initial phase and then apply the iterative method for a total of $O(\frac{n}{p} + \log n)$ time. The total **work** that is done for an algorithm is actually the number of processors times the number of instructions each processor must execute, e.g., in the first consideration, the work is $O(p \log n) = O(n \log n)$.

Remark It is important to specify the memory contention model, so we know the reads are done before the writes terminate. Later, processor models will be discussed.

Network Problems

Let us consider the problem of each machine finding, in a distributed network (see Figure 3) of size n, the machine that has a given song out of s songs. On a sequential single system, we could have a hash table (see CLR [2] page 219) that assigns each song name (the key) to the location in memory that song may be found (the value).

```
Method 1:
  Insert: ‘‘Tell Everyone.’’
  Lookup: ‘‘Look at what people told me.’’
```

Method 1 has $O(n)$ messages for the insertion stage, as the machine that has the song broadcasts to everyone, and no message overhead for the lookup stage (already have the data local to each machine). This lookup efficiency incurs a substantial space cost of $O(s)$ per machine.

```
Method 2:
  Insert: ‘‘Do nothing.’’
  Lookup: ‘‘Ask everyone.’’
```

The insertion incurs no communication, but $O(n)$ messages are generated every time a machine would like to find a song. The nice property of this method is it is tolerant to a rapidly changing network.

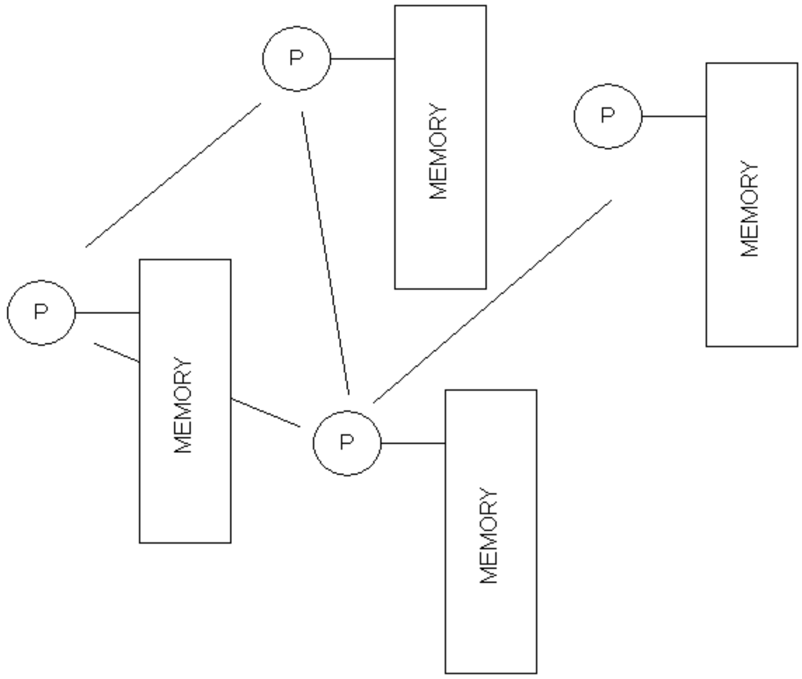

Figure 3: Distributed Network of Processors with Memory

```
Method 3:
  Insert: ''Tell hash(obj) machine you have it.''
  Lookup: ''Ask hash(obj) for the song.''
```

Both insertion and lookup now use a single machine as the point of reference for telling machines where a song is and finding any given song. It only requires at most $O(1)$ communication for insertion and lookup. The space required for that point of reference may still be $O(s)$ space on a single machine, but with an appropriate hash function, it will be on average around $O(\frac{s}{n})$ space per machine. A drawback of this algorithm is that the network must be static. To make the algorithm more fault tolerant, we might consider having a hash function that returns two possible points of reference? as well as, every-so-often flushing our data and readjusting the hash functions with new points of reference (need to synchronize somehow)?

For a study on measuring peer-to-peer systems including a look at resilience and availability of Gnutella and Napster, see [4]. The bibliography also contains links to other interesting papers on this specific subject.

Load Balancing

We would like to distribute load across servers. There are several methods to consider:

- Assign new jobs to least loaded server.
- Assign new jobs to a random server.
- Choose 2 servers at random and pick the best.

In the case of large networks, you cannot use the first method because it would take a long time to find the least loaded server (especially if several of the servers time out in the beginning when you have not found a very responsive server yet). The second and third methods are nice in that they take constant time, but how well do they load balance (i.e., what is their maximum load bounded by with high probability?)?

Consider we have N jobs and n servers, or abstractly N balls and n bins. The maximum load is the maximum number of balls that fall in a bin. The first method will always choose the least loaded server, so it will have a max load of $O(N/n)$, or if we assume $N = O(n)$, the max load is $O(1)$. The second method we will show has a max load of $O(\log n)$ with high probability (it actually has the better bound $O(\frac{\log n}{\log \log n})$. And amazingly, the third method has a max load of $O(\log \log n)$ – an exponential improvement. The proof may be found in [1] and the implementation in PRAM may be found in [3] with reference to the older methods and developments as well.

Remark There are a few essential rules of probability that are useful to know.

- Count (Pr[roll $\leq$ 3 on fair 6 sided die] = (1+1+1+0+0+0)/6 = 1/2).
- Add and Multiply (Pr[roll $\leq$ 3 on (2,1,1,1,1,1) die] = (1*2+1+1)/7 = 4/7).
- Union bound (Pr[roll $\leq$ 3 or odd] $\leq$ Pr[$\leq$ 3]+Pr[odd]).

To show the second method's max load, consider a single bin.

$$
\begin{aligned}
\Pr[\text{Bin 1 has} \geq k \text{ balls}] &= \sum_{i=k}^{n} \left(\frac{n!}{i!(n-i)!}\right) \left(\frac{1}{n}^{i}\right) \left(1 - \frac{1}{n}\right)^{n-i} \\
&\leq \sum_{i=k}^{n} \left(\frac{ne}{i}\right)^{i} \left(\frac{1}{n}\right)^{i} \\
&\leq \sum_{i=k}^{n} \left(\frac{e}{i}\right)^{i} \\
&\leq 2\left(\frac{e}{k}\right)^{k} \\
&\leq 2\left(\frac{1}{2}\right)^{k} \\
&\leq 2\left(\frac{1}{2}\right)^{2\log n} \\
&= \frac{2}{n^2}
\end{aligned}
$$

for $k > 2e$ and $k = 2\log n$ (so $n > 2^e \approx 6.6$). Then by union bound, Pr[any bin has $\geq 2\log n$ balls] $\leq \frac{2}{n}$. As n becomes large, the probability becomes very small, so the probability of the opposite happening becomes very high (close to 1). Therefore, with high probability, the maximum load is $O(\log n)$. (This bound may be tightened by using $\frac{\log n}{\log \log n}$ instead of $\log n$ for k.)

References

[1] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *Siam J. Comput.*, 29(1):180–200, 1994. `http://citeseer.nj.nec.com/azar94balanced.html`.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, 1998.

[3] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient pram simulation on a distributed memory machine. *Twenty-fourth ACM SYmposium on Theory of Computing*, 1992. `http://citeseer.nj.nec.com/karp92efficient.html`.

[4] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. *Proceedings of Multimedia Computing and Networking*, 2002. `http://citeseer.nj.nec.com/saroiu02measurement.html`.