Chapter 12

Parallel Algorithms

12.1 Models of parallel computation

There is a perpetual need for faster computation which is unlikely to be ever satisfied. With device technologies hitting physical limits, alternate computational models are being explored. The *Big Data* phenomenon precedes the coinage of this term by many decades. One of the earliest and natural direction to speed up computation, was to deploy multiple processors instead of a single processor for running the same program. The ideal objective is speed up a program p fold by using p processors simultaneously. A common caveat is that, an egg cannot be boiled faster by employing multiple cooks ! Analogously, a program cannot be executed faster indefinitely by using more and more processors. This is not just because of physical limitations but dependencies between various fragments of the code, imposed by precedence constraints.

At a lower level, namely, in digital hardware design, parallelism is inherent - any circuit can be viewed as a parallel computational model. Signals travel across different paths and components and combine to yield the desired result. In contrast, a program is coded in a very sequential manner and the data flows are often dependent on eachother - just think about a loop that executes in a sequence. Second, for a given problem, one may have to redesign a sequential algorithm to extract more parallelism. In this chapter, we focus on designing fast parallel algorithms for fundamental problems.

A very important facet of parallel algorithm design in the underlying architecture of the computer, viz., how do the processors communicate with each other and access data concurrently. Moreover, is there a common clock across which we can measure the actual running time ? *Synchronization* is an important property that makes parallel algorithm design somewhat more tractable. In more generalized asynchronous models, there are additional issues like deadlock and even convergence is very challenging to analyze.

In this chapter, we will consider synchronous parallel models (sometimes called SIMD) and look at two important models - *Parallel Random Access Machine* (PRAM) and the *Interconnection Network* model. The PRAM model is the parallel counterpart of the popular sequential RAM model where p processors can simultaneously access a common memory called *shared* memory. Clearly, enormous hardware support is required to enable processors to access the shared memory concurrently which will scale with increasing number of processors and memory size. Nevertheless, we adopt a uniform access time assumption for reads and writes. The weakest model is called EREW PRAM or *exclusive read exclusive write* PRAM where all the processors can access memory simultaneously provided that there is no conflict in the accessed locations. The exclusiveness must be guaranteed by the algorithm designer. There are other varations as well, called CREW[1] and CRCW PRAMs that allow read conflicts and write conflicts. Although these are abstract models that are difficult to build, they provide conceptual simplicity for designing algorithms which can subsequently be *compiled* into the weaker models.

The interconnection networks are based on some regular graph topology where the nodes are processors and the edges provide a physical link. The processors communicate with eachother via message passing through the wired links where each link is assumed to take some fixed time. The time to send a message between two processors is proportional to the number of links (edges) between the two processors. This would motivate us to add more links, but there is a tradeoff between the number of edges and the cost and area of the circuit, which is usually built as VLSI circuits. Getting the right data at the right processor is the key to faster execution of the algorithm. This problem is commonly referred to as *routing*. Towards the end of this chapter we will discuss routing algorithms that provides a bridge between the PRAM algorithms and the interconnection networks.

12.2 Sorting and comparison problems

12.2.1 Finding maximum

This is considered to be a trivial problem in the sequential context and there are several ways of computing the maximum using $n - 1$ comparisons. A simple scan suffices where one maintains the maximum of the elements seen so far.

Exercise 12.1 *Show that $n - 1$ comparisons are necessary to find the maximum of n elements.*

[1]**C** denotes concurrent

We want to do many comparisons in parallel so that we can eliminate many elements from further consideration - every comparison eliminates the smaller element. We assume that in each round, each of the available processors compares a single pair. If we want to minimize the number of rounds, we can use $\binom{n}{2}$ processors to do all the pairwise comparisons and output the element that *wins* across all all comparisons. The second phase of locating the element that has not *lost* requires more details in the parallel context and may require several rounds. But is the first phase itself efficient ? We need roughly $\Omega(n^2)$ processors and so the total number of operations far exceeds the sequential bound of $O(n)$ comparisons and does not seem to be cost effective.

Can we reduce the number of processors to $O(n)$. That seems unlikely as we can do at most $\frac{n}{2}$ comparisons in one round by pairing up elements and there will be at least $n/2$ potentail maximum at the end of the first round. We can continue doing the same - pair up the *winners* and compare them in the second round and keep repeating this till we find the maximum. This is similar to a *knockout* tournament where after i rounds there are at most $\frac{n}{2^i}$ potential winners. So after $\log n$ rounds, we can pick the maximum.

How many processors do we need ?

If we do it in a straight-forward manner by assigning one processor to each of the comparisons in any given round, we need $\frac{n}{2}$ processors (which is the maximum across across all rounds). So the processor-time product is $\Omega(n \log n)$, however the total number of comparisons is $\frac{n}{2} + \frac{n}{4} + \ldots \leq n$ which is optimum. So we must explore the reason for the inefficient use of processors.

One possibility is to reduce the number of processors to $p \ll n$ and slow down each round. For example, the $\frac{n}{2}$ first round comparisons can be done using p processors in rougly $\lceil \frac{n}{2p} \rceil$ rounds. This amounts to slowing down round i by a factor $\frac{n}{2^i \cdot p}$ [2] so that the total number of rounds is

$$O(\frac{n}{p} \cdot (\frac{1}{2} + \frac{1}{2^2} + \ldots \frac{1}{2^i}) \leq \frac{n}{p}.$$

By definition, this is optimal work as processor-time product is linear. There is a caveat - we are ignoring any cost associated with assigning the available processors to the prescribed comparisons in each round. This is a crucial component of implementing parallel algorithms called *load balancing* which itself is a non-trivial parallel procedure requiring attention at the system level. We will sketch some possible approaches to this in the section on parallel prefix computation. For now, we ignore this component and therefore we have a parallel algorithm for finding maximum of n elements that require $O(\frac{n}{p})$ parallel time. But this tells us that we can find maximum in $O(1)$ time using $p = \Omega(n)$! Clearly we cannot do it in less than $O(\log n)$ round by the previous algorithm.

[2]ignoring the ceilings to keep the expression simple

So there is a catch - when the number of comparisons falls below p, the time is at least 1, a fact that we ignored in the previous summation. So let us split the summation into two components - one when the number of camparisons is $\geq p$ and the subsequent ones when they are less than p. When they are less than p, we can run the first version in $O(\log p)$ rounds which is now an additive term in the expression for parallel time that is given by $O(\frac{n}{p} + \log p)$. It is now clear that for $p = \Omega(n)$, the running time is $\Omega(\log n)$ and the more interesting observation is that it is minimized for $p = \frac{n}{\log n}$. This leads to a processor-time product $O(n)$ with parallel running time $O(\log n)$.

A simpler way to attain the above bound will be to first let the $p = \frac{n}{\log n}$ processors sequentially find the maximum of (disjoint subsets of) $\log n$ elements in $\log n$ comparisons and then run the first version of $\frac{n}{\log n}$ elements using p processors in $\log(\frac{n}{\log n}) \leq \log n$ parallel steps. This has the added advantage that practically no *load-balancing* is necessary as all the comparisons can be carried out by the suitably indexed processor.

Exercise 12.2 *If a processor has index i , $1 \leq i \leq p$, find out a way of preassigning the comparisons to each of the processors.*
Use the binary tree numbering to label the comparison number.

Can we reduce the number of rounds without sacrificing efficiency ?

Let us revisit the 1-round algorithm and try to improve it. Suppose we have $n^{3/2}$ processors which is substantially less than n^2 processors. We can divide the elements into $\sqrt{n}$ disjoint subsets, and compute their maximum using n processors in a single round. After this round we are still left with $\sqrt{n}$ elements which are candidates for the maximum. However, we can compute their maximum in another round using the one round algorithm. [3] Taking this idea forward, we can express the algorithm in a recursive manner as follows

The recurrence for parallel time can be written in terms of $T^{||}(x, y)$ which represents the parallel time for computing maximum of x elements using y processors. Then, we can write

$$T^{||}(n, n) \leq T^{||}(\sqrt{n}, \sqrt{n}) + T^{||}(\sqrt{n}, n)$$

The second term yields $O(1)$ and with appropriate terminating condition, we can show that $T^{||}(n, n)$ is $O(\log \log n)$. This is indeed better than $O(\log n)$ and the processor-time product can be improved further using the previous technqiues.

Exercise 12.3 *Show how to reduce the number of processors further to $\frac{n}{\log \log n}$ and still retain $T^{||}(n, n/\log \log n) = O(\log \log n)$.*

[3]We are ignoring the cost of load-balancing

Can we improve the parallel running time further ?

This is a very interesting question that requires a different line of argument. We will provide a sketch of the proof. Consider a graph $G = (V, E)$ where $|V| = n$ and $|E| = p$. Every vertex corresponds to an element and an edge denotes a comparison between a pair of elements. We can think about the edges as the set of comparisons done in a single round of the algorithm. Consider an independent subset $W \subset V$. We can assign the largest values to the elements associated with W. Therefore, at the end of the round, there are still $|W|$ elements that are candidates for the maximum. In the next round, we consider the (reduced) graph G_1 on W and the sequence of comparisons in round i corresponds to the independent subsets of G_{i-1}. The number of edges is bound by p. The following result on the size of independent set of a graph, known as Turan's theorem will be useful in our context.

Lemma 12.1 *In an undirected graph with n vertices and m edges, there exists an independent subset of size at least least $\frac{n^2}{m+n}$.*

Proof: We will outline a proof that is based on probabilistic reasoning. Randomly number the vertices V in the range 1 to n, where $n = |V|$ and scan them in an increasing order. A vertex i is added to the independent set $\mathcal{I}$ if all its neighbours are numbered higher than i. Convince yourself that $\mathcal{I}$ is an independent set. We now find a bound for $\mathbb{E}[|\mathcal{I}|]$. A vertex $v \in \mathcal{I}$ iff all the $d(v)$ neighbors ($d(v)$ is the degree of vertex v) are numbered higher than v and the probability of this event is $\frac{1}{d(v)+1}$. Let us define a indicator random variable $I_v = 1$ if v is chsen in $\mathcal{I}$ and 0 otherwise. Then

$$\mathbb{E}[|\mathcal{I}|] = \mathbb{E}[\sum_{v \in V} I_v] = \sum_{v \in V} \frac{1}{d(v)+1}$$

Note that $\sum_v d(v) = 2m$ and the above expression is minimized when all the $d(v)$ are equal, i.e., $d(v) = \frac{2m}{n}$. So $\mathbb{E}[|\mathcal{I}|] \geq \frac{n}{2m/n+1} = \frac{n^2}{2m+n}$. Since the expected value of $|\mathcal{I}|$ is at least $\frac{n^2}{2m+n}$, it implies that for at least one permutation, $\mathcal{I}$ attains this value and therefore the lemma follows. $\Box$

Let n_i $i = 0, 1, 2 \ldots$ denote the vertices in the sequence $G = G_0, G_1, G_2 \ldots G_i$ as defined by the independent sets in the algorithm. Then, from the previous claim

$$n_i \geq \frac{n}{3^{2^i-1}}$$

Exercise 12.4 *For $p = n$, show that after $j = \frac{\log\log n}{2}$ rounds, $n_j > 1$.*

Procedure Odd-even transposition sort for processor(i)

```
1 for $j = 1$ to $\lceil n/2 \rceil$ do
      for $p = 1, 2$ do
2         if If $i$ is odd then
3             Compare and exchange with processor $i + 1$ ;
          else
4             Compare and exchange with processor $i - 1$ ;
5         if If $i$ is even then
6             Compare and exchange with processor $i + 1$ ;
          else
7             Compare and exchange with processor $i - 1$ ;
```

Figure 12.1: Parallel odd-even transposition sort

12.2.2 Sorting

Let us discuss sorting on the interconnection network model where each processor initially holds an element and after sorting the processor indexed i should contain the rank i element. The simplest interconnection network is a linear arry of n processing elements. Since the diameter is $n - 1$, we cannot sort faster than $\Omega(n)$ parallel steps.

An intuitive approach to sort is compare and exchange neighboring elements with the smaller element going to the smaller index. This can be done simultaneously for all (disjoint) pairs. To make this more concrete, we will we define rounds with each round containing two phases - odd-even and even-odd. In the odd-even phase, each odd numbered processor compares its element with the larger even number element and in the odd-even phase, each even numbered processor compares with the higher odd numbered processor.

We repeat this over many rounds till the elements are sorted. To argue that it will indeed be sorted, consider the smallest element. In every comparison it will start moving towards processor numbered 1 which is its final destination. Once it reaches, it will continue to be there. Subsequently, we can consider the next element which will finally reside in the processor numbered 2 and so on. Note that once elements reach their final destination and all the smaller elements have also in their correct location, we can ignore them for future comaprisons. Therefore the array will be sorted after no more than n^2 rounds as it takes at most n rounds for any element to reach its final destination. This analysis is not encouraging from the presective of speed-up as it only matches bubble-sort. To imrove our analysis, we must *track* the simultaneous movements of elements rather than 1 at a time. To simplify our analysis, we invoke the following result.

Lemma 12.2 (0-1 principle) *If any sorting algorithm sorts all possible inputs of 0's and 1's correctly, then it sorts all possible inputs correctly.*

We omit the proof here but we note that there are only 2^n possible 0-1 inputs of length n where as there are n permutations. The converse clearly holds.

So, let us analyze the above algorithm, called **odd-even transposition** sort for inputs restricted to $\{0, 1\}^n$. Such an input is considered sorted if all the 0's are to the left of all 1's. Let us track the movement of the leftmost 0 over successive rounds of comparisons. It is clear that the leftmost 0 will keep moving till it reaches processor 1[4]. If this 0 is in position k in the beginning it will reach within at most $\lceil k/2 \rceil$ rounds. If we consider the next 0 (leftmost 0 among the remaining elements), to be denoted by 0_2, the only element that can block its leftward progress is the leftmost 0 and this can happen at most once. Indeed, after the leftmost 0 is no longer the immediate left neighbor of 0_2, this elements will keep moving left till it reaches its final destination. If we denote the sequence of 0's using 0_i for the i-th zero, we can prove the following by induction.

Exercise 12.5 *The element 0_i may not move for at most i phases ($\lceil i/2 \rceil$ rounds) in the beginning and subsequently it moves in every phase until it reaches its final destination.*

Since the final destination of 0_i is i, and it can be at most $n - i$ away from the final destination, the total number of phases for it to reach processor i is $i + n - i = n$. Note that the above argument holds simultaenously all the elements and so all the 0's (and therefore all 1's) are in their final positions within n phases or $\lceil n/2 \rceil$ rounds.

Next, we consider the two dimensional mesh which is a widely used parallel architecture. For sorting, we can choose from some of the standard indexing schemes like row-major - every row i contains elements smaller than the next row, and the elements in a row are sorted form left to right. Column-major has the same property across columns, snake-like row major where the alternate rows are sored from left to right and the others from right to left.

Suppose we sort rows and columns in successive phases. Does this converge to a sorted array ? No, you can construct an input where each row is sorted and every column is sorted (top to bottom) but not all elements are in their final position. A small change fixes this problem - sort rows according to snake-like row major and the columns from top to bottom. The more interesting question is how many rounds of row/column sorts are required ?

[4]We are assuming that there is at least one 0 in the input otherwise there is nothing to prove.

Procedure Shearsort algorithm for rectangular mesh(m, n)

```
1 for $j = 1$ to $\lceil \log m \rceil$ do
2     Sort rows in alternating directions ;
3     Sort columns from top to bottom ;
```

Figure 12.2: Shearsort

Each row/column can be sorted using the odd-even transposition sort, so if we need t iterations, then the total parallel steps will be $O(t\sqrt{n})$ for a $\sqrt{n} \times \sqrt{n}$ array. To simplify our analysis we will again invoke the 0-1 principle. First consider only two rows of 0's and 1's. Let us sort the first row from left to right and the second row from right to left. Then we do the column sort.

Lemma 12.3 *Either the top row will contain only 0's or the bottom row will contain only 1's - at least one of the conditions will hold.*

We define a row *clean* if it consists of only 0's or only 1's and *dirty otherwise.* According the above observation (prove it rigorously), after the row and column sort, at least one of the rows is *clean* so that in the next itseration (sorting rows), the array is sorted. Now extend the analysis to an $m \times m$ array. After one row sort and column sort, at least half the rows are clean. Indeed each consecutive pair of rows produces at least one clean row and they continue to remain clear thereafter. In each iteration, the number of dirty rows reduce by at least a factor of 2, leading to $\log m$ iterations for all (but one) row to be clean. One more row sorting completes the ordering.

Lemma 12.4 *The number of iterations of alternately sorting rows and columns results in a sorted array after at most* $\log m$ *iterations in a* $m \times n$ *array.*

Therefore a $\sqrt{n} \times \sqrt{n}$ array can be sorted in $O(\sqrt{n} \log n)$ parallel steps. This rather simple algorithm, called **Shearsort** is close to being optimal, within a factor of $O(\log n)$. In the exercises, you will be lead through a $O(\sqrt{n})$ algorithm based on a recursive variation of Shearsort.

It is not difficult to extend Shearsort to higher dimesnional meshes but it doesn't lead to a $O(\log n)$ time sorting algorithm in the hypercubic network. Obtaining an ideal speed-up sorting algorithm on an n-processor interconnection network is very challenging and had required many non-trivial ideas both in terms of algorithms and the network topology.

In the shared memory model like PRAM, one can obtain an $O(\log n)$ time algorithm by generalizing the idea of quicksort.

Procedure Parallel partition Sort

```
  Input $X = \{x_1, x_2 \ldots x_n\}$ ;
1 if $n \leq C$ then
2     Sort using any sequential algorithm
  else
3     Choose a uniform random sample $\mathcal{R}$ of size $\sqrt{n}$ ;
4     Sort $\mathcal{R}$ - let $r_1, r_2 \ldots$ denote the sorted set ;
5     Let $N_i = \{x | r_{i-1} \leq x \leq r_i\}$ be the $i$-th subproblem ;
6     In parallel do ;
7     Recursively partition sort $X_i$ ;
```

Figure 12.3: Partition Sort

The analysis requires use of probabilistic inequalities like Chernoff bounds that enable us to obtain good control on the subproblem sizes for the recursive calls. Roughly speaking, if we can induce $O(\sqrt{n})$ bound on the size of the recursive calls when we partition n elements into $\sqrt{n}$ intervals, the number of levels in bounded by $O(\log \log n)$[5]. Moreover, each level can be done in time $O(\log n_i)$ where n_i is the maximum subproblem size in level i. Then the total parallel running time is $\sum_i O(\log n) + \log(\sqrt{n}) + \ldots \log(n^{1/2^i}) = O(\log n)$.

Here we outline the proof for bounding the size of subprobems using a uniforrmly sampled subset $\mathcal{R} \subset S$ where $|S = n$ and $\mathcal{R}$ has size about r.

Lemma 12.5 *Suppose every element of S is sampled uniformly and independently with probability $\frac{r}{n}$. Then the number of unsampled elements of S in any interval induced by $\mathcal{R}$ is bounded by $O(\frac{n \log r}{r})$ with probabiity at least $1 - \frac{1}{r^{\Omega(1)}}$.*

Proof: Let $x_1, x_2 \ldots$ be the sorted sequence of S and let random variables

$$X_i = 1 \text{ if } x_i \text{ is sampled and } 0 \text{ otherwise } 1 \leq i \leq n$$

So the expected number of sampled elements $= \sum_{i=1}^{n} \Pr[X_i = 1] = n \cdot \frac{r}{n} = r$. The actual number may vary but let us assume that $|R| = r$. Suppose, the number of unsampled elements between two consecutive sampled elements $[r_i, r_{i+1}]$ be denoted by Y_i (Y_0 is the number of elements before the first sampled element). Since the elements are sampled independently, $\Pr[|Y_i| = k] = \frac{r}{n} \cdot (1 - \frac{r}{n})^k$. It follows that

$$\Pr[|Y_i| \geq k] = \sum_{i=k} \frac{r}{n} \cdot (1 - \frac{r}{n})^i \leq (1 - \frac{r}{n})^k$$

[5]This is true for size of subproblems bounded by n^c for any $c < 1$

For $k = \frac{cn \log r}{r}$, this is less than $e^{c \log r} \leq \frac{1}{r^c}$.

If any of the intervals has more than $k = c\frac{n \log r}{r}$ unsampled elements then some pair of consecutive sampled elements $[r_i, r_{i+1}]$ have more than k elements unsampled elements between them and we have computed the probability of this event. So among the $\binom{r}{2}$ pairs of elements, the r consecutive pairs are the ones that are relevant events for us. In other words, the previous calculations showed that for a pair (r', r''),

$$\Pr[|(r', r'') \cap S| \geq k | r', r'' \text{ are consecutive }] \leq \frac{1}{r^c}$$

Since $Pr[A|B] \geq \Pr[A \cap B]$ we obtain

$$\Pr[|(r', r'') \cap S| \geq k \text{ and } r', r'' \text{ are consecutive }] \leq \frac{1}{r^c}$$

So, for all the pairs, by the union bound, the probability that there is any consecutive sampled pair with more than k unsampled elements is $O(\frac{r^2}{r^c})$. For $c \geq 3$, this is less than $1/r$.

From Markov's inequality, the number of samples exceeds r^2 is less than $\frac{1}{r}$. The reason that our sampling fails to ensure gaps less than $cn \log r/r$ is due to one of the following events
(i) Sample size exceeds n^2 (ii) Given that sample size is less than r^2, the gap exceeds k.
This works out as $O(\frac{1}{r})$ as the union of the probabilities. Note that we can increase c to keep the union bound less than $1/r$ for r^2 samples. □

12.3 Parallel Prefix

Given elements $x_1, x_2 \ldots x_n$ and an associative binary operator $\odot$, we want to compute

$$y_i = x_1 \odot x_2 \ldots x_i \quad i = 1, 2, \ldots n$$

Think about $\odot$ as addition or multiplication and while this may seem trivial in the sequential context, the prefix computation is one of the most fundamental problems in parallel computation and have extensive applications.

Note that $y_n = x_1 \odot x_2 \ldots x_n$ can be computed as a binary tree computation structure in $O(\log n)$ parallel steps. We need the other terms as well. Let $y_{i,j} = x_i \odot x_{i+1} \ldots x_j$. Then we can express a recursive computation procedure as given in Figure 12.4. Let $T^{||}(x, y)$ represent the parallel time to compute the prefix of x inputs using y processors. For the above algorithm, we obtain the following recurrence

$$T^{||}(n, n) = T^{||}(n/2, n/2) + O(1)$$

Procedure Prefix computation of n elements prefix(x_a, x_b)

```
1 if  If $b - a \geq 1$ then
2     $c = \lfloor \frac{a+b}{2} \rfloor$ ;
3     In parallel do
4     prefix (a,c) , prefix (c+1,b) ;
5     end parallel ;
6     Return ( prefix (a,c) , $y_{a,c}\odot$ prefix (c+1 , b) (* $y_{a,c}$ is available in prefix
        (a,c) *)
  else
7     Return $x_a$ ;
```

Figure 12.4: Parallel Prefix Computation

The first term represents the time for two (parallel) recursive calls of half the size and the second set of outputs will be multiplied with the term $y_{1,n/2}$ that is captured by the additive constant. The solution is $T^{||}(n,n) = O(\log n)$. Note that this is not optimal work since the prefix can be computed sequentially in n operations, whereas the processor-time product is $O(n \log n)$ for the above algorithm.

This can be improved by reducing the processors by a factor of $\log n$ and computing the prefix of disjoint blocks of $\log n$ elements and then combining them according to the method described below.

Let $z_i = x_{(i-1)k+1} \odot x_{(i-1)k+2} \odot \ldots x_{ik}$ for some integer k. We can perform prefix computation on z_is and from these we can compute the prefix on the x_i's easily by combining with the prefix sums within each k-block of the x_is. For example let us consider elements $x_1, x_2 \ldots x_{100}$ and $k = 10$. Then

$$y_{27} = x_1 \odot x_2 \ldots x_{27} = (x_1 \odot x_2 \ldots \odot x_{10}) \odot (x_{11} \odot x_{12} \ldots \odot x_{20}) \odot (x_{21} \odot \ldots x_{27})$$

$$= z_1 \odot z_2 \odot (x_{21} \odot x_{22} \ldots \odot x_{27}).$$

The last term (in paranthesis) is computed within the block as prefix of 10 elements $x_{21}, x_{22} \ldots x_{30}$ and the rest is computed as prefix on the z_is.

Exercise 12.6 *Analyze the parallel running time of the above approach by writing the appropriate recurrence.*
What is the best value of k for minimizing the parallel running time ?

12.3.1 Applications

Parallel compaction

A very scenario in parallel computation in periodic compaction of active processes so that the available processors can be utilized effective. Even in a structured process flow graph like a binary tree, at every level, half of the elements drop out of future computation. The available processors should be equitably distributed to the active elements, so that the parallel time is minimized.

One way to achieve is to think of the elements[6] in an array where we tag them as **0** or **1** to denote if they are dead or active. If we can compress the 1's to one side of the array, then we can find out how many elements are active, say m. If we have p processors then we can distribute them equally such that every processor is allocated roughly $\frac{m}{p}$ active elements.

This is easily achieved by running parallel prefix on the elements with the operation $\odot$ defined as addition. This is known as *prefix sum*. The i-th 1 gets a label i and it can be moved to the i-th location without any conflicts. Consider the array

$$1\ ,\ 0,\ 0\ 1,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0,\ 0\ ,\ 1,\ 1$$

After computing prefix sum, we obtain the labels y_i's as
1 , 1, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 6

Then we can move the 1's to their appropriate locations and this can be done in parallel time n/p for $p \leq n/\log n$.

Simulating a DFA

Given a DFA M, let $\delta_M(Q, w)$ denote the *transition vector* whose i-th component is $\delta(q_i, w)$, $q_i \in Q$, where δ denotes the transition function of M. Recall that $\delta(q, a \cdot w) = \delta(\delta(q, a), w)$ for $a \in \Sigma$. So the transition vector gives us the final states for each of the $|Q|$ starting states.

Let $w = w_1 w_2 \ldots w_k$ where $w_i \in \Sigma$ - for convenience let us assume that the length of the input string is a power of 2. We will use the notation $w_{i,j}$ to denote the substring $w_1 \cdot w_{i+1} \ldots w_j$.

Claim δ_M is associative, i.e., $\delta_M(Q, w_1 \cdot (w_2 w_3)) = \delta_M(Q, w_1 \cdot (w_2 \cdot w_3))$. Moreover

$$\delta_M(Q, w_{i,\ell}) = \delta_M(\delta_M(Q, w_{i,j}), w_{j+1,\ell}).$$

Alternately, you can express $\delta_M(a)$ as a $|Q| \times |Q|$ matrix A where $A^a_{i,j} = 1$ if $\delta(q_i, a) = q_j$ and 0 otherwise. Then

$$\delta_M(w_1 w_2 \ldots w_k) = A^{w_1} \otimes A^{w_2} \otimes \ldots A^{w_k}$$

where $\bigotimes$ corresponds to matrix multiplication.

For example, let $Q = \{q_0, q_1\}$ and

[6]Can also be thought of as labels of processes

	0	1
q_0	q_0	q_0
q_1	q_0	q_1

For $w = 1011$, $\delta_M(10) = \begin{bmatrix} q_0 \\ q_0 \end{bmatrix}$ and $\delta_M(11) = \begin{bmatrix} q_0 \\ q_1 \end{bmatrix}$.

This yields $\delta_M(1011) = \begin{bmatrix} q_0 \\ q_0 \end{bmatrix}$

So, we can use the prefix computation to compute all the intermediate states in $O(\log n)$ time using $n/\log n$ processors. So this gives us the intermediate states for all possible starting states of which we choose the one corresponding to the actual starting state.

The addition of two binary numbers can be easily represented as state transition of a finite state machine. For example, if the numbers are 1011 and 1101 respectively then one can design a DFA for an adder that takes an input stream (11, 10, 01, 11) which are the pairs of bits starting form the LSB. The successive transitions are made according to the previous carry, so there are two states correspoding to carry 0 and carry 1. Once the carry bits are known then the sum bits can be generated in constant time.

12.4 Basic graph algorithms

Many efficient graph algorithms are based on DFS (depth first search) numbering. A natural approach to designing parallel graph algorithms will be to design an efficient parallel algorothm for DFS. This turned out to be very challenging and no simple solutions are known and there is evidence to suggest that it may not be possible. This has led to interesting alternate techniques for designing parallel graph algorithms. We will consider the problem of constructing connected components in an undirected graph.

12.4.1 List ranking

A basic parallel subroutine involves finding the distance of every node of a given linked list to the end of the list. For concreteness and simplicity, let us consider an array $A[1 \ldots n]$ of size n. Each location i contains an integer j that corresponds to $A[j]$ being the successor of $A[i]$ in the linked list. The head of the list is identified as k such that $A[k]$ contains k, i.e., it points to itself. The purpose of list ranking is to find the distance of every element form the head of the list where the distance of the head to itself is considered as 0.

A sequential algorithm can easily identify the tail of the list (the integer in $[1, n]$ which doesn't appear in the array) and simply traverses the list in n steps. For the parallel algorithm, let us initially assume that we have a processor for every element

Procedure Parallel List ranking(p_i)

```
1 Initialize If $A[i] \neq i$ then $d[i] = 1$ else d[i] = 0 ;
2 while $A[i] > 0$ do
3     $A[i] \leftarrow A[A[i]]$ ;
4     $d[i] \leftarrow d[i] + d[A[i]]$ ;
  Return $d[i]$ ;
```

Figure 12.5: Parallel List ranking

and each processor executes the algorithm in Figure 12.5. To analyze the algorithm, let us renumber the list elements such that x_0 is the head of the list and x_i is at distance i from x_0. The crux of the algorithm is a *doubling* strategy. After $j \geq 1$ steps the processor responsible for x_i, say p_i points to an element k such that k is 2^{j-1} steps away from j. So, in the next step, the distance doubles to 2^j. Of course it cannot be further than the head of the list, so we have to account for that. Notice that, when a processor points to the head, all the smaller numbered processors must also have reached the head. Moreover, they will also have the correct distances.

Lemma 12.6 *After j iterations, p_i points to an element x_k where $k = \max\{i - 2^{j-1}, 0\}$. Equivalently, the distance function from i, $d(i)$ is given by $\min\{2^{j-1}, i\}$.*

Proof: We shall prove it by induction on i which is the distance from the head. For the base case $i = 0$, it is clearly true as it keeps pointing to itself and $d[i]$ does not change. Suppose it is true for all elements $< i$ where $i \geq 1$. Let $l(i)$ be defined as $2^{l(i)} \leq i < 2^{l(i)+1}$.

Further. observe that if after j iterations, a node k points 2^{j-1} ahead, then all nodes $> k$ will also point 2^{j-1} by symmetry. If $2^{l(i)} = i$, then after exactly $l(i)$ iterations, p_i will point to an element that is distance $2^{l(i)-1}$ away since all the elements $k \in [2^{l(i)-1}, 2^{l(i)} - 1]$, will have their lengths doubled during each of the $l(i)$ iterations from induction hypothesis. In the $l(i)+1$ iteration, it will point to the head at correct distance $(2^{l(i)-1} + 2^{l(i)-1} = 2^{l(i)}$, since $x_{2^{l(i)-1}}$ has the correct distance by then.

If $2^{l(i)} < i$, then, a similar argument applies, so that after the $l(i)+1$ iterations, it points $2^{l(i)}$ ahead. During the last iteration, the length would not double but increase additively by $i - 2^{l(i)}$. □

As a corollary, the overall algorithm terminates in $O(\log n)$ steps using n processors. It is a challenging exercise to reduce the number of processors to $\frac{n}{\log n}$ so that the efficiency becomes comparable with the sequential algorithm.

Exercise 12.7 *Reduce the number of processors in the list ranking algorithm to $\frac{n}{\log n}$ without increase the asymptotic time bound.*

In a PRAM model, each iteration can be implemented in $O(1)$ parallel steps but it may require considerably more time in interconnection network since the array elements may be far away and so the pointer updates cannot happen in $O(1)$ steps.

The above algorithm can be generalized to a tree where each node contains a pointer to its (unique) parent. The root points to itself. A list is a special case of a degenerate tree.

Exercise 12.8 *Generalize the above algorithm to a tree and analyze the performance.*

12.4.2 Connected Components

Given an undirected graph $G = (V, E)$, we are interested to know if there is a path from u to w in G where $u, w \in V$. The natural method to solve this problem is to compute maximal subsets of vertices that are connected to eachother[7].

Since it is difficult to compute DFS and BFS numbering in graphs in parallel, the known approaches adopt a strategy similar to computing minimum spanning trees. This approach is somewhat similar to Boruvka's algorithm described earlier. Each vertex starts out as singleton components and they interconnect among eachother using incident edges. A vertex u *hooks* to another vertex w using the edge (u, w) and intermediate connected components are defined by the edges used in the hooking step. The connected components are then merged into a single *meta-vertex* and this step is repeated until the meta-vertices do not have any edges going out. These meta-vertices define the connected components. There are several challenges in order to convert this high level procedure into an efficient parallel algorithm.

C1 What is an appropriate data structure to maintain the meta-vertices ?

C2 What is the hooking strategy so that the intermediate structures can be contracted into a meta-vertex ?

C3 How to reduce the number of parallel phases.

Let us address these issues one by one. For C1, we pick a representative vertex from each component, called the *root* and let other vertices in the same component point to the root. This structure is called a *star* and it can be thought of as a (directed) tree of depth 1. The root points to itself. This is a very simple structure and it is easy to verify if a tree is a star. Each vertex can check if it is connected to the root (which is unique because it points to itself). With sufficient number of processors it can be done in a single parallel phase.

[7]this is an equivalence relation on vertices

For the hooking step, we will enable only the root vertices to perform this, so that the intermediate structures have a unique root (directed trees) that can be contracted into a star. We still have to deal with the following complications.
How do we prevent two (root) vertices hooking on to eachother ? This is a typical problem of *symmetry breaking* in a parallel algorithm where we want exactly of the many (symmetric) possibilities to succeed using some discriminating properties. In this case, we can follow a convention that the smaller numbered vertex can hook on to a larger numbered vertex. We are assuming that all vertices have a unique id between $1 \ldots n$. Moreover, among the eligible vertices that it can hook onto, it will choose one arbitrarily [8] . This leaves open the possibility of several vertices hooking to the same vertex but that does not affect the algorithm.

Let us characterize the structure of the subgraph formed by hooking. The largest numbered vertex in a component cannot hook to any vertex. Each vertex has at most one directed edge going out and there cannot be a cycle in this structure. If we perform shortcut operations for every vertex similar to list ranking then the directed tree will be transformed into a star.

For the hooking step, all the edges going out of a tree are involved, as the star can be considered as a meta-vertex. If all the directed trees are stars and we can ensure that a star combines with another one in every parallel hooking phase, then the number of phases is at most $\log n$. The number of stars will decrease by a factor of two (except those that are already maximal connected components). This would require that we modify the hooking strategy, so that every star gets a chance to combine with another.

Figure 12.6 gives an example where only one star gets hooked in every step because of the symmetry breaking rule. So, we can add another step where a star that could not combine since the root had a larger number (but it lost out to other large numbered roots) can hook to a smaller numbered root. Since the smaller numbered root must have hooked to some *other* tree (since the present tree continues to be a star), this cannot create any cycles and is therefore safe.

The algorithm is described formally in Figure 12.7.

The analysis is based on a potential function that captures the progress of the algorithm in terms of the heights of the trees. Once all the connected components are hooked together, the algorithm can take at most $\log n$ iterations to transform them into stars, based on our analysis of pointer jumping.

We define a potential function $\Phi_i = \sum_{T \in \mathcal{F}} d_i(T)$ where $d_i(T)$ is the depth of a tree T (star has depth 1) in iteration i. Here $\mathcal{F}$ denotes the forest of trees. Note that a tree contains vertices from a single connected component. We can consider each of the components separately and calculate the number of iterations that it takes to

[8]This itself requires symmetry breaking in disguise but we will appeal to the model.

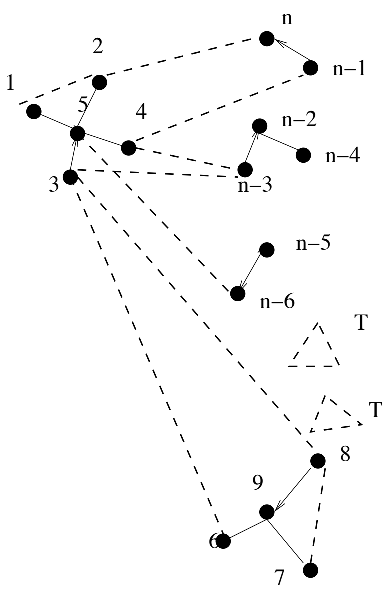

Figure 12.6: The star rooted at vertex 5 is connected all the stars (dotted edges) on the right that have higher numbered vertices. It can only hook on to one of the trees at a time that makes the process effectively sequential. The other stars have no mutual connectivity.

form a single star from the component starting with singleton vertices. The initial value of Φ is $|C|$ where $C \subset V$ is a maximal component and finally we want it to be 1, i.e., a single star.

If T_1 and T_2 are two trees that combine in a single tree T after hooking, it is easily seen that $\Phi(T) \leq \Phi(T_1) + \Phi(T_2)$. For any tree (excluding a star), the height must reduce by a factor of almost 1/2. Actually a tree of depth 3, reduces to 2, which is the worst case. So, $\Phi(C) = \sum_{T \in C} d(T)$ must reduce by a factor 2/3 in every iteration, resulting in overall $O(\log n)$ iterations. The total number of operations in each iteration is proportional to $O(|V| + |E|)$.

The reader is encouraged to analyze a variation of this algorithm, where we perform repeated pointer jumping in step 3, so as to convert a tree into a star before we proceed to the next iteration.

Exercise 12.9 *Compare the two variants of the connectivity algorithm.*

12.5 Basic Geometric algorithms

The Quickhull algorithm described in section 6.5 is a good candidate for a parallel algorithm as most of the operations can be done simultaneously. These are $O(1)$ time

Procedure Parallel Graph connectivity(G)

```
1 Initialize For all $v \in V$, we set $p(v) = v$ ;
2 while not all vertices of a connected component are not in star do
3     for $(u, v) \in E$ do
4         if Isroot(p(u)) and (p(u) < p(v)) then
5             hook $(p(u), p(v))$
6         if IsStar (r) then
7             hook((p(v), p(u)) ;
8     for $v \in V$ do
9         $p(v) \leftarrow p(p(v))$ ;
```

Function IsStar(w)

```
1 This is run on all the vertices $v \in V$ ;
2 if $p(w) \neq p(p(w))$ then
3     the tree containing $w$ is not a star ;
  Return the tag of $v$ ;
```

Procedure hook(u, w)

```
1 $(u, w)$ is an edge ;
2 if IsStar ( u) then
3     $p(p(w)) \leftarrow p(p(u))$
```

Function IsRoot(v)

```
1 if $p(v) = v$ then
2     true
  else
      false
```

Figure 12.7: Parallel Connectivity

$left - turn$ tests involving the sign of a 3×3 determinant, based on which some of the points are elminated from further consideration. The subproblems are no more than $\frac{3}{4}$ of the original problem implying that there are $O(\log n)$ levels of recursion. The number of operations in each level is proportional to $O(n)$ and so if each level of recursion can be done in $t(n)$ parallel time, the total time will be $O(t(n) \cdot \log n)$. If $t(n)$ is $O(1)$, it would lead to an $O(\log n)$ time parallel algorithm which is often regarded as the best possible because of a number of related lower-bounds. Although the $left - turn$ tests can be done in $O(1)$ steps, the partitioning of the point sets into contiguous locations in an array is difficult to achieve in $O(1)$ time. Without this, the we will not be able to apply the algorithm recursively that works with the points in contigious locations. We know that compaction can be done in $O(\log n)$ time using prefix computation, so we will settle for an $O(\log^2 n)$ time parallel algorithm.

The number of processors is $O(n/\log n)$, that will enable us to do $O(n)$ $left - turn$ tests in $O(\log n)$ time. Unlike the (sequential) Quickhull algorithm, the analysis is not sensitive to the output size. For this, we will relate the parallel running time with the sequential bounds to obtain an improvement of the following kind.

Theorem 12.1 *There is a parallel algorithm to construct a planar convex hull in* $O(\log^2 n \cdot \log h)$ *parallel time and total work* $O(n \log h)$ *where* n *and* h *are the input and output sizes respectively.*

We will describe a very general technique for load distribution in a parallel algorithm. Suppose there are T parallel phases in an algorithm where there is no dependence between operations carried out within a phase. If there are p processors available, then by sharing the tasks equally among them the m_i, $1 \leq i \leq T$, tasks in phase i can be completed in time $O(\lceil \frac{m_i}{p} \rceil)$. So the total parallel time is given by $\sum_i^T O(\lceil \frac{m_i}{p} \rceil) = O(T) + O(\frac{\sum_i m_i}{p})$.

To this we also need to add the time for load balancing based on prefix computation, namely, $O(\frac{m_i}{p})$ for phase i as long as $m_i \geq p \log p$. So, this implies that each of the $O(\log n)$ phases require $\Omega(\log p)$ steps since $m_i/p \geq \log p$. So, we can state the result as follows

Lemma 12.7 (Load balancing) *In any parallel algorithm that has* T *parallel phases with* m_i *operations in phase* i*, the algorithm can be executed in* $O(T \log p + \frac{\sum_i m_i}{p})$ *parallel steps using* p *processors.*

Let us apply the previous result in the context of the Quickhull algorithm. There are $\log n$ parallel phases and in each phase there are at most n operations as the points belonging to the different subproblems are disjoint. From the analysis of the sequential algorithm, we know that $\sum_i m_i = O(n \log h)$ where h is the number of output points. Then an application of the above load balancing technique using p

processors will result in a running time of $O(\log n \cdot \log p) + O(\frac{n \log h}{p})$. Using $p \leq \frac{n}{\log^2 n}$ processors yields the required bound of Theorem 12.1.

Note that using $p = \frac{n \log h}{\log^2 n}$ would yield a superior time bound of $O(\log^2 n)$, but h being an unknown parameter, we cannot deploy these in advance.

12.6 Relation between parallel models

The PRAM model is clearly stronger than the *interconnection network* since all processors can access any data in $O(1)$ steps from the shared memory. More formally, any single step of an interconnection network can be simulated by the PRAM in one step. The converse is not true since data redistribution in the network could take time proportional to its diameter.

The simplest problem related to redistribution of data is called 1-1 *permutation routing*. Here every processor is a source and a destination of exactly one data item. The ideal goal is to achieve this routing in time proportion to $\mathcal{D}$ which is the diameter. There are algorithms for routing in different architectures that achieve this bound.

One of the simplest algorithm is greedy where the data item is sent along the shortest route to its destination. A processor can send and receive one data item to/from each of its neighbors in one step.

Exercise 12.10 *Show that in a linear array of n processors, permutation routing can be done in n steps.*

If a processor has multiple data items to be sent to any specific neighbor then only data item is transmitted while the rest must wait in a queue. In any routing strategy, the maximum queue length must have an upper bound for scalability. In the case of linear array, the queue length can be bounded by a constant.

To simulate a PRAM algorithm on interconnection network, one needs to go beyond permutation routing. More specifically, one must be able to simulate concurrent read and concurrent write. There is a rich body of literature that describes emulation of PRAM algorithms on low diameter networks like hypercubes and butterfly network that takes $O(\log n)$ time using constant size queues. This implies that PRAM algorithms can run on interconnection networks incurring no more than a logarithmic slowdown.

12.6.1 Routing on a mesh

Consider an $n \times n$ mesh of n^2 processors whose diameter is $2n$. Let a processor be identified by (i, j) where i is the row number and j is the column number. Let the destination of a data packet starting from (i, j) be denoted by (i', j').

A routing strategy is defined by

(i) Path selection
(ii) Priority scheme between packets that contend for the same link.
(iii) Maximum queue size in any node.

In the case of a linear array, path selection is a greedy choice which is unique and priority is redundant since there were never two packets trying to move along a link in the same direction (we assume that links are bidirectional allowing two packets to move simultaneously in opposite direction on the same link). There is no queue build up during the routing process.

However, if we change the initial and final conditions by allowing more than one packet to start from the same processor, then a priority order has to be defined between contending packets. Suppose we have n_i packets in the i-th processor, $1 \leq i \leq n$, and $\sum_i n_i \leq cn$ for some constant c. A natural priority ordering is defined as *furthest destination first*. Let n'_i denote the number of packets that have destination in the i-th node. Clearly $\sum_i n_i = \sum_i n'_i$ and let $m = \max_i\{n_i\}$ and $m' = \max_i\{n'_i\}$. The greedy routing achieves a routing time of cn steps using a queue size that is $\max\{mm'\}$.

Here is an analysis using the *furthest destination first* priority scheme. For a packet starting from the i-th processor, with destination $j \quad j > i$, it can be delayed by packets with destination in $[j+1, n]$. If $n'_i = 1$ for all i, then there are exactly $n - j - 1$ such packets so the packet will reach with $n - j - 1 + (j - i) = n - i - 1$ steps that is bounded by $n - 1$. Note that once a packet starts moving it can never be delayed further by any other packet. This can be easily argued starting from the rightmost moving packet and the next packet which can be delayed at most once and so on. When n'_i can exceed 1, then a packet can get delayed by $\sum_{i=j}^{i=n} n'_i$. The queue sizes can only increase when a packet reaches its destination.

Using the previous strategy, we will etend it to routing on a mesh. Let us use a path such that a packet reached the correct column and then it goes to the destination row. If we allow unbounded queue size then it can be easily done using two phases of one-dimensional routing, requiring a maximum of $2n$ steps. However the queue sizes could become as large as n. (Think about a bad routing instance that achieves this bound.) To avoid this situation, let us *distribute* the packets within the same column such that the packets that have to reach a specific column are distributed across different rows.

A simple way to achieve this is for every packet to choose a random intermediate destination within the same column. From our previous observations, this would take at most $n+m$ steps where m is the maximum number of packets in any (intermediate) destination. Subsequently, the time to route to the correct column will depend on the maximum number of packets that end up in any row. The third phase of routing

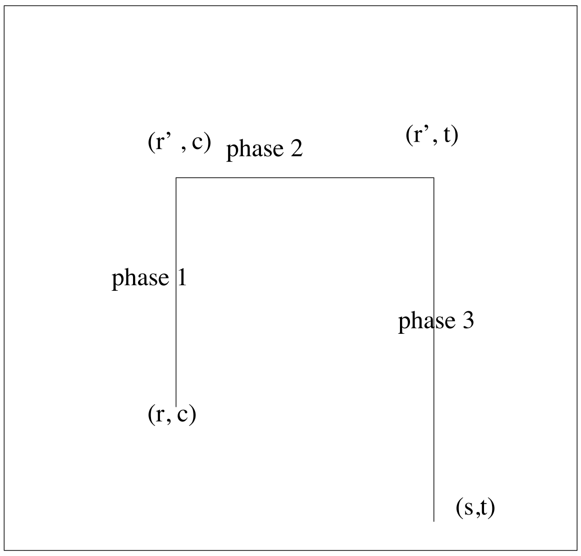

Figure 12.8: Starting from (r, c), the packet is routed a random row r' within the same column c. Subsequently it reaches the destination column t and finally the destination (s, t).

will take no more than n steps since every processor is a destination of exactly one packet. Figure ?? illusrtates the path taken by a packet in a three phase routing.

To analyse phases 1 and 2, we will get a bound on the expected number of packets that choose a given destination and the number of packets in a row that are destined for any specific column. Since the destinations are chosen uniformly at random, the probability that a specific processor P is chosen is $\frac{1}{n}$. Let P_i be a 0-1 random variable which is 1 if processor P is chosen by the data packet i and 0 otherwise. Then the number of packets that will end up in P is also a random variable $X = \sum_{i=1}^{i=n} P_i$. So,

$$\mathbb{E}[X] = \mathbb{E}[\sum_i P_i] = \sum_i \Pr[P_i = 1] = 1$$

Using a similar calculation, the expected number of packets in a row i after phase 1, that have column C as final destination is $\sum_j n_j \times \mathbb{E}[X_{j,C}]$. Here n_j is the number of such packets in column j and $X_{j,C}$ is the number of packets in processor (i, j) that started from column j and picked row i as the random intermediate destination and that. All these packets have column C as the destination in phase 2. Along the lines of our previous calculation $\mathbb{E}[X_{j,C}] = \frac{n_j}{n}$, so [9] the expected number of packets in row i destined for column C, denoted by $Y_{i,C}$ is bounded by $1/n \sum_j n_j = 1$ since $\sum_j n_j = n$.

[9]sum of n_j 0-1 variables each with expectation $1/n$

Let Y_i be the total number of packets in row i (over all destination columns), then using similar calculations $\mathbb{E}[Y_i] = n$.

If the random destinations are chosen independently, then the random variables are Binomially distributed (sum of independent Bernoulli trials) so using Chernoff bounds, it follows that

$$\Pr[X \geq \Omega(\log n / \log \log n)] \leq \frac{1}{n^2} \qquad \Pr[Y_i \geq n + \Omega(\sqrt{n \log n})] \leq \frac{1}{n^2}$$

Note that the random variables X and $Y_{i,C}$ are the bounds on the queue sizes at the end of phase 1 and phase 2 respectively and have a similar distribution. Using the union bound, all the queue sizes can be bounded by $\log n / \log \log n$ in phase 1 and phase 2. Moreover the routing time in phase 2, can be bounded by $n + O(\sqrt{n \log n})$ in phase 2. Thus the total routing time over all the 3 phases can be bound by $3n + O(\sqrt{n \log n})$ using $O(\log n / \log \log n)$ sized queues.

The above bound can be improved to $2n + o(n)$ routing time and $O(1)$ queue size by using more sophisticated analysis and overlapping phases 2 and 3, i.e., a packet begins its phase 3 as soon as it completes phase 3, rather than wait for all the other packets to complete phase 2.

Chapter 13

Memory hierarchy and caching

13.1 Models of memory hierarchy

Designing memory architecture is an important component of computer organization that tries to achieve a balance between computational speed and memory speed, viz., the time to fetch operands from the memory. The computational speeds are much faster since it happens within the chip whereas a memory access could involve off chip memory units. To offset this disparity, the modern computer has several layers of memory, called cache memory that provide faster access to the operands. Because of technological and cost limitations, the cache memories offer a range of speed-cost tradeoffs. For example the L1 cache ,the fastest level is usually also of the smallest size. The L2 cache is larger, say by a factor of ten but also considerably slower. The secondary memory which is the disk is largest but could be 10,000 times slower than the L1 cache. For any large size application most of the data resides on disk and transferred to the faster levels of cache when required.[1]

This movement of data is usually beyond the control of the normal programmer and managed by the operating system and hardware. By using empirical principles called *temporal* and *spatial* locality of memory access, several replacement policies are used to maximize the chances of keeping the operands in the faster cache memory levels. However, it must be obvious that there will be occasions that the required operand is not present in L1, so one has to reach out to L2 and beyond and pay the penalty of higher access cost. In other words memory access cost is not uniform as discussed in the beginning of this book but for simplicity of analysis, we pretend that it remains same.

In this chapter we will do away with this assumption; however for simpler exposition, we will deal with only two levels of memory *slow* and *fast* where the slower

[1]We are ignoring a predictive technique called prefetching here.

memory has infinite size while the slower one is limited, say, size M and significantly faster. Consequently we can pretend that the faster memory has zero (negligible) cost and the slower memory has cost 1. For any computation, the operands must reside inside the cache. If they are not present in the cache, they must be fetched from the slower memory, paying a unit cost (scaled appropriately). To offset this transfer cost, we transfer a contiguous chunk of B memory locations. This applies to both read and writes to the slower memory. This model is known as the *External memory model* with parameters M, B and will be denoted by $\mathcal{C}(M, B)$. Note that $M \geq B$ and in most practical situations $M \geq B^2$.

We will assume that the algorithm designer can use the parameters M, B to design appropriate algorithms to achieve higher efficiency in $\mathcal{C}(M, B)$. Later we will discuss that even without the explicit use of M, B one can design efficient algorithms, called *cache oblivious* algorithms. To focus better on the memory management issues, we will not account for the computational cost and only try to minimize memory transfers between cache and secondary memory. We will also assume appropriate instructions available to transfer a specific block block from the secondary memory to the cache. If there is no room in the cache then we have to replace an existing block in the cache and we can choose the block to be evicted.[2] A very simple situation is to add n elements stored as n/B memory blocks where initially they are all in the secondary memory. Clearly, we will encounter at least n/B memory transfers just to read all the elements.

We plan to study and develop techniques for designing efficient algorithms for some fundamental problems in this two level memory model and highlight issues that are ignored in the conventional algorithms.

13.2 Transposing a matrix

Consider an $p \times q$ matrix A that we want to transpose and store in another $q \times p$ matrix A'. initially this matrix is stored in the slower secondary memory and arranged in a *row-major* pattern. Since the memory is laid out in a linear array, a *row-major* format stores all the elements of first row, followed by the second row elements and so on. The *column major* layout stores the elements of column 1 followed by column 2 and so on. Therefore computing $A' = A^T$ involves changing the layouts from row-major to column-major.

The straightforward algorithm for transpose involves moving an element $a_{i,j} \in A$ to $b_{j,i} \in A'$ for all i, j. In the $\mathcal{C}(M, B)$ model, we would like to accomplish this for B elements simultaneously since we always transfer B elements at a time. If the

[2]This control is usually not available to the programmers in a user mode and left to the operating system responsible for memory management.

Procedure Computing transpose efficiently in for matrix $A(p, q)$

```
1 Input A is a $p \times q$ matrix in row-major layout in external memory ;
2 for $i$ = 1 to $p/B$ do
3     for $j$ = 1 to $q/B$ do
4         Transfer $A_{t(i,j)}$ to the cache memory $\mathcal{C}$ ;
5         Compute the transpose $A^T_{t(i,j)}$ within $\mathcal{C}$ in a a conventional element-wise
            manner ;
6         Transfer to $A'_{t(j,i)}$
7 $A'$ contains the transpose of $A$ in the external memory ;
```

Function Transfer($D_{t(k,l)}, r, s$)

```
1 Input transfer a $B \times B$ submatrix located at $i \cdot B - 1, j \cdot B - 1$ of an $r \times s$
    matrix to cache memory ;
2 for $i$ = 1 to $B$ do
3     move block starting at $(k \cdot B + i) \cdot r + B \cdot l$ into the $i$-th block in $\mathcal{C}$ ;
4 Comment A similar procedure is used to transfer from $\mathcal{C}$ to the external
    memory ;
```

Figure 13.1: Transposing a matrix using minimal transfers

matrices are laid out in row-major form, then we fetch B elements from the same row (assuming that p, q are multiples of B), but they must be in different columns so then cannot be transferred simultaneously. So these will take B transfers. This is clearly an inefficient scheme, but it is not difficult to improve it with a little thought. Partition the matrix into $B \times B$ submatrices (see Figure 13.2) and denote these by $A_{t(a,b)}$ $1 \leq a \leq p/B$ $1 \leq b \leq q/B$ for matrix A. These submatrices define a *tiling* of the matrix A and the respective tiles for A' are denoted by $A'_{t(a,b)}$ $1 \leq a \leq q/B$ $1 \leq b \leq q/B$.

Figure 13.1 describes an alternate procedure for computing $B = A^T$.

The algorithm described in Figure 13.1 makes $O(pq/B)$ block transfers which is clearly optimal.

13.3 Sorting in external memory

For sorting n elements in $\mathcal{C}(M, B)$, we will focus on the number of data movement steps so as to minimize the number of transfers between cache and external memory. Unlike the traditional sorting algorithms, the number of comparisons is not accounted

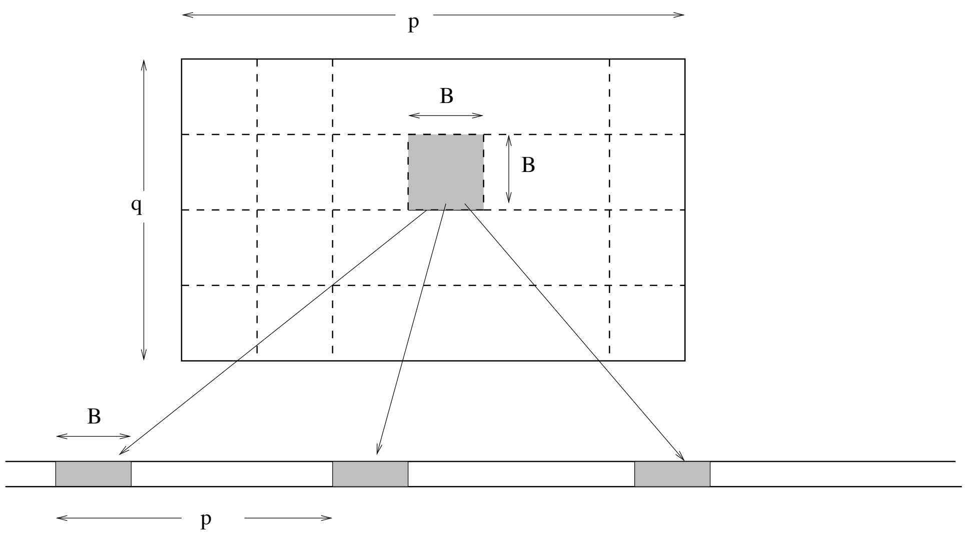

Figure 13.2: The tiling of a matrix in a row major layout.

in this model. We will adapt mergesort to this model by choosing a larger degree of merge.

Exercise 13.1 *Instead of partitioning the input into two almost-equal halves, sorting them recursively and doing a binary merge, if we partition into $k \geq 2$ parts and do a k-ary merge, show that the number of comparisons remain unchanged.*

We will choose an appropriate value of k so that the recursive depth of the algorithm reduces. Note that for each level of the mergesort algorithm, we make a pass through the entire data. For the convention binary mergesort, we have to make $\log n$ passes through the data while merging sorted sequences of lengths $n/2^i$ for level i. When we merge k sorted sequences, it suffices to keep the leading (smallest) blocks of each sequence in the cache memory and chose the smallest element among them for the next output. The number of passes required is $\log_k n$. To economize memory transfer, we want to read and write contiguous chunks of B elements, so we write only after B elements are output. Note that the smallest B elements must occur among the leading blocks (smallest B elements) of the sorted sequence. Since all the $k+1$ sequences including the k input and 1 output sequence must be within the cache, the largest value value of k is $O(M/B)$. We need some extra space to store the data structure for merging (a k-ary min-heap) but we will not discuss any details of this implementation since it can be done using any conventional approach within the cache memory. So we can assume that $k = \frac{M}{cB}$ for some appropriate constant $c > 1$.

We shall first analyze the number of memory block transfers does it take to merge k sorted sequences of lengths ℓ each. As previously discussed, we maintain the leading block of each sequence in the cache memory and fetch the next block, after this is exhausted. So we need $\ell/B = \ell'$ block transfers for each sequence which may be thought of as the number of blocks in the sequence (if it is not a multiple of B, then we count the partial block as an extra block). Likewise the output is written out as blocks and this must be the sum of all input sequences which is $\frac{k}{\ell}'$. In other words, the number of block transfers for merging is proportional to the sum of the sequences being merged.

In any level of k-way mergesort, all sequences are counted exactly once as they participate in exactly one merge and so the total cost is n/B. For $k = \Omega(M/B)$, there are $\log_{M/B}(n/B)$ levels of recursion as the smallest size of a sequence is at least B. So the total number of block transfers is $O(\frac{n}{B}\log_{M/B}(n/B))$ for sorting n elements in $\mathcal{C}(M, B)$.

Recall that this is only the number of memory block transfers - the number of comparisons remains $O(n \log n)$ like conventional mergesort. For $M > B^2$, note that $\log_{M/B}(n/B) = O(\log_M(n/B)$.

Exercise 13.2 *For $M = O(B)$, what is the I-O complexity (number of block transfers) to transpose an $n \times n$ matrix ?*

Exercise 13.3 *The FFT computation based on the butterfly network in Figure 7.1 is a very important problem. Show how to accomplish this in $O(\frac{n}{B}\log_{M/B}(n/B))$ I-O's in $\mathcal{C}(M, B)$.*
Hint: Partition the computation into FFT sub-networks of size M.

13.3.1 Can we improve the algorithm

We will develop some formal arguments to obtain a lower bound for the problem of permuting n elements where any rearrangement have to be routed through a buffer of bounded size, in this case, M. Any lower bound on permutation is also applicable to sorting since permutation can be done by sorting on the destination index of the elements. If $\pi(i) = j$, then one can sort on $j's$ where $\pi()$ is the permutation function.

We will make some assumptions to simplify the arguments for the lower-bound. These assumptions can be removed with some loss of constant factors in the final bound. There will be exactly one copy of any element, viz., when the element is fetched from slower memory then there is no copy left in the slower memory. Likewise, when an element is stored in the slower memory then there is no copy in the cache. With a little thought, the reader can convince himself that maintaining multiple copies in a permutation algorithm is of no use since the final output has only one copy that can be traced backwards as the relevant copy.

The proof is based on a simple counting argument on how many orderings are possible after t block transfers. For a worst-case bound, the number of possible orderings must be at least $n!$ for n elements. We do not insist that the elements must be in contiguous locations. If $\pi(i) > \pi(j)$ then $R_i > R_j$, where R_i is the final location of the i-th element for all pairs i, j.

A typical algorithm has the following behavior.

1. Fetch a block from the slow memory into the cache.
2. Perform computation within cache to facilitate the permutation.
3. Write out a block form the cache to the slower memory.

Note that Step 2 does not require block transfers and is *free* since we are not counting operations within the cache. So we would like to count the additional orderings generated by Steps 1-3.

Once a block of B elements is read into the cache, it can induce additional orderings with respect to the $M - B$ elements present in the cache. This number is $\frac{M!}{B! \cdot (M-B)!} = \binom{M}{B}$ which is the relative orderings between $M - B$ and B elements. Further, if these B elements were not written out before, i.e., these were never present in cache before then there are $B!$ ordering possible among them. (If the block was written out in a previous step, then they were in cache together then these orderings would have been already accounted for.) So this can happen at most n/B times, viz., only for the initial input blocks.

In Step 3, during the t-th output, there are at most $n/B + t$ places relative to the existing blocks. There were n/B blocks to begin with and $t - 1$ previously written blocks, so the t-th block can be written out in $n/B + t$ intervals relative to the other blocks. Note that there may be arbitrary gaps between blocks as long as the relative ordering is achieved.

From the previous arguments, we can bound the number of attainable orderings after t memory transfers by

$$(B!)^{n/B} \cdot \prod_{i=0}^{i=t-1} (n/B + i) \cdot \binom{M}{B}$$

If T is the worst case bound on number of block transfers, then

$$\begin{aligned}(B!)^{n/B} \cdot \prod_{i=1}^{i=T} (n/B + i) \cdot \binom{M}{B} &\leq (B!)^{n/B} \cdot (n/B + T)! \cdot \binom{M}{B}^T \\ &\leq B^n \cdot (n/B + T)^{n/B+T} \cdot (M/B)^{BT}\end{aligned}$$

using Stirling's approximation $n! \sim (n/e)^n$ and $\binom{n}{k} \leq (en/k)^k$.

From the last inequality it follows that

$$B^n \cdot (n/B + T)^{n/B+T} \cdot (M/B)^{BT} \geq n! \geq (n/e)^n$$

Taking logarithm on both sides and rearranging, we obtain

$$BT\log(M/B) + (T + n/B)\cdot\log(n/B + T) \geq n\log n - n\log B = n\log(n/B) \quad (13.3.1)$$

We know that $(n/B)\log(n/B) \geq T \geq (n/B)$,[3] so $(T + n/B)\log(n/B + T) \leq 4T\log(n/B)$ and we can rewrite the inequality above as

$$T\left(B\log(M/B) + 4\log(n/B)\right) \geq n\log(n/B)$$

For $4\log(n/B) \leq B\log(M/B)$, we obtain $T = \Omega(\frac{n}{B}\log_{M/B}(n/B))$. For $\log(n/B) > B\log(M/B)$, we obtain $T = \Omega(n\frac{\log(n/B)}{\log(n/B)} = \Omega(n)$.

Theorem 13.1 *Any algorithm that permutes n elements in $\mathcal{C}(M.B)$ uses $\Omega(\frac{n}{B}\cdot\log_{M/B}(n/B))$ block transfers in the worst case.*

Exercise 13.4 *Show that the average case lower bound for permutation is asymptotically similar to the worst-case bound.*

As a consequence of the Theorem 13.1, the lower bound for sorting matches the bound for the mergesort algorithm and hence the algorithm cannot be improved in asymptotic complexity.

13.4 Cache oblivious design

Consider the problem of searching a large static dictionary in the external memory of n elements. If we use B-tree type data structure, then we can easily search using $O(\log_B n)$ memory transfers. This is can be explained by effectively end up doing a B-way search. Each node of the B-tree contains B records that we can fetch using a single block transfer.

Building a B-tree requires the knowledge of B. Since we are dealing with a static dictionary, we can consider doing a straightforward B-ary search in a sorted data set. Still, it requires the knowledge of B. What if the programmer is not allowed to use the parameter B ? Consider the following alternative of doing a $\sqrt{n}$-ary search presented in Figure 13.3.

The analysis of this algorithm in $\mathcal{C}(M, B)$ depends crucially on the elements being in contiguous locations. Although S is initially contiguous, S' is not, so the indexing of S has to be done carefully in a recursive fashion. The elements of S' must be indexed before the elements of $S - S'$ and the indexing of each of the $\sqrt{n}$ subsets of

[3]From the previous bound on mergesort

```
Procedure Search($x, S$)
1 Input A sorted set $S = \{x_1, x_2 \ldots x_n\}$ ;
  if $|S| = 1$ then
      return Yes or No according to whether $x \in S$
  else
2     Let $S' = \{x_{i\sqrt{n}}\}$ be a subsequence consisting of every $\sqrt{n}$-th element of $S$. ;
3     Search $(x, S')$ ;
      Let $p, q \in S'$ where $p \leq x < q$ ;
4     Return Search $(x, S \cap [p, q])$ - search the relevant interval of $S'$ ;
```

Figure 13.3: Searching in a dictionary in external memory

$S - S'$ will also be indexed recursively. Figure **??** shows the numbering of a set of 15 elements.

The number of memory transfers $T(n)$ for searching satisfies the following recurrence

$$T(n) = T(\sqrt{n}) + T(\sqrt{n}) \quad T(k) = O(1) \quad \text{for} \quad k \leq B$$

since there are two calls to subproblems of size $\sqrt{n}$. This yields $T(n) = O(\log_B n)$. Note that although the algorithm did not rely on the knowledge of B, the recurrence made effective use of B, since searching within contiguous B elements requires one memory block transfer (and at most two transfers if the memory transfers are not aligned with block boundaries). After the block resides within the cache, no further memory transfers are required although the recursive calls continue till the terminating condition is satisfied.

13.4.1 Oblivious matrix transpose

In this section, we assume that $M = \Omega(B^2)$ which is referred to as *tall cache* assumption. Given any $m \times n$ matrix A, we use a recursive approach for transposing it into an $n \times m$ matrix $B = A^T$.

$$\begin{bmatrix} A_1^{m\times n/2} A_2^{m\times n/2} \end{bmatrix} \Rightarrow \begin{bmatrix} B_1^{n/2\times m} \\ B_2^{n/2\times m} \end{bmatrix} \quad \text{where } n \geq m \text{ and } B_i = A_i^T$$

$$\begin{bmatrix} A_1'^{m/2\times n} A_2'^{m/2\times n} \end{bmatrix} \Rightarrow \begin{bmatrix} B_1'^{n\times m/2} \\ B_2'^{n\times m/2} \end{bmatrix} \quad \text{where } m \geq n \text{ and } B_i' = A_i'^T$$

```
Procedure Transpose( $A, B$)
1 Input $A$ is an $m \times n$ matrix ;
2 if $\max\{m, n\} \leq c$ then
3     perform transpose by swapping elements
4 if $n \geq m$ then
5     Transpose ( $A_1, B_1$) ; Transpose ($A_2, B_2$)
  else
6     Transpose ( $A'_1, B'_1$); Transpose ($A'_2, B'_2$)
```

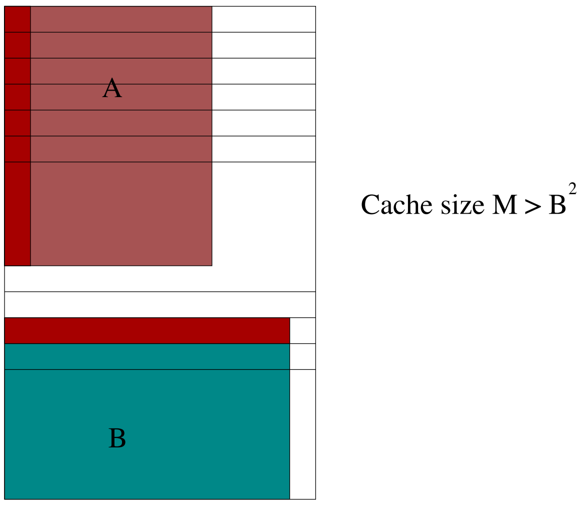

Figure 13.4: Base case: Both A,B fit into cache - no further cache miss

The formal algorithm based on the previous recurrence is described in Figure Transpose .

When $m, n \leq B/4$, then there are no more cache misses, since each row (column) can occupy at most two cache lines. The algorithm actually starts moving elements from external memory when the recursion terminates at size $c \ll B$. Starting from that stage, untill $m, n \leq B/4$, there are no more cache misses since there is enough space for submatrices of size $B/4 \times B/4$. The other cases of the recurrence addresses the recursive cases corresponding to splitting across columns or rows - whichever is larger. Therefore, the number of memory block transfers $Q(m, n)$ for an $m \times n$ matrix satisfies the following recurrence.

$$Q(m,n) \leq \begin{cases} 4m & n \leq m \leq B/4 \text{ in cache} \\ 4n & m \leq n \leq B/4 \text{ in cache} \\ 2Q(m, \lceil n/2 \rceil) & m \leq n \\ 2Q(\lceil m/2 \rceil, n) & n \leq m \end{cases}$$

Exercise 13.5 *Show that $Q(m,n) \leq O(mn/B)$ from the above recurrence. You may want to rewrite the base cases to simplify the calculations.*

When the matrix has less than B^2 elements ($m \leq n \leq B$ or $n \leq m \leq B$), the recursive algorithm brings all the required blocks - a maximum of B, transposes them within the cache and writes them out. All this happens without the explicit knowledge of the parameters M, B but requires support from the memory management policy. In particular, the recurrence is valid for the *Least Recently Used* (LRU) policy. Since the algorithm is parameter oblivious, there is no explicit control on the blocks to be replaced and hence its inherent dependence on the replacement policy. The good news is that the LRU poicy is known to be competitive with respect to the ideal *optimal* replacement policy

Theorem 13.2 *Suppose OPT is the number of cache misses incurred by an optimal algorithm on an arbitrary sequence of length m with cache size p. Then the number of misses incurred by the LRU policy on the same sequence with cache size $k \geq p$ can be bounded by $\frac{k}{k-p} \cdot m$*[4]*.*

It follows that for $k = 2p$, the number of cache misses incurred LRU is within a factor two of the optimal replacement.

We can pretend that the available memory is $M/2$ which preserves all the previous asymptotic calculations. The number of cache misses by the LRU policy will be within a factor two of this bound. Theorem 13.2 is a well-known result in the area of *competitive algorithms* which somewhat out of scope of the discussion here but we will present a proof of the theorem.

Consider a sequence of m requests $\sigma_i \in \{1, 2 \ldots N\}$ which can be thought of as the set of cache lines. We further divide this sequence into subsequences s_1, s_2 such that every subsequence has $k+1$ distinct requests from $\{1, 2 \ldots N\}$ and the subsequence is of minimal length, viz., it ends the first time when we encounter the $k+1$-st distinct request without including this request. The LRU policy will incur at most k misses in each subsequence. Now consider any policy (including the optimal policy) that has cache size p where $k > p$. In each phase, it will incur at least $k - p$ misses since

[4]A more precise ratio is $k/(k-p+1)$.

$$\sigma_{i_1}\sigma_{i_1+1}\ldots\sigma_{i_1+r_1}\Big|\sigma_{\mathbf{i_2}}\sigma_{i_2+1}\ldots\sigma_{i_2+r_2}\Big|\sigma_{\mathbf{i_3}}\sigma_{i_3+1}\ldots\sigma_{i_3+r_3}\Big|\ldots\Big|\sigma_{\mathbf{i_t}}\sigma_{i_t+1}\ldots$$

Figure 13.5: The subsequence $\sigma_{i_1}\sigma_{i_1+1}\ldots\sigma_{i_1+r_1}\sigma_{\mathbf{i_2}}$ have $k+1$ distinct elements whereas the subsequence $\sigma_{i_1}\sigma_{i_1+1}\ldots\sigma_{i_1+r_1}$ have k distinct elements.

it has to evict at least that many items to handle k distinct requests. Here we are assuming that out of the k distinct requests, there are p cache lines from the previous phase and it cannot be any better. In the first phase, both policies will incur the same number of misses (starting from an empty cache).

Let f^i_{LRU} denote the number of cache misses incurred by LRU policy in subsequence i and f^i_{OPT} denote the number of cache misses by the optimal policy. Then $\sum_{i=1}^t f^i_{LRU} \leq (t-1)\cdot k$ and $\sum_{i=1}^t f^i_{OPT} \geq (p-k)\cdot(t-1)+k$. Their ratio is bounded by

$$\frac{\sum_{i=1}^t f^i_{LRU}}{\sum_{i=1}^t f^i_{OPT}} \leq \frac{(t-1)\cdot k + k}{(t-1)\cdot(p-k)+k} \leq \frac{(t-1)\cdot k}{t-1)\cdot(k-p)} = \frac{k}{k-p}$$

Chapter 14

Streaming Data Model

14.1 Introduction

In this chapter, we consider a new model of computation where the data arrives a very long sequence of elements. Such a setting has become increasingly important in scenarios where we need to handle huge amount of data and do not have space to store all of it, or do not have time to scan the data multiple times. As an example, consider the amount of traffic encountered by a network router – it sees millions of packets every second. We may want to compute some properties of the data seen by the router, for example, the most frequent (or the top ten) destinations. In such a setting, we cannot expect the router to store details about each of the packet – this would require terabytes of storage capacity, and even if we could store all this data, answering queries on them will take too much time. Similar problems arise in the case of analyzing web-traffic, data generated by large sensor networks, etc.

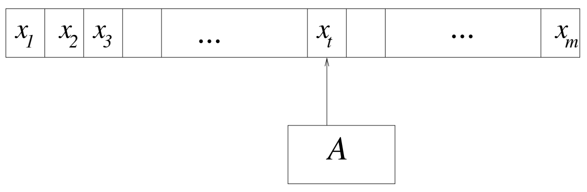

Figure 14.1: The algorithm A receives input x_t at time t, but has limited space.

In the data streaming model, we assume that the data is arrive as a long stream $x_1, x_2, \ldots, x_m$, where the algorithm receives the element x_i at step i (see Figure 14.1). Further we assume that the elements belong to a universe $U = \{e_1, \ldots, e_n\}$. Note that the stream can have the same element repeated multiple times[1]. Both the quantities m and n are assumed to be very large, and we would like our algorithms to take sub-linear space (sometimes, even logarithmic space). This implies that the classical approach where we store all the data and can access any element of the data (say, in the RAM model) is no longer valid here because we are not allowed to store all of the data. This also means that we may not be able to answer many of the queries exactly. Consider for example the following query – output the most frequent element in the stream. Now consider a scenario where each element arrives just once, but there is one exceptional element which arrives twice. Unless we store all the distinct elements in the stream, identifying this exceptional element seems impossible. Therefore, it is natural to make some more assumptions about the nature of output expected from an algorithm. For example, here we would expect the algorithm to work only if there is some element which occurs much more often than other elements. This is an assumption about the nature of the data seen by the algorithms. At other times, we would allow the algorithm to output approximate answers. For example, consider the problem of finding the number of distinct elements in a stream. In most practical settings, we would be happy with an answer which is a small constant factor away from the actual answer.

In this chapter, we consider some of the most fundamental problems studied in the streaming data model. Many of these algorithms will be randomized. In other words, they will output the correct (or approximate) answer with high probability.

14.2 Finding frequent elements in stream

In this section, we consider the problem of finding *frequent* elements in a stream. As indicated in above, this happens to be a very useful statistic for many applications. The notion of frequent elements can be defined in many ways:

- **Mode** : The element (or elements) with the highest frequency.
- **Majority**: An element with more than 50% occurrence – note that there may not be any element.
- **Threshold**: Find out all elements that occur more than f fraction of the length of the stream, for any $0 < f \leq 1$. Finding majority is a special case with $f = 1/2$.

[1]There are more general models which allow both insertion and deletion of an element. We will not discuss these models in this chapter, though some of the algorithms discussed in this chapter extend to this more general setting as well.

Procedure Finding Majority of n elements in Array a

```
1 $count \leftarrow 0$ ;
2 for $i$ = 1 to $n$ do
3     if $count = 0$ then
          $maj \leftarrow a[i]$  (* initalize $maj$ *)
4     if $maj = a[i]$ then
5         $count \leftarrow count + 1$
      else
6         $count \leftarrow count - 1$ ;
7 Return $maj$ ;
```

Figure 14.2: Boyer-Moore Majority Voting Algorithm

Observe that the above problems are hardly interesting from the classical algorithmic design perspective because they can be easily reduced to sorting. Designing more efficient algorithms requires more thought (for example, finding the mode). Accomplishing the same task in a streaming environment with limited memory presents interesting design challenges. Let us first review a well known algorithm for *Majority* finding among n elements known as the *Boyer-Moore Voting algorithm.* Recall that a majority element in a stream of length m is an element which occurs more than $m/2$ times in the stream. If no such element exists, the algorithm is allowed to output *any* element. This is always acceptable if we are allowed to scan the array once more because we can check if the element output by the algorithm is indeed the majority element. Therefore, we can safely assume that the array has a majority element.

The algorithm is described in Figure Finding Majority of n elements in Array a. The procedure scans the array sequentially [2] and maintains one counter variable. It also maintains another variable `maj` which stores the (guess for) majority element. Whenever the algorithm sees an element which identical to the one stored in `maj`, it increases the counter variable, otherwise it decreases it. If the counter reaches 0, it *resets* the variable `maj` to the next element. It is not obvious why it returns the majority element if it exists.

As mentioned above, we begin by assuming that there is a majority element, denoted by a. We need to show that when the algorithm stops, the variable `maj` is same as a. The algorithm tries to prune elements without affecting the majority. More formally, we will show that at the beginning each step t (i.e., before arrival of x_t), the algorithm maintains the following invariant: let S_t denote the *multi-set*

[2] Often we will think of the stream as a long array which can be scanned once only. In fact, there are more general models which allow the algorithm to make a few passes over the array.

consisting of the elements $x_t, x_{t+1}, \dots, x_m$ and `count` many copies of the element `maj`. We shall prove that for all time t, a will be the majority element of S_t. This statement suffices because at the end of the algorithm (when $t = m + 1$), S_t will be a multi-set consisting of several copies of the element `maj` only. The invariant shows that a will be the majority element of S_t, and so, must be same as the variable `maj`.

We prove this invariant by induction over t. Initially, S_t is same as the input sequence, and so, the statement follows by the definition of a. Suppose this fact is true at the beginning of step t. A key observation is that if there is a majority of a set of elements, it will remain a majority if some other element is deleted along with an instance of the majority element. Indeed if $m_1 > m/2$ then $m_1 - 1 > (m - 2)/2$. So, if x_t happens to be different from `maj`, we decrement count. This means that S_{t+1} is obtained from S_t be removing x_t and one copy of `maj`. Since these two elements are different, the observation above shows that the majority element does not change. So, a continues to be the majority element of S_{t+1}. The other case is when x_t happens to be same as `maj`. Here, the set S_{t+1} is same as S_t – we replace x_t be one more copy of `maj`. So the invariant holds trivially. This shows that the invariant holds at all time, and so, the algorithm outputs the majority element.

Another alternate argument is as follows. The total number of times we decrease the count variable is at most the number of times we increase it. Therefore, we can decrease it at most $m/2$ times. Since the majority variable appears more than $m/2$ times, it has to survive in the variable `maj` when the algorithm stops.

This idea can be generalized to finding out elements whose frequency is more than $\frac{n}{k}$ for any integer k. Observe that there can be at most $k - 1$ elements. So instead of one counter, we shall use $k - 1$ counters. When we scan the next element, we can either increment the count, if there exists a counter for the element or start a new counter if number of counters used is less than $k - 1$. Otherwise, we decrease the counts of all the existing counters. If any counter becomes zero, we discard that element and instead assign a counter for the new element. In the end the counters return the elements that have non-zero counts. As before, these are potentially the elements that have frequencies at least $\frac{n}{k}$ and we need a second pass to verify them.

The proof of correctness is along the same lines as the majority. Note that there can be at most $k - 1$ elements that have frequencies exceeding $\frac{n}{k}$, i.e., a fraction $\frac{1}{k}$. So, if we remove such an element along with $k - 1$ distinct elements, it still continues to be at least $\frac{1}{k}$ fraction of the remaining elements – $n_1 > \frac{n}{k} \Rightarrow n_1 - 1 > \frac{n-k}{k}$.

The previous algorithms have the property that the data is scanned in the order it is presented and the amount of space is proportional to the number of counters where each counter has $\log n$ bits. Thus the space requirement is logarithmic in the size of the input.

Procedure Algorithm for threshold

```
1 $cur$ : current element of stream ;
  $S$: current set of elements with non-zero counts, $|S| \leq k$ ;
2 if $cur \in S$ then
3     increment counter for $cur$
  else
      if $|S| < k$ then
4         Start a new counter for $cur$, update $S$
      else
5         decrement all counters ;
6         If a counter becomes 0 delete it from $S$
7 Return $a[maj\_ind]$ ;
```

Figure 14.3: Mishra-Gries streaming algorithm for frequent elements

14.3 Distinct elements in a stream

The challenging aspect of this problem is to count the number of distinct elements d in the input stream with limited memory s, where $s \ll d$. If we were allowed space comparable to d, then we could simply hash the elements and the count the number of non-zero buckets. Uniformly sampling a subset of elements from the stream could be misleading. Indeed, if some elements occur much more frequently than others, then multiple occurrence of such elements would be picked up by the the uniform sample and it doesn't provide any significant information about the number of distinct elements.

Instead we will hash the incoming elements uniformly over a range, $[1, n]$ such that if there are k distinct elements then they will be roughly n/k apart. If g is the gap between two consecutive hashed elements, then we can estimate $k = n/g$. Alternately, we can use the position of the first hashed position as as estimate of g. This is the underlying idea behind the algorithm given in Figure 14.4. The algorithm keeps track of the smallest value to which an element gets hashed (in the variable Z). Again, the idea is that if there are d distinct elements, then the elements get mapped to values in the array which are roughly p/d apart. So, the reciprocal of Z should be a good estimate of d/p.

This procedure will be analyzed rigorously using the property of universal hash family family discussed earlier. The parameter of interest will be the *expected* gap between consecutive hashed elements. Our strategy will be to prove that the Z lies between $k_1 p/d$ and $k_2 p/d$ with high probability, where k_1 and k_2 are two constants.

```
Procedure Finding the number of distinct elements in a stream $S(m,n)$
1 Input A stream $S = \{x_1, x_2 \ldots x_m\}$ where $x_i \in [1,n]$ ;
2 Suppose $p$ is a prime in the range $[n, 2n]$. Choose $0 \le a \le p-1$ and
    $0 \le b \le p-1$ uniformly at random ;
3 $Z \leftarrow \infty$ ;
4 for $i = 1$ to $m$ do
5     $Y = (a \cdot x_i + b) \mod p$ ;
6     if $Y < Z$ then
          $Z \leftarrow Y$
7 Return $\lceil \frac{p}{Z} \rceil$ ;
```

Figure 14.4: Counting number of distinct elements

It will then follow that the estimate p/Z is within constant factor of d.

Let $Z_i = (a \cdot x_i + b) \mod p$ be the sequence of hashed values from the stream. Then we can claim the following.

Claim 14.1 *The numbers Z_i , $1 \le i \le m$ are distributed uniformly at random in the range $[0, p-1]$ and are also pair-wise independent , viz., for i $\neq k$*

$$\Pr[Z_i = r, Z_k = s] = \Pr[Z_i = r] \cdot \Pr[Z_k = s] = \frac{1}{p^2}$$

Proof: For some fixed $i_0 \in [0, p-1]$ and $x \in [1,n]$, we want to find the probability that x is mapped to i_0. So

$$\begin{aligned} i_0 &\equiv (ax+b) \mod p \\ i_0 - b &\equiv ax \mod p \\ x^{-1}(i_0 - b) &\equiv a \mod p \end{aligned}$$

where x^{-1} is the multiplicative inverse of x in the multiplicative prime field modulo p and it is unique since p is prime[3]. For any fixed b, there is a unique solution for a. As a is chosen uniformly at random, the probability of this happening is $\frac{1}{p}$ for any *fixed* choice of b. Therefore this is also the unconditional probability that x is mapped to i_0.

For the second part consider $i_0 \neq i_1$. We can consider $x \neq y$ such that x, y are mapped respectively to i_0 and i_1. We can write the simultaneous equations similar to the previous one.

$$\begin{bmatrix} x & 1 \\ y & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix} \equiv_p \begin{bmatrix} i_0 \\ i_1 \end{bmatrix}$$

[3]By our choice of p, $x \not\equiv 0 \mod p$

The 2×2 matrix is invertible for $x \neq y$ and therefore there is a unique solution corresponding to a fixed choice of (i_0, i_1). The probability that a, b matches the solution is $\frac{1}{p^2}$ as they are chosen uniformly at random. $\square$

Recall that d denotes the number of distinct elements in the stream. We will show the following.

Claim 14.2 *For any constant $c \geq 2$,*

$$Z \in \left[\frac{p}{cd}, \frac{cp}{d}\right] \text{ with probability } \geq 1 - \frac{2}{c}$$

Proof: Note that if $Z = p/d$, then the algorithm returns d which is the number of distinct elements in the stream. Since Z is a random variable, we will only be able to bound the probability that it is within the interval $\left[\frac{p}{cd}, \frac{cp}{d}\right]$ with significant probability implying that the algorithm with return an answer in the range $[p/c, pc]$ with significant probability. Of course, there is a risk that it falls outside this window and that is the inherent nature of a *Monte Carlo* randomized algorithm.

First we will find the probability that $Z \leq s - 1$ for some arbitrary s. For sake of ease of notation, assume that the d distinct elements are $x_1, x_2, \ldots, x_d$. Let us define a family of indicator random variables in the following manner

$$X_i = \begin{cases} 1 & \text{if } (ax_i + b) \mod p \leq s - 1 \\ 0 & \text{otherwise} \end{cases}$$

So the total number of x_i that map to numbers in the range $[0, s-1]$ equals $\sum_{i=1}^{d} X_i$ (recall that we assumed that $x_1, \ldots, x_d$ are distinct). Let $X = \sum_{i=1}^{d} X_i$ and we therefore have

$$\mathbb{E}[X] = \mathbb{E}[\sum_i X_i] = \sum_i \mathbb{E}[X_i] = \sum_i \Pr[X_i = 1] = d \cdot \Pr[X_i = 1] = \frac{sd}{p}$$

The last equality follows from the previous result as there are s (viz., $0, 1 \ldots s-1$) possibilities for x_i to be mapped and each has probability $\frac{1}{p}$.

If we choose $s = \frac{p}{cd}$ for some constant c, then $\mathbb{E}[X] = 1/c$. From Markov's inequality, $\Pr[X \geq 1] \leq \frac{1}{c}$, implying that with probability greater than $1 - 1/c$ no x_i will be mapped to numbers in the range $[0, \lceil \frac{p}{cd} \rceil]$.

For the other direction, we will will Chebychev inequality, which requires computing the variance of X, which we shall denote by $\sigma^2(X)$. We know that

$$\sigma^2[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}^2[X]$$

Since $X = \sum_{i=1}^{d} X_i$, we can calculate (assume that all indices i and j vary from 1 to d)

$$\begin{aligned}
\mathbb{E}[X^2] &= \mathbb{E}[(\sum_{i=1} X_i)^2] \\
&= \mathbb{E}[\sum_{i=1} X_i^2 + \sum_{i \neq j} X_i \cdot X_j] \\
&= \mathbb{E}[\sum_{i} X_i^2] + \mathbb{E}[\sum_{i \neq j} X_i \cdot X_j] \\
&= \sum \mathbb{E}[X_i^2] + \sum_{i \neq j} \mathbb{E}[X_i] \cdot \mathbb{E}[X_j]
\end{aligned}$$

which follows from linearity of expectation and pairwise independence of X_i and X_j[4]. So the expression simplifies to $d \cdot \frac{s}{p} + d(d-1) \cdot \frac{s^2}{p^2}$. This yields the expression for

$$\sigma^2(X) = \frac{sd}{p} + \frac{d(d-1)s^2}{p^2} - \frac{s^2 d^2}{p^2} = \frac{sd}{p} \cdot (1 - \frac{s}{p} \leq \frac{sd}{p}$$

For $s = \frac{cp}{d}$, the variance is bounded by c. From Chebychev's inequality, we know that for any random variable X,

$$\Pr[|X - \mathbb{E}[X]| \geq t] \leq \frac{\sigma^2(X)}{t^2}$$

Using $t = \mathbb{E}[X] = \frac{sd}{p} = c$, we obtain $\Pr[|X - \mathbb{E}[X]| \geq \mathbb{E}[X]] \leq \frac{c}{c^2} = \frac{1}{c}$. The event $|X - \mathbb{E}[X]| \geq \mathbb{E}[X]$ is the union of two disjoint events, namely

(i) $X \geq 2\mathbb{E}[X]$ and
(ii) $\mathbb{E}[X] - X \geq \mathbb{E}[X]$, or $X \leq 0$

Clearly, both events must have probability bounded by $\frac{1}{c}$ and specifically, the second event implies that the probability that none of the N elements is mapped to the interval $[0, \frac{cp}{d}]$ is less than $\frac{1}{c}$. Using the union bound yields the required result. $\square$

So the algorithm outputs a number that is within the range $[\frac{d}{c}, cd]$ with probability $\geq 1 - \frac{2}{c}$.

[4]This needs to be rigorously proved from the previous result on pairwise independence of (X_i, X_j) being mapped to (i_0, i_1). We have to technically consider all pairs in the range $(1, s-1)$.

14.4 Frequency moment problem and applications

Suppose the set of elements in a stream $S = \{x_1, \ldots, x_m\}$ belong to a universe $U = \{e_1, \ldots, e_n\}$. Define the frequency f_i of element e_i as the number of occurrences of e_i in the stream S. The k^{th} frequency moment of the stream is defined as

$$F_k = \sum_{i=1}^{n} f_i^k.$$

Note that F_0 is exactly the number of distinct elements in the stream. F_1 counts the number of elements in the stream, and can be easily estimated by keeping a counter of size $O(\log m)$. The second frequency moment F_2 captures the non-uniformity in the data – if all n elements occur with equal frequency, i.e., m/n (assume that m is a multiple of n for the sake of this example), then F_2 is equal to m^2/n; whereas if the stream contains just one element (with frequency m), then F_2 is m^2. Thus, larger values of F_2 indicate non-uniformity in the stream. Higher frequency moments give similar statistics about the stream – as we increase k, we are putting more emphasis on higher frequency elements.

The idea behind estimating F_k is quite simple : suppose we sample an element uniformly at random from the stream, call it X. Suppose X happens to be the element e_i. Conditioned on this fact, X is equally likely to be any of the f_i occurrences of e_i. Now, we observe how many times e_i occurs in the stream for now onwards. Say it occurs r times. What can we say about the expected value of r^k ? Since e_i occurs f_i times in the stream, the random variable r is equally likely to be one of $\{1, \ldots, f_i\}$. Therefore,

$$\mathbb{E}[r^k] = \frac{1}{f_i} \sum_{j=1}^{i} j^k.$$

Looking at the above expression, we see that $\mathbb{E}[r^k - (r-1)^k] = \frac{1}{f_i} \cdot f_i^k$. Now, we remove the conditioning on X, we see that

$$\mathbb{E}[r^k - (r-1)^k] = \mathbb{E}[r^k - (r-1)^k | X = e_i] \Pr[X = e_i] = \frac{1}{f_i} \cdot f_i^k \cdot \frac{f_i}{m} = \frac{1}{m} \cdot F_k.$$

Therefore, the random variable $m(r^k - (r-1)^k)$ has expected value as F_k.

The only catch is that we do not know how to sample a uniformly random element of the stream. Since X is a random element of the stream, we want

$$\Pr[X = x_j] = \frac{1}{m},$$

for all values of $j = 1, \ldots, m$. However, we do not know m in advance, and so cannot use this expression directly. Fortunately, there is a more clever sampling procedure, called *reservoir sampling*, described in Figure Combining reservoir sampling with the estimator for F_k. Note that at iteration i, the algorithm just tosses a coin with probability of Heads equal to $1/i$. We show in the exercises that at any step i, X is indeed a randomly chosen element from $\{x_1, \ldots, x_i\}$.

We now need to show that this algorithm gives a good approximation to F_k with high probability. So far, we have only shown that there is a random variable, namely $Y := m(r^k - (r-1)^k)$, which is equal to F_k in expectation. But now, we want to compute the probability that Y lies within $(1 \pm \varepsilon)F_k$. In order to do this, we need to estimate the variance of Y. If the variance is not too high, we can hope to use Chebychev's bound. We know that the variance of Y is at most $\mathbb{E}[Y^2]$. Therefore, it is enough to estimate the latter quantity. Since we are going to use Chebychev's inequality, we would like to bound $\mathbb{E}[Y^2]$ in terms of $E[Y^2]$, which is same as F_k^2. The first few steps for estimating $\mathbb{E}[Y^2]$ are identical to those for estimating $\mathbb{E}[Y]$:

$$\begin{aligned}\mathbb{E}[Y^2] &= \sum_{i=1}^{n} \mathbb{E}[Y^2|X=e_i] \cdot \Pr[X=e_i] = \sum_{i=1}^{n} m^2 \cdot \mathbb{E}[(r^k-(r-1)^k)^2|X=e_i] \cdot \frac{f_i}{m} \\ &= \sum_{i=1}^{n} m f_i \cdot \frac{1}{f_i} \sum_{j=1}^{f_i} (j^k-(j-1)^k)^2 = m \cdot \sum_{i=1}^{n} \sum_{j=1}^{f_i} (j^k-(j-1)^k)^2. \qquad (14.4.1)\end{aligned}$$

We now show how to handle the expression $\sum_{j=1}^{f_i}(j^k-(j-1)^k)^2$. We first claim that

$$j^k - (j-1)^k \leq k \cdot j^{k-1}.$$

This follows from applying the mean value theorem to the function $f(x) = x^k$. Given two points $x_1 < x_2$, the mean value theorem states that there exists a number $\theta \in [x_1, x_2]$ such that $f'(\theta) = \frac{f(x_2)-f(x_1)}{x_2-x_1}$. We now substitute $j-1$ and j for x_1 and x_2 respectively, and observe that $f'(\theta) = k\theta^{k-1} \leq kx_2^{k-1}$ to get

$$j^k - (j-1)^k \leq k \cdot j^{k-1}.$$

Therefore,

$$\sum_{j=1}^{f_i}(j^k-(j-1)^k)^2 \leq \sum_{j=1}^{f_i} k \cdot j^{k-1} \cdot (j^k-(j-1)^k) \leq k \cdot f_i^{k-1} \sum_{j=1}^{f_i}(j^k-(j-1)^k) = k \cdot f_\star^{k-1} \cdot f_i^k,$$

where $f_\star$ denotes $\max_{i=1}^n f_i$. Substituting this in (14.4.1), we get

$$\mathbb{E}[Y^2] \leq k \cdot m \cdot f_\star^{k-1} F_k.$$

Recall that we wanted to bound $\mathbb{E}[Y^2]$ in terms of F_k^2. So we need to bound $m \cdot f_\star^{k-1}$ in terms of F_k. Clearly,

$$f_\star^{k-1} = (f_\star^k)^{\frac{k-1}{k}} \leq F_k^{\frac{k-1}{k}}.$$

In order to bound m, we apply Jensen's inequality to the convex function x^k to get

$$\left(\frac{\sum_{i=1}^n f_i}{n}\right)^k \leq \frac{\sum_{i=1}^n f_i^k}{n},$$

which implies that

$$m = \sum_{i=1}^n f_i \leq n^{1-1/k} \cdot F_k^{1/k}.$$

Combining all of the above inequalities, we see that

$$\mathbb{E}[Y^2] \leq k \cdot n^{1-1/k} \cdot F_k^2.$$

If we now use Chebychev's bound, we get

$$\Pr[|Y - F_k| \geq \varepsilon F_k] \leq \frac{\mathbb{E}[Y^2]}{\varepsilon^2 F_k^2} \leq k/\varepsilon^2 \cdot n^{1-1/k}.$$

The expression on the right hand side is (likely to be) larger than 1, and so this does not give us much information. The next idea is to further reduce the variance of Y by keeping several independent copies of it, and computing the average of all these copies. More formally, we maintain t i.i.d. random varaibles $Y_1, \ldots, Y_t$, each of which has the same distribution as that of Y. If we now define Z as the average of these random variables, linearity of expectation implies that $\mathbb{E}[Z]$ remains F_k. However, the variance of Z now becomes $1/t$ times that of Y (see exercises).

Therefore, if we now use Z to estimate F_k, we get

$$\Pr[|Z - F_k| \geq \varepsilon F_k] \leq \frac{k}{t \cdot \varepsilon^2} \cdot n^{1-1/k}.$$

If we want to output an estimate within $(1 \pm \varepsilon)F_k$ with probability at least $1 - \delta$, we should pick t to be $\frac{1}{\delta\varepsilon^2} \cdot n^{1-1/k}$. It is easy to check that the space needed to update one copy of Y is $O(\log m + \log n)$. Thus, the total space requirement of our algorithm is $O\left(\frac{1}{\delta\varepsilon^2} \cdot n^{1-1/k} \cdot (\log m + \log n)\right)$.

14.4.1 The median of means trick

We now show that it is possible to obtain the same guarantees about Z, but we need to keep only $O\left(\frac{1}{\varepsilon^2} \cdot \log\left(\frac{1}{\delta}\right) \cdot n^{1-1/k}\right)$ copies of the estimator for F_k. Note that we have replaced the factor $1/\delta$ by $\log(1/\delta)$. The idea is that if we use only $t = \frac{4}{\varepsilon^2} \cdot n^{1-1/k}$ copies of the variable Y in the analysis above, then we will get

$$\Pr[|Z - F_k| \geq \varepsilon F_k] \leq 1/4.$$

Procedure Reservoir Sampling

```
1 $X \leftarrow x_1$ ;
2 for  $i = 2$ to $m$ do
3     Sample a binary random variable $t_i$, which is 1 with probability $1/i$ ;
4     if  $t_i = 1$ then
5         $X \leftarrow x_i$
6 Return $X$
```

Procedure Combining reservoir sampling with the estimator for F_k

```
1 $X \leftarrow x_1, r \leftarrow 1$ ;
2 for  $i = 2$ to $m$ do
3     Sample a binary random variable $t_i$, which is 1 with probability $1/i$ ;
4     if  $t_i = 1$ then
5         $X \leftarrow x_i, r \leftarrow 1$
6     else
7     if $X = x_i$ then
          $r \leftarrow r + 1$ ;
8 Return $m\left(r^k - (r-1)^k\right)$;
```

Figure 14.5: Estimating F_k

Although this is not good enough for us, what if we keep several copies of Z (where each of these is average of several copies of Y) ? In fact, if we keep $\log(1/\delta)$ copies of Z, then at least one of these will give the desired accuracy with probability at least δ – indeed, the probability that all of them are at least εF_k far from F_k will be at most $(1/2)^{\log(1/\delta)} \leq \delta$. But we will not know *which* one of these copies is correct! Therefore, the plan is to keep slightly more copies of Z, say about $4\log(1/\delta)$. Using Chernoff bounds, we can show that with probability at least $1-\delta$, roughly a majority of these copies will give an estimate in the range $(1\pm\varepsilon)F_k$. Therefore, the *median* of all these copies will give the desired answer. This is called the "median of means" trick.

We now give details of the above idea. We keep an array of variables Y_{ij}, where i varies from 1 to $\ell := 4\log(1/\delta)$ and j varies from 0 to $t := \frac{2}{\varepsilon^2}\cdot n^{1-1/k}$. Each row of this array (i.e., elements Y_{ij}, where we fix i and vary j) will correspond to one copy of the estimate described above. So, we define $Z_i = \sum_{j=1}^{t} Y_{ij}/t$. Finally, we define Z as the median of Z_i, for $i = 1,\ldots,\ell$. We now show that Z lies in the range $(1\pm\varepsilon)F_k$ with probability at least $1-\delta$. Let E_i denote the event: $|Z_i - F_k| \geq \varepsilon F_k$. We already know that $\Pr[E_i] \leq 1/4$. Now, we want to show that the number of such events will be close to $\ell/4$. We can use Chernoff bound to prove that the size of the set $\{i : E_i \text{ occurs}\}$ is at most $\ell/2$ with at least $(1-\delta)$ probability (see exercises).

Now assume the above happens. If we look at the sequence $Z_i, i = 1,\ldots,\ell$, at least half of them will lie in the range $(1\pm\varepsilon)F_k$. The median of this sequence will also lie in the range $(1\pm\varepsilon)F_k$ for the following reason: if the median is (say) above $(1+\varepsilon)F_k$, then at least half of the events E_i will occur, which is a contradiction. Thus, we have shown the following result:

Theorem 14.1 *We can estimate the frequency moment F_k of a stream with $(1\pm\varepsilon)$ multiplicative error with probability at least $1-\delta$ using $O\left(\frac{1}{\varepsilon^2}\cdot\log\left(\frac{1}{\delta}\right)\cdot n^{1-1/k}\right)\cdot(\log m + \log n)$ space.*

14.4.2 The special case of second frequency moment

It turns out that we can estimate the second frequency moment F_2 using logarithmic space only (the above result shows that space requirement will be proportional to $\sqrt{n}$). The idea is again to have a random variable whose expected value is F_2, but now we will be able to control the variance in a much better way. We will use the idea of universal hash functions. We will require binary hash functions, i.e., they will map the set $U = \{e_1,\ldots,e_n\}$ to $\{-1,+1\}$. Recall that such a set of functions H is said to be k-universal if for any set S of indices of size at most k, and values $a_1,\ldots,a_k \in \{-1,+1\}$,

$$\Pr_{h\in H}[\wedge_{i\in S} x_i = a_i] = \frac{1}{2^{|S|}},$$

where h is a uniformly chosen hash function from H. Recall that we can construct such a set H which has $O(n^k)$ functions, and a hash function $h \in H$ can be stored using $O(k \log n)$ space only. We will need a set of 4-universal hash functions. Thus, we can store the hash function using $O(\log n)$ space only.

The algorithm for estimating F_2 is shown in Figure Second Frequency Moment. It maintains a running sum X – when the element x_t arrives, it first computes the hash value $h(x_t)$, and then adds $h(x_t)$ to X (so, we add either $+1$ or -1 to X). Finally, it outputs X^2. It is easy to check that expected value of X^2 is indeed F_2. First observe that if f_i denotes the frequency of element e_i. then $X = \sum_{i=1}^n f_i \cdot h(e_i)$. Therefore, using linearity of expectation,

$$\mathbb{E}[X^2] = \sum_{i=1}^{n} \sum_{j=1}^{n} f_i f_j \mathbb{E}[h(e_i)h(e_j)].$$

The sum above splits into two parts: if $i = j$, then $h(e_i)h(e_j) = h(e_i)^2 = 1$; and if $i \neq j$, then the fact that H is 4-universal implies that $h(e_i)$ and $h(e_j)$ are pair-wise independent random variables. Therefore, $\mathbb{E}[h(e_i)h(e_j)] = \mathbb{E}[h(e_i)] \cdot \mathbb{E}[h(e_j)] = 0$, because $h(e_i)$ is ± 1 with equal probability. So

$$\mathbb{E}[X^2] = \sum_{i=1}^{n} f_i^2 = F_2.$$

As before, we want to show that X^2 comes close to F_2 with high probability. We need to bound the variance of X^2, which is at most $\mathbb{E}[X^4]$. As above, we expand take the fourth power of the expression of X:

$$\mathbb{E}[X^2] = \sum_{i,j,k,l=1}^{n} f_i f_j f_k f_l \mathbb{E}[h(e_i)h(e_j)h(e_k)h(e_l)].$$

Each of the summands is a product of 4 terms – $h(e_i), h(e_j), h(e_k), h(e_l)$. Consider such a term. If an index is distinct from the remaining three indices, then we see that its expected value is 0. For example, if i is different from j, k, l, then $\mathbb{E}[h(e_i)h(e_j)h(e_k)h(e_l)] = \mathbb{E}[h(e_i)]\mathbb{E}[h(e_j)h(e_k)h(e_l)]$ (we are using 4-universal property here – any set of 4 distinct hash values are mutually independent). But $\mathbb{E}[h(e_i)] = 0$, and so the expected value of the whole term is 0. Thus, there are only two cases when the summand need not be 0: (i) all the four indices i, j, k, l are same – in this case $\mathbb{E}[h(e_i)h(e_j)h(e_k)h(e_l)] = \mathbb{E}[h(e_i)^4] = 1$, because $h(e_i)^2 = 1$, or (ii) exactly two of i, j, k, l take one value and the other two indices take another value – for example, $i = j$, $k = l$, but $i \neq k$. In this case, we again get $\mathbb{E}[h(e_i)h(e_j)h(e_k)h(e_l)] =$

```
Procedure Second Frequency Moment
1 $X \leftarrow 0$, $h \leftarrow$ uniformly chosen $\pm 1$ hash function from a 4-universal family. ;
2 for $i = 1$ to $m$ do
3     $X \leftarrow X + h(x_i)$
4 Return $X^2$
```

Figure 14.6: Estimating F_2

$\mathbb{E}[h(e_i)^2 h(e_k)^2 = 1$. Thus, we can simplify

$$\mathbb{E}[X^4] = \sum_{i=1}^{n} f_i^4 + \sum_{i=1}^{n} \sum_{j \in \{1,\ldots,n\} \setminus \{i\}} f_i^2 f_j^2 \leq 2F_2^2.$$

Thus we see that the variance of the estimator X^2 is at most $2\mathbb{E}[X^2]^2$. Rest of the idea is the same as in the previous section.

Exercises

Exercise 14.1 *Let f_i be the frequency of element i in the stream. Modify the Mishra-Gries algorithm (Figure Algorithm for threshold) to show that for a stream of length m, one can compute quantities $\hat{f}_i$ for each element i such that*

$$f_i - \frac{m}{k} \leq \hat{f}_i \leq f_i$$

Exercise 14.2 *Recall the reservoir sampling algorithm described in Figure Combining reservoir sampling with the estimator for F_k. Prove by induction on i that after i steps, the random variable X is a uniformly chosen element from the stream $\{x_1, \ldots, x_i\}$.*

Exercise 14.3 *Let $Y_1, \ldots, Y_t$ be t i.i.d. random variables. Show that the variance of Z, denoted by $\sigma^2(Z)$, is equal to $\frac{1}{t} \cdot \sigma^2(Y_1)$.*

Exercise 14.4 *Suppose $E_1, \ldots, E_k$ are k independent events, such that each event occurs with probability at most $1/4$. Assuming $k \geq 4\log(1/\delta)$, prove that the probability that more than $k/2$ events occur is at most δ.*

Exercise 14.5 *Let $a_1, a_2, \ldots, a_n$ be an array of n numbers in the range $[0, 1]$. Design a randomized algorithm which reads only $O(1/\varepsilon^2)$ elements from the array and estimates the average of all the numbers in the array within additive error of $\pm\varepsilon$. The algorithm should succeed with at least 0.99 probability.*

Exercise 14.6 *Consider a family of functions H where each member $h \in H$ is such that $h : \{0,1\}^k \rightarrow \{0,1\}$. The members of H are indexed with a vector $r \in \{0,1\}^{k+1}$. The value $h_r(x)$ for $x \in \{0,1\}^k$ is defined by considering the vector $x_0 \in \{0,1\}^{k+1}$ obtained by appending 1 to x and then taking the dot product of $x0$ and r modulo 2 (i.e., you take the dot product of x_0 and r, and $h_r(x)$ is 1 if this dot product is odd, and 0 if it is even). Prove that the family H is three-wise independent.*

Exercise 14.7 *Recall the setting for estimating the second frequency moment in a stream. There is a universe $U = \{e_1, \ldots, e_n\}$ of elements, and elements $x_1, x_2, \ldots$ arrive over time, where each x_t belongs to U. Now consider an algorithm which receives* **two** *streams – $S = x_1, x_2, x_3, \ldots$ and $T = y_1, y_2, y_3, \ldots$. Element x_t and y_t arrive at time t in the two streams respectively. Let f_i be the frequency of e_i in the stream S and g_i be its frequency in T. Let G denote the quantity $\sum_{i=1}^{n} f_i g_i$.*

- *As in the case of second frequency moment, define a random variable whose expected value is G. You should be able to store X using $O(\log n + \log m)$ space only (where m denotes the length of the stream).*
- *Let $F_2(S)$ denote the quantity $\sum_{i=1}^{n} f_i^2$ and $F_2(T)$ denote $\sum_{i=1}^{n} g_i^2$. Show that the variance of X can be bounded by $O(G^2 + F_2(S) \cdot F_2(T))$.*

Exercise 14.8 *You are given an array A containing n distinct numbers. Given a parameter ϵ between 0 and 1, an element x in the array A is said to be a near-median element if its position in the sorted (increasing order) order of elements of A lies in the range $[n/2 - \epsilon n, n/2 + \epsilon n]$. Consider the following randomized algorithm for finding a near-median : pick t elements from A, where each element is picked uniformly and independently at random from A. Now output the median of these t elements. Suppose we want this algorithm to output a near-median with probability at least $1 - \delta$, where δ is a parameter between 0 and 1. How big should we make t? Your estimate on t should be as small as possible. Give reasons.*