L5: I/O Extensions

(Cache Oblivious, Parallel External)

scribe(s): *Evan Young and Daniel Perry*

5.1 Overview

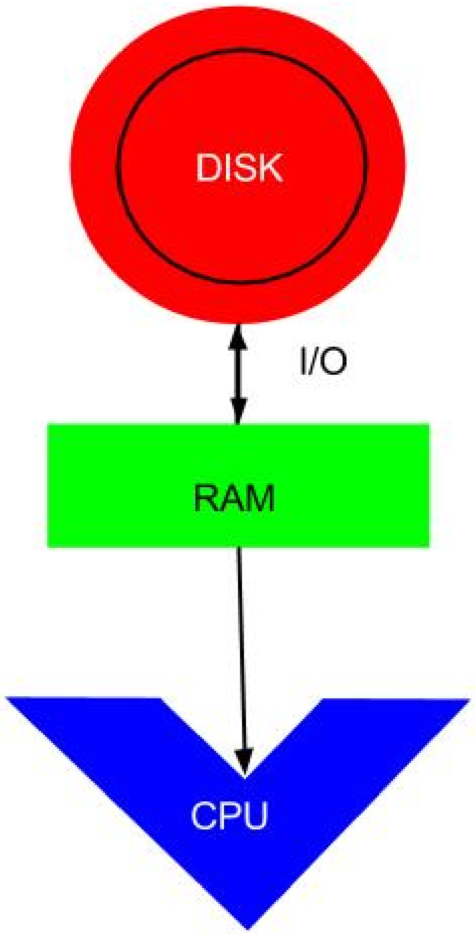

Figure 5.1: A Sample System

Here we have the standard setup. We have one disk D, one RAM, and a CPU. This might not be representative of the most common system. There are systems that are more complicated with multiple CPU's and disks. We will look at those later in these notes. As always we will use N for input size, B for block size, and M for memory size.

5.1.1 Some Familiar Bounds

Below we have some bounds that we have talked about before.

- Scan : $\mathcal{O}(N/B)$
- Sort : $\mathcal{O}(N/B \log_{M/B}(N/B))$
- I/O Search (B-Tree) : $\mathcal{O}(\log_B(N/B))$

Some of the things we'll look at are some other models to see how these bounds change.

5.2 Cache Oblivious Algorithms

So first we should look at is trying to get rid of the problem of having multiple bottlenecks in the hierarchy. This can be addressed with Cache Oblivious Algorithms. The goal of these algorithms is to have a bound on the I/O's in terms of N, M, and B but with no knowledge of M and B. Keep in mind that the our bounds will be, partially, in terms of B but we want to have to know what B is ahead of time. There are some advantages to this:

- They can run on multiple architectures.
- You don't have to find B and recompile.
- They allow for M to vary over time.
- Abstraction allows for the algorithm to work, no matter the heiarchy- i.e. it doesn't matter where the bottleneck is.
- The code is portable and can be moved to many different systems.

(*Note: often algorithms with knowledge of M and B will have better bounds or run faster.*)

5.2.1 Model Assumptions

In order for these algorithms to work there are a number of assumptions about our model that we must make.

- **Ideal Cache** - It must be such that every time you put something new in your cache you have to replace some thing that is already there and this must be the item furthest in the future. This actually quite impossible however if we use the Least Recently Used (LRU), or an approximation, we can achieve this within a constant factor.
- **Full Associativity** - This means that any block you grab from disk can go anywhere in the cache. This can be paritcularly difficult. This difficulty can be avoided by using some tricks from hashing.
- **Tall Cache** This can also be stated as the Large Cache Assumption. You generally need a pretty tall cache with at least $M > B^2$ and usually $M > B^{1+a}$ for some constant $a > 0$.

5.2.2 Scanning

So what is the simplest algorithm that we can look at that is cache oblivious; it is scanning. Let's say we have a large data set and we want to find the max element in the set. As we have known this process before it can be done in $\mathcal{O}(N/B)$. What are the bounds for scanning as a cache oblivious algorithm. First, we can assume that the data is stored in a contiguous part of the disk. It helps to think of the disk as a long tape and you start at the beginning of the tape. You have no control over the arrangement of the blocks so the worst case scenario is $\lceil N/B + 1 \rceil$ I/O's.

5.2.3 Divide and Conquer

Another of these algorithms that we can look at is Divide and Conquer. Most algorithms do some sort of this. If we are doing the quicksort, for example, we are going to keep dividing until this size is less than or equal to M and then we can solve in memory. We can still have this structure in a cache oblivious form. We don't know what M is so we are going to need to divide smaller than we would otherwise. This can work really well when working with a GPU because there is some parallelizing that can be done. What else can we do to improve the bounds here:

Algorithm 5.2.1 Median(D, $W/2$)

```
Split $D$ into $N/5$ sets ($M_i$) of size $5$
**for** each set $M_i$ find median **do**
  Compute Median of $M_i$
Split $D$ into $L = \{d = D \mid d < m\}$ and $R = \{d = D \mid d \geq m\}$
**if** $|L| \geq N/2$ **then**
  Run Median $(L, N/2)$
**else**
  Run Median $(R, N/2 - |L|)$
```

Analysis: For this algorithm we see the following costs

(A) Split D into $N/5$ sets of size 5: This step is basically free.

(B) Find median of each set M_i : 2 scans (2(N/B) I/O's)

(C) Recursively compute median m on Q: This set is size $N/5$. So it will decrease geometrically.

(D) Split D into L and R: This step is linear because (C) decreases geometrically.

(E) Recur on L or R: This is a recursive call of $N(7/10)$ because $\max\{|L|, |R|\} \leq N(7/10)$

So basically our run time is

$$T(N) = \mathcal{O}(N/B) + N/5 + T((7/10)N) = \mathcal{O}(N/B).$$

For further matrerial on this please see this pdf.

5.2.4 Binary Search

For a Binary Search we can get the bound of

$$\Theta(\log N - \log B).$$

5.2.5 Merge Sort

For Merge Sort we get the bound

$$\mathcal{O}((N/B)\log_2(N/B)).$$

5.3 Parallel Disk Model

Another setup we will look at is the Parallel Disk Model. Here we have one CPU and one RAM but instead of one disk you have d disks $(D_1, D_2, \ldots, D_d)$. There is block size B. With this set up you can increase your speed by a factor of d. For example, you can scan in

$$\mathcal{O}\left(\frac{N}{B \cdot d}\right)$$

by parallelizing. Also you can parallelize and sort at

$$\mathcal{O}\left(\frac{N}{B \cdot d}\log_{M/B}\frac{N}{B}\right)$$

.

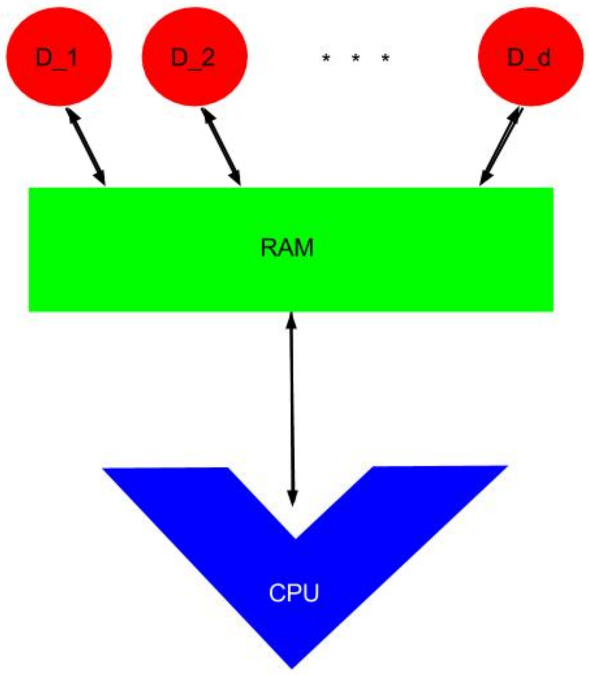

Figure 5.2: Parallel Disk Model

5.4 Parallel External Memory

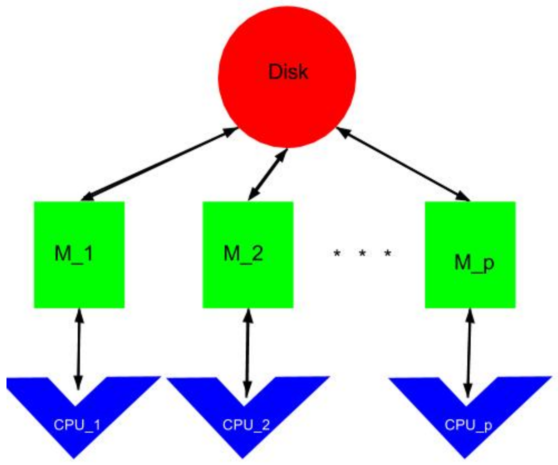

Figure 5.3: Parallel Disk Model

The last setup we'll look at is the Parallel External Memory. In this setup we have one disk and p CPU's each with a private cache of size M. The question is can we increase speed by a factor of p? Yes we can. We can divide and conquer and do everything in memory. We can scan with runtime

$$\mathcal{O}\left(\frac{N}{B \cdot p} + \log p\right)$$

and we can sort in

$$\mathcal{O}\left(\frac{N}{B \cdot p} \log_{M/B} \frac{N}{B}\right).$$