CSE341T/CSE549T **08/27/2014**

Genome Sequencing

Lecture 2

1 Why Learn About Parallel Algorithms

Due to physical and economical constraints, a typical machine we can buy now has 4 to 8 computing cores, and soon this number will be 16, 32, and 64. While the number of cores grows at a rapid pace, the per-core speed hasn't increased much over the past several years. Additionally, graphics processing units (GPUs) are highly parallel platforms with hundreds of cores readily available in commodity machines today. This gives a compelling reason to study parallel algorithms. In this course, we will learn algorithmic techniques that allow you to design and analyze parallel algorithms.

2 Cilk Plus and Cilk View Overview

We are going to use a programming language called Cilk Plus for our programming assignments. Cilk Plus is an extension of C++ programming language and simply adds a few extra keywords so that you can specify the parallelism in your program. There are many aspects of this language, but we are mostly interested in just three keywords, **spawn**, **sync** and **parallel_for**. [1] Let us look at a code example to understand the language.

```
Fib(n)
1  if n ≤ 1
2     then return 1
3  x ← spawn FIB(n − 1)
4  y ←FIB(n − 2)
5  sync
6  return x + y
```

Adding the spawn keyword before the function means that the runtime is allowed to execute that function in parallel with the ***continuation***, the remainder of the parent function. The ***sync*** keyword means that all functions that have been spawned so far have to return before control can

[1] In the language itself, they are `cilk_spawn`, `cilk_sync`, and `cilk_for` but we will just use spawn and sync in the pseudocode.

go beyond this point. Therefore, in the above code, the values of $\text{FIB}(n-1)$ and $\text{FIB}(n-2)$ can be calculated in "parallel."

Another important concept is the parallel for loop. This construct gives the runtime system permission to execute all the iterations of the for-loop in parallel with each other. For example, if you have an array $A[1..n]$ and you have to create an array B such that, for all i, $B[i] = A[i] + 1$, then you can simply write:

```
1  parallel_for i ← 1 to n
2        do B[i] ← A[i] + 1
```

In general, two things can be done in parallel if they don't depend on each other. Lets say I want to calculate the sum of all the elements of an array. Can I use the following program?

```
1  parallel_for i ← 1 to n
2        do s ← s + A[i]
```

These iterations are not independent since the variable s is changed in all the iterations. The program can lead to several different values of s depending on how the instructions are interleaved (scheduled) when run in parallel. These are called ***race conditions***. A race condition occurs when more than one (logically) parallel operations access the same memory location and one of the accesses is a write.

In this course, we are concerned with ***deterministic parallel programs***, that is, programs that give the same result regardless of the schedule. While there are many definitions of determinism, we will concentrate on programs that have no race conditions. You must be careful to make sure that when you use the spawn keyword, you are not introducing race conditions in your program.

Exercise 1 *If you want to do the sum computation in parallel without determinacy races, how would you do it?*

3 Sequencing a Genome

Being able to sequence genomes is a great scientific advance of the 20th century. This advance depends on algorithms. Note that genomes consist of many millions of characters. However, there is no way of reading long strands accurately. Most machines can only read strands of at most 1000 characters. In fact, the more modern, high-speed machines are restricted to reading strands of 100-200 characters. Therefore, the genome must be chopped up into these strands to be read, and then they must be recombined to make a single genome.

One method is to painstakingly cut the strands sequentially, but this requires a large amount of human effort in a wet-lab. A faster method is to chop up the strand randomly, but this destroys the

order. For example, the strand cattaggagtat might turn into, say, ag, gag, catt, tat, destroying the original ordering.

If we could make copies of the original sequence, is there something we could do differently to order the pieces? Let's look at the shotgun method, which according to Wikipedia is the de facto standard for genome sequencing today. It works as follows:

1. Take a DNA sequence and make multiple copies. For example, if we are cattaggagtat, we produce many copies of it:

   cattaggagtat
   cattaggagtat
   cattaggagtat

2. Randomly cut up the sequences using a "shotgun", well, actually using radiation or chemicals. For instance, we could get

   [catt] [ag] [gagtat]
   [cat] [tagg] [ag] [tat]
   [ca] [tta] [gga] [gtat]

3. Sequence each of the short fragments, which can be done in parallel with multiple sequencing machines.

4. Reconstruct the original genome from the fragments.

Steps 1–3 are done in a wet lab, and **Step 4 is where algorithms come in.** In Step 4, we want to solve the following problem: *Given a set of overlapping genome subsequences, construct the "best" sequence that includes them all.*

Definition 1 (The Shortest Superstring (SS) Problem) *Given an alphabet set Σ and a finite set of finite, non-empty strings $S \subseteq \Sigma^+$, return a shortest string r that contains every $s \in S$ as a substring of r.*

Note that in this definition, we require each $s \in S$ to appear as a contiguous block in r. That is, "ag" is a substring of "ggag" but is *not* a substring of "attg".

Observation 1 *We can throw out the strings that are contained in other strings.*

So from now on, we assume that no string is a substring of another.

Observation 2 *Each string must start at a distinct position in the result. That is, we can find a total ordering of the strings.*

We'll define a function $\texttt{join}(s_i, s_j)$ that appends s_j to s_i and removes the maximum overlap. For example, $\texttt{join}(\text{“tagg”}, \text{“gga”}) = \text{“tagga”}$. We will also define ***overlap*** between two strings. Let $\texttt{overlap}(s_i, s_j)$ denote the maximum overlap for s_i followed by s_j. This would mean $\texttt{overlap}$ (“tagg”,“gga”) $= 2$.

3.1 Brute Force Algorithm

What is the easiest algorithm? Try all orders. For each order, concatenate the consecutive strings removing the overlapping portions. Pick the order with the shortest resulting string.

For example, the permutation

catt tta tagg gga gagtat

will give us cattaggagtat after removing the overlaps (the excised parts are underlined). Note that this result happens to be the original string and also the shortest superstring.

Does this give us the shortest string?

Lemma 1 *Given a finite set of finite strings $S \subseteq \Sigma^+$, the brute force method finds the shortest superstring.*

Proof. Let r^* be any shortest superstring of S. We know that each string $s \in S$ appears in r^*. Let i_s denote the beginning position in r^* where s appears. Since we have eliminated duplicates, it must be the case that all i_s's are distinct numbers. Now let's look at all the strings in S, $s_1, s_2, \ldots, s_{|S|}$, where we number them such that $i_{s_1} < i_{s_2} < \cdots < i_{s_{|S|}}$. It is not hard to see that the ordering $s_1, s_2, \ldots, s_{|S|}$ gives us r^* after removing the overlaps.

The problem with this approach is that it has to try a very large number $(n!)$ of permutations. Therefore, even though it is highly parallel, it will still take a very long time on large number of strings. Parallelism does not convert exponential algorithms into fast algorithms, since the number of processors we have is still a polynomial number.

Exercise 2 *How would you write this brute force algorithm using parallel constructs such as **spawn**, **sync**, and **parallel for**?*

3.2 Algorithm 2: Reducing to Another Problem

We now consider converting the shortest superstring problem to another seemingly unrelated problem: the traveling salesperson[2] (TSP) problem. The two major variants of the problem are *symmetric* TSP and *asymmetric* TSP, depending on whether the graph has undirected or directed edges,

[2]This is formerly known as the traveling salesman problem.

respectively. The particular variant we're reducing to is the asymmetric version, which can be described as follows.

Definition 2 (The Asymmetric Traveling Salesperson (aTSP) Problem) *Given a weighted directed graph, find the shortest path that starts at a vertex s and visits all vertices exactly once before returning to s.*

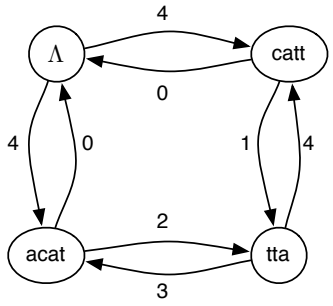
That is, find a Hamiltonian cycle of the graph such that the sum of the edge weights along the cycle is the minimum of all such cycles.

Motivated by the observation that the shortest superstring problem can be solved exactly by trying all permutations, we'll make the TSP problem try all the combinations for us. For this, we will set up a graph so that each valid Hamiltonian cycle corresponds to a permutation.

The Reduction. Now we build a graph $D = (V, A)$.

- The vertex set V has one vertex per string and a special "source" vertex Λ where the cycle starts and ends.
- The arc (directed edge) from s_i to s_j has weight $w_{i,j} = |s_j| - \texttt{overlap}(s_i, s_j)$. This quantity represents the increase in the string's length if s_i is followed by s_j. As an example, if we have "tagg" followed by "gga", then we can generate "tagga" which only adds 1 character—indeed, $|\text{"gga"}| - \texttt{overlap}(\text{"tagg"}, \text{"gga"}) = 3 - 2 = 1$.
- The weights for arcs incident to Λ are set as follows: $(\Lambda, s_i) = |s_i|$ and $(s_i, \Lambda) = 0$. That is, if s_i is the first string in the permutation, then the arc (Λ, s_i) pays for the whole length s_i.

To see this reduction in action, the input {catt, acat, tta} results in the following graph (not all edges are shown).



In this graph, a cycle through the graph that visits each vertex once corresponds to a permutation in the brute force method. Furthermore, the sum of the edge weights in that cycle is equal to the length of the superstring produced by the permutation. Since TSP considers all Hamiltonian cycles, it also corresponds to considering all orderings in the brute force method. Since the TSP

finds the min cost cycle, and assuming the brute force method is correct, then TSP finds the shortest superstring. Therefore, if we could solve TSP, we would be able to solve the shortest superstring problem.

However, TSP is an NP-hard problem. There is no known polynomial time algorithm to solve TSP. What we have accomplished so far is that we have reduced one NP hard problem to another, but the advantage is that there is a lot known about TSP, so maybe this helps.

In general, NP-hardness only suggests that there are families of instances on which the problem is hard in the worst-case. It doesn't rule out the possibility of algorithms that compute near optimal answers or algorithms that perform well on real world instances.

For this particular problem, we know efficient approximation algorithms that (1) give theoretical bounds that guarantee that the answer (i.e., the length) is within a constant factor of the optimal answer, and (2) in practice do even better than the bounds suggest.

3.3 Algorithm 3: Greedy

The third algorithm we'll consider is a simple "greedy" algorithm, which finds an "approximate" solution directly. It is not guaranteed to find the shortest superstring but there are theoretical bounds on how close it is to the optimal, and it works very well in practice.

A greedy algorithm works in steps and at each step it makes a locally optimal choice, in hopes that it will lead to a globally optimal solution, or a solution that is near optimal. Once it makes a choice, it will never reconsider whether to change that choice in a later step. We will cover greedy algorithms in more detail later in this course.

The greedy approximation algorithm for the Shortest Superstring Problem is as follows:

GreedySS(S)
1 **if** $|S| = 1$
2 **then return** S
3 **parallel_for** pair (s_i, s_j), where $s_i \neq s_j$
4 **do** Calculate $\texttt{overlap}(s_i, s_j)$
5 Calculate the maximum among all these overlaps, and say pair (s_i, s_j) have this maximum overlap.
6 Let $s_k = \texttt{join}(s_i, s_j)$
7 Set $S' = S \cup \{s_k\} / \{s_i, s_j\}$
8 **return** GREEDYSS(S')

Given a set of strings S, the algorithm finds the pair of strings s_i and s_j in S that are distinct and have the maximum overlap. The algorithm then replaces s_i and s_j with $s_k = \texttt{join}(s_i, s_j)$ in S. The new set S' is therefore one smaller. It recursively repeats this process on this new set of strings until there is only a single string left. The algorithm is greedy because at every step it takes the pair of strings that when joined will remove the greatest overlap, a locally optimal decision.

Upon termination, the algorithm returns a single string that contains all strings in the original S. However, the superstring returned is not necessarily the shortest superstring.

Exercise 3 *In the code we remove s_i, s_j from the set of strings but do not remove any strings from S that are contained within s_k = `join`(s_i, s_j). Argue why there cannot be any such strings.*

Exercise 4 *Prove that algorithm* GREEDSS *indeed returns a string that is a superstring of all original strings.*

Exercise 5 *Give an example input S for which* GREEDYSS *does not return the shortest superstring.*

Exercise 6 *Consider the following greedy algorithm for TSP. Start at the source and always go to the nearest unvisited neighbor. When applied to the graph described above, is this the same as the algorithm above? If not what would be the corresponding algorithm for solving the TSP?*

Although GREEDYSS does not return the shortest superstring, it returns an "approximation" of the shortest superstring. In particular, it is known that it returns a string that is within a factor of 3.5 of the shortest and conjectured that it returns a string that is within a factor of 2. In practice, it typically *does much better* than the bounds suggest. The algorithm also generalizes to other similar problems, e.g., when we don't require that the overlap be perfect but allow for single errors.

Of course, given that the SS problem is NP-hard, and `greedyApproxSS` does only polynomial work (see below), we cannot expect it to give an exact answer on all inputs—that would imply **P** = **NP**, which is unlikely. In literature, algorithms such as GREEDYSS that solve an NP-hard problem to within a constant factor of optimal, are called *constant-factor approximation algorithms*.

Cost Analysis We will tackle the cost of parallel algorithms in the next lecture. For now, let us analyze the sequential cost.

Let n be the size of the input S and k be the maximum length of each string. The running time of the greedy algorithm is $O(n^3k)$.

Exercise 7 *Come up with a more efficient way of implementing the greedy method.*

In the real world. Often when abstracting a problem we can abstract away some key aspects of the underlying application that we want to solve. Indeed this is the case when using the Shortest Superstring problem for sequencing genomes. In actual genome sequencing there are two shortcomings with using the SS problem. The first is that when reading the base pairs using a DNA sequencer there can be errors. This means the overlaps on the strings that are supposed to overlap perfectly might not. Generally, this can be dealt with by generalizing the Shortest Superstring problem to deal with approximate matching. Describing such a generalization is beyond the scope of this course, but basically one can give a score to every overlap and then pick the best one for each pair of fragments. The nice thing is that the same algorithmic techniques we discussed for the SS problem still work for this generalization, only the "overlap" scores will be different.

The second shortcoming of using the SS problem by itself is that real genomes have long repeated sections, possibly much longer than the length of the fragments that are sequenced. The SS problem does not deal well with such repeats. In fact when the SS problem is applied to the fragments of an initial string with longer repeats that the fragment sizes, the repeats or parts of them are removed. One method that researchers have used to deal with this problem is the so-called *double-barrel shotgun method*. In this method strands of DNA are cut randomly into lengths that are long enough to span the repeated sections. After cutting it up one can read just the two ends of such a strand and also determine its length (approximately). By using the two ends and knowing how far apart they are it is possible to build a "scaffolding" and recognize repeats. This method can be used in conjunction with the generalization of the SS discussed in the previous paragraph. In particular the SS method allowing for errors can be used to generate strings up to the length of the repeats, and the double barreled method can put them together.