Computer Science Division
UC Berkeley
CS273: Parallel Models, Spring 2001 Lecturer: Satish Rao
Lecture #4 and 5

Lecture Outline:

1. **PRAM definition**
   - **Linear relaxation on CRCW, CREW, EREW.**
   - **Feasibility.**
2. **Module PRAM definition**
3. **Simulation of PRAM on Module PRAM**
4. **Discussion of Data Parallel Programming on Vector Processors.**

1 Informal Definitions

We consider a program to have efficient speedup for a problem on a p-processor computer when the time on the parallel computer is at least a factor of $p/2$ less than the time on a sequential computer. Perfect speedup is when the speedup is exactly $p - o(p)$.

2 The PRAM

The PRAM (Parallel Random Access Machine) consists of p processors and a shared memory. The processors operate synchronously with a single program and each can access any piece of memory in unit time. Furthermore, one must specify how or even whether to deal with the problem of two or more processors writing to or reading from the same memory location.

2.1 Programming PRAMS: An example

The most powerful type of PRAM is the CRCW + (Concurrent Read, Concurrent Write, plus) PRAM. In such a PRAM, several processors may read a memory location and several processors may write a memory location. When more than one processor writes a location, we assume that the resulting value is the sum of the written values. This is, of course, a very powerful operation considering that one can sum p numbers in one step using this model.

We sketch an algorithm for implementing iterative relaxation (repeated sparse matrix vector multiply.) We restate the problem below as a graph relaxation problem.

Problem : Directed graph $G = (V, E)$; $w : E \to R$; length $|V|$ vector x^{input}; integer T

Computation: $x^0 = x^{input}$, $x_j^t = \sum_{(ji) \in E} w(ji) x_j^{(t-1)}$

Output: x^T.

A CRCW program for performing iterative relaxation is described below.

Relax (G, T, x^{input})
$x^{old} = x^{input}$
for(t=1 **to** T)
  $x^{new} = \overline{0}$
  for each $e = (j, i) \in E$ **in PARALLEL**
    x_i^{new} = **Combine** $w(j,i) x_j^{old}$ **with** $+$
  $x^{old} = x^{new}$
return x^{new}

The Combine command indicates that x_j^{new} is the sum of all the values of the form $w(i,j)x_i^{old}$. This program runs on an $|E|$ processor CRCW PRAM in time T, a sequential computer requires $T|E|$. Thus, we obtain perfect speedup of p on any iterative relaxation problem where

$$|E| > p$$

(and as long as p divides $|E|$ exactly due to load imbalance problems. Without this condition we still obtain efficient speedup rather than perfect speedup.)

A perhaps weaker model for contention resolution in the PRAM is the Concurrent Read Exclusive Write (CREW). Here we can write

Relax (G, T, x^{input})
1: $x^{old} = x^{input}$
2: **for**(t=1 **to** T)
3: $x^{new} = \overline{0}$
4:   **for each node** $i \in V$ **in PARALLEL**
5:   **for each edge** $(j, i) \in E$ **SEQUENTIALLY**
6: $x_i^{new} = x_i^{new} + w(j,i) x_j^{old}$
7: $x^{old} = x^{new}$
8: **return** x^{new}

Notice that here there are no concurrent writes. Every write done to a location x_i^{new} is done by a single processor. Notice that there are, however, concurrent reads. More than one processor may read the value of x_j^{old} in step 6.

The program runs in time d per step of iterative relaxation using $|V|$ processors, where d is the maximum in-degree of any node in G. Thus, when each node has the same degree, the algorithm obtains a speedup of p when

$$|V| > p,$$

(and p divides $|V|$ exactly.) Notice that since $|E|$ is typically larger than $|V|$ and that since many graphs have varying degrees the CRCW PRAM program obtains efficient speedup on a larger class of graphs than this CREW program.

Still, this program has quite general applicability as we will see later in this course.

Finally, we consider the weakest model for contention. That is, that the program is not allowed to contend on either reads or writes.

Relax (G, T, x^{input})
1: $x^{old} = x^{input}$ 2: for(i=1 to T)
3: $x^{new} = \overline{0}$
4: for each node $j \in V$ in PARALLEL
5: for each edge $(j, i) \in E$ SEQUENTIALLY
6: $update(j, i) = w(j, i)x_j^{old}$
7: for each node $i \in V$ in PARALLEL
8: for each edge $(j, i) \in E$ SEQUENTIALLY
9: $x_i^{new} = x_i^{new} + update(j, i)$
10: $x^{old} = x^{new}$
11: return x^{new}

Here, there are no concurrent accesses to memory. Each location x_j^{old} is only read by the processor responsible for node j in step 6. Each location $update(j, i)$ is only written by the processor responsible for vertex j in step 6, and is only read by the processor responsible for vertex i in step 9.

Thus, when each node has the same degree, we obtain a speedup of p when

$$|V| > p,$$

as for the CREW PRAM. We were, however, required to take a little extra care to write the program. Later in this course, we will discuss the relative implementability of the contention resolution models of PRAMs.

2.2 Implementing PRAMS: Some Difficulties.

For now we list some of the difficulties in building any form of the PRAMS used in the previous section.

1. Large Area: $\Omega(pm)$ wire area is required where p is the number of processors and m is the number of memory locations.
2. Long Wires: $\Omega(\sqrt{pm})$ length wires required.
3. Switching Complexity : $\Omega(m + p * \log p)$ switches required.
4. Bandwidth/Latency penalties when crossing chip boundaries, board boundaries and rack boundaries are not accounted for.

While, there are numerous hurdles, perhaps the most glaring is the presence of m, the number of memory locations, in the bounds above. The fact that, for example, a switch is required for every memory location is completely unreasonable; A typical parallel computer consists of perhaps a few hundred processors and megawords of "shared" memory. Thus, we will proceed by showing how to remove the assumption that each location must be simultaneously accessible from each processor.

3 The Module PRAM

We define a module PRAM as a set of p processors and (for the sake of this discussion) p memory modules. In one round the processors can compute for t time and can issue t memory requests. The total time of the round is the

$$\max(t, \max_{b \in \{0,\ldots,p-1\}} C_b),$$

where C_b is defined as the number of memory requests to memory bank b. (Notice that this is, in some sense, an $(1,1)$-BSP machine.)

4 Simulation of the PRAM on the Module PRAM

We wish to simulate a v-processor PRAM program on a p-processor Module PRAM (MPRAM) with p memory banks, in time $v/p + o(v/p)$. We refer to the ratio v/p as the *parallel slackness* of the simulation. In some sense, this defines the level of excess parallelism in the program.

We accomplish our task by hashing the PRAM memory locations into the MPRAM's memory banks.

That is, we choose a hash function, $h : \{1...M\} \to \{0, ..., p-1\}$, that maps each memory location in the PRAM to a memory bank in the MPRAM. We assume that the PRAM program has no knowledge of the particular choice of hash function. (We can assume this since the PRAM programmer should not be adversarial. If a PRAM programmer was adverserial and did know the particular choice of hash functions, the PRAM program could, of course, be written to require v time on the MPRAM for each step of the PRAM.)

For this lecture, we consider a hash function where each memory location is mapped to a random memory bank. That is, for all i, $h(i)$ is chosen uniformly at random from $\{0, ..., p-1\}$. We can prove the following lemma.

Lemma 1 ***For a sequence of*** v ***memory requests*** $\mathcal{B} = \{m_0, ..., m_{v-1}\}$ ***and*** $v/p = \omega(\log p)$ ***and any constant*** c***,***

$$C_b(\mathcal{M}) = v/p + o(v/p),$$

with probability $1 - 1/p^c$***, where*** $C_b(\mathcal{M})$ ***is defined as the number of messages in*** $|\{m_i \in \mathcal{M} | h(m_i) = b\}|$.

We can use this lemma to conclude that with high probability a p**-processor MPRAM can simulate any** v**-processor PRAM program that runs for at most polynomial time in the number of processors in time** $v/p + o(v/p)$ **per step of PRAM program as long as** $v/p = \omega(\log p)$. **Thus, this gives a** $1 + \epsilon$ **efficient simulation of a PRAM on an MPRAM as long as** v **is sufficiently larger than** p**, i.e., there is sufficient parallel slackness.**

We first prove the following lemma with weaker bounds. We can use the following lemma to show a constant efficiency simulation.

Lemma 2 ***For a sequence of*** v ***memory requests*** $\mathcal{B} = \{m_0, ..., m_{v-1}\}$ ***and*** $v/p = \Omega(\log p)$ ***and any constant*** c***,***

$$C_b(\mathcal{M}) = O(v/p),$$

with probability $1 - 1/p^c$***, where*** $C_b(\mathcal{M})$ ***is defined as the number of messages in*** $|\{m_i \in \mathcal{M} | h(m_i) = b\}|$.

Proof: **We wish to bound**

$$Pr[C_b > c_1 v/p]$$

where we let C_b **denote** $C_b(\mathcal{M})$. **When the event** $C_b > c_1 v/p = L$ **occurs, we know that at least** L **messages from** $\mathcal{M}$ **must have chosen bank** b. **For any set of** L **messages the probability that they all choose** b **is** $(1/p)^L$ **since each message chooses** b **with probability** $1/p$ **and its choice is independent of any other messages choice. The total number of ways to choose** L **messages from the** v **messages in** $\mathcal{M}$ **is** $\binom{v}{L}$.

Using these facts, we can derive the following inequalities.

$$\begin{aligned} Pr[C_b > c_1 v/p] &\le \binom{v}{L}\left(\frac{1}{p}\right)^L \\ &\le \left(\frac{ve}{L}\right)^L \left(\frac{1}{p}\right)^L \\ &\le \left(\frac{ve}{c_1(v/p)}\right)^{c_1(v/p)} \left(\frac{1}{p}\right)^{c_1(v/p)} \\ &\le \left(\frac{e}{c_1}\right)^{c_1(v/p)} \end{aligned}$$

The last expression can be bounded by $1/p^c$ for arbitrarily large c by choosing c_1 sufficiently large and by assuming that $v/p = \Omega(\log p)$.

(We remark here that for this proof the only fact that we used about the hash function is that for the probability that a message chooses a bank b is $1/p$ even given that $L-1$ messages choose bank b. For the technically oriented reader, this means that we only used L-wise independence.) $\Box$

We complete this section by proving lemma 1.

Proof of lemma 1: In this proof we use a lemma of Chernoff/Hoeffding.

Lemma 3 *Given n independent $0,1$ random variables $x_1,\ldots,x_n$ each with mean μ,*

$$Pr[\sum_i x_i > (\mu+\alpha)n] \le e^{-\frac{\alpha^2 n}{3\mu}},$$

as long as $\alpha < \min(\mu, 1-\mu)$.

We consider $n = v$ variables x_i, where x_i is an indicator variable for the event that $h(m_i) = b$ for message $m_i \in \mathcal{M}$. The total contention C_b is the $\sum_i x_i$. The mean of each x_i is $0 \cdot Pr[x_i = 0] + 1 \cdot Pr[x_i = 1] = 1/p$. That is, $\mu = 1/p$.

We can now apply Hoeffding's lemma.

$$\begin{aligned} Pr[\sum_i x_i > (1/p+\alpha)v] &\le e^{-\frac{\alpha^2 v}{3\cdot 1/p}} \\ &\le e^{-\frac{\alpha^2 vp}{3}}. \end{aligned}$$

The probablity can be made smaller than $1/p^c$ for any constant c, by choosing $\alpha > \sqrt{3c\log p/vp}$. Thus, the total overload, $\alpha v = \sqrt{3cv\log p/p}$ is $o(v/p)$ as long as v/p is $\omega(v/p)$. This proves lemma 1. $\Box$

Thus, one can obtain an $1+o(1)$ efficient simulation of a PRAM on an MPRAM as long as there is $\omega(\log p)$ paralell slackness.

5 "Hashing" to reduce memory contention: An Example.

While the arguments of the previous section seem quite unconnected to practice, hashing has been used to avoid memory bank contention in systems. We consider, for example, a vector supercomputer where the CPU typically issues memory references faster than the cycle time of any particular memory module. The architecture thus typically consists of several memory modules (or banks) and some kind of network to route the reference to the appropriate memory module.

By squinting a bit, one can see this situation is analogous to that described above with the Module PRAM.

Here, as in the previous discussion, a particular program can interact with the mapping of the address space to the modules either well or badly.

For the CRAY and other vector multiprocessors, the designers considered an important class of programs to be "Dense Matrix" operations, e.g., matrix multiply. With these applications in mind the CRAY designers mapped the address space so that either a row or a column can be scanned into the CPU so that each successive memory reference is to a separate memory module (with restrictions such as the row and column dimensions must be prime relative to the number of memory modules.) They in turn implemented programming constructs that issued a sequence of memory requests where each is request is a fixed distance from the previous memory request in the memory address space. We refer to such memory request streams as "fixed stride" streams. (Think about how to address the row or column of a matrix using such a construct.)

This, specially engineered "hashing" of the address space to the modules and its use in programming worked beautifully for the well-structured applications that it was designed for.

On the other hand, applications that were less well-structured needed more powerful constructs. For example, to perform linear relaxation on an unstructured graph one should be able to quickly issue a memory request for the neighbors of each node in sequence. There is no particular reason to believe that these memory requests obey the "fixed stride" condition. In general, they do not obey the "fixed stride" condition.

Some vector computers included a more general pair of commands to address this problem. One command allows one to index the contents of an array A using an index array I and pack the result into an array R. That is, they implemented the command

$$Gather(A, I, R, NumItems),$$

which performs the operation $R[i] = A[I[i]]$ for $i \in \{0, ..., NumItems - 1\}$. A "reverse" command,

$$Scatter(A, I, R, NumItems),$$

performs the operation $A[I[i]] = R[i]$ for $i \in \{0, \ldots, NumItems - 1\}$.

As an exercise, you should think about how one implements either any PRAM program or in particular a linear relaxation algorithm using such constructs.

We note that your program may have bad performance if, for example, in some $Gather(A, I, R...)$ command many contiguous elements of the sequence $\{A[I[0]], A[I[1], ...\}$ were mapped to the same memory bank. By using the results of the previous

section, however, one can argue that for a sufficiently long stream of memory requests, one usually can use the memory banks nearly as well as for the "fixed stride" memory access patterns that these machines are typically used for.

Thus, if the vector computers have the ability to efficiently route more requests than memory banks one can do general computations at the peak CPU speed.

(This approach is essentially the approach used in NESL. This is a data parallel (or PRAM style) language developed at CMU by Guy Blelloch and his students. Please refer to their paper for a high-level description of some aspects of their system.)

6 Conclusion

As in lecture 1, we (hope to) have made two points. First, we introduced some analytical tools for discussing and analyzing certain aspects of parallel computing. Second, we showed that by keeping a general computation model in mind, one can increase the applicability of a machine by either programming the machine appropriately or by modifying the machine (hopefully slightly) to allow the hardware to support the general model.