Lecture 2 — Algorithmic Cost Models

Parallel and Sequential Data Structures and Algorithms, 15-210 (Qatar-Spring 2014)

Lectured by Kemal Oflazer — 14 January 2014

Today:

- Overview of algorithmic cost models for analyzing algorithms

1 Cost Models

When we analyze the cost of an algorithm formally, we need to be reasonably precise about the model we are performing the analysis in.

Question 1.1. *How precise should this model be? For example, would it help to know the exact running time for each instruction?*

A model can be arbitrarily precise but it is often helpful to abstract away from some details such as the exact running time of each (kind of) instruction. For example, the model can posit that each instruction takes a single step (unit of time) whether it is an addition, a division, or a memory access operation. Some more advanced models, which we will not consider in this class, separate between different classes of instructions, for example, a model may require analyzing separately calculation (e.g., addition or a multiplication) and communication (e.g., memory read).

Question 1.2. *Why would such abstractions help?*

There are two ways in which such abstraction helps: simplicity and portability. Abstraction simplifies analysis and makes the analysis applicable to many different actual hardware and machines, rather that just one or a few, that are consistent with the model. A model too precise can hurt portability because machines are all different.

Example 1.3. 1 **fun** $\texttt{add}(x : int, y : int) = x + y$
This ML function, when invoked with two integers performs 1 *unit of work in a model where all operations take unit time.*

Question 1.4. *What happens when we don't exactly know the input?*

†Lecture notes by Umut Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, with additional edits by Kemal Oflazer

Often for analysis all we need is the input size. We often analyze algorithms by using the input size, usually written n, m, etc..

Example 1.5. *Consider the following function:*

```
1  datatype tree = Leaf of int | Node of tree * tree
2  fun addTree(t : tree) =
3    case t of
4      Leaf x => x
5    | Node(l, r) => addTree(l) + addTree(r)
```

This ML function, when invoked with a tree of n internal nodes and m leaves, performs $4n+m$ work in a model where case statement, function call, and the add operations all take unit time. If we count returns as well the work is $5n+2m$.
Since in a binary tree m is at most $n+1$, the work can be written just in terms of n as $7n+1$.

Question 1.6. *How do we know whether we need to count the "return" or not?*

At the level of abstractions that we write our algorithms in this course, there will be "hidden" instructions such as return or looping instructions, when for example working with a set. These can be difficult to know, without using a lower level of abstraction.

Fortunately, for our purposes, asymptotic analysis will be sufficient. Instead of performing an exact analysis, we will instead analyze asymptotic costs, e.g., using big-O notation.

Example 1.7.
```
1  fun add(x : int, y : int) = x + y
```
This ML function performs $\Theta(1)$ work.

Example 1.8. *Consider the following function:*

```
1  datatype tree = Leaf of int | Node of tree * tree
2  fun addTree(t : tree) =
3    case t of
4      Leaf x => x
5    | Node(l, r) => addTree(l) + addTree(r)
```

This ML function, when invoked with a tree of n nodes and m leaves, performs $\Theta(n+m)$ work in a model where case statement, function call, and the add operations all take unit time. Since in a binary tree m is at most $n+1$, the work is $\Theta(n)$.

Asymptotic analysis thus adds another level of abstraction, making our analysis further removed from the actual machine.

Question 1.9. *Do the aforementioned advantages of this kind of abstraction, simplicity and portability still apply?*

The advantages continue to hold. Asymptotic analysis further enables comparing algorithms in terms of how they scale to large inputs.

Example 1.10. *Some sorting algorithms have $\Theta(n \log n)$ work and others $\Theta(n^2)$. Clearly the $\Theta(n \log n)$ algorithm scales better, but perhaps the $\Theta(n^2)$ is actually faster on small inputs.*

In this class we are concerned with how algorithms scale, and therefore asymptotic analysis is indeed what we want. Because we are using asymptotic analysis, the exact constants in the model do not matter, but what matters is that the asymptotic costs are well defined.

Remark 1.11. Since you have seen big-O, big-Theta, and big-Omega in 15-122, 15-150 and 15-251 (if you have taken it) we will not be covering it here but would be happy to review it in office hours.

There are two important ways to define cost models: machine based, and language based, where the cost is defined directly on programming constructs. Both types can be applied to analyzing either sequential and parallel computations.

Question 1.12. *What are the advantages of using a machine-based and a language-based model?*

When using a machine model, we have to reason about how a program translates to that machine. For sequential programs this can be straightforward when using low-level languages such as C. For parallel programs, however, such a translation can be difficult to pin down because it requires reasoning about the scheduling of parallel computations on parallel hardware.

Traditionally machine models have been preferred, but in this course, as in 15-150, we will use a language-based cost model. We first review the traditional machine model.

2 The RAM model for sequential computation

Traditionally, algorithms have been analyzed in the Random Access Machine (RAM) model.[1] This model assumes a single processor accessing unbounded memory indexed by the non-negative integers. The processor interprets sequences of machine instructions (code) that are stored in the memory. Instructions include basic arithmetic and logical operations (e.g. +, -, *, and, or, not), reads from and writes to arbitrary memory locations, and conditional and unconditional jumps to other locations in the code. The cost of a computation is measured in terms of the number of instructions executed by the machine, and is referred to as *time*.

[1] Not to be confused with Random Access Memory (RAM)

This model has served well for analyzing the asymptotic runtime of sequential algorithms and most work on sequential algorithms to date has used this model. It is therefore important to understand the model. One reason for its success is that it is reasonably easy to reason about the cost of algorithms and code because there is an easy mapping from algorithmic pseudo code and sequential languages such as C and C++ to the model. As mentioned earlier, the model should only be used for deriving asymptotic bounds (*i.e.*, using big-O, big-Theta and big-Omega) and not for trying to predict exact runtimes. One reason for this is that on a real machine not all instructions take the same time, and furthermore not all machines have the same instructions.

We note that one problem with the RAM model is that it assumes that accessing all memory locations has the same cost. On real machines this is not at all the case. In fact, there can be a factor of over 100 between the time for accessing a word of memory from the first level cache and accessing it from main memory. Various extensions to the RAM model have been developed to account for this cost. For example one variant is to assume that the cost for accessing the the i^{th} memory location is $f(i)$ for some function f, e.g. $f(i) = \log(i)$. Fortunately, however, most of the algorithms that turn out to be the best in these more detailed models are also good in the RAM. Therefore analyzing algorithms in the simpler RAM model is a reasonable approximation to analyzing in the more refined models. Hence the RAM has served quite well despite not fully accounting for memory costs.

The model we use in this course also does not directly account for the variance in memory costs. Towards the end of the course, if time permits, we will discuss how it can be extended to capture memory costs.

3 Parallel RAM Model

The problem with the RAM for our purposes is that the model is sequential. There is an extension of the RAM model for parallelism, which is called the parallel random access machine (PRAM). It consists of p processors sharing a memory. All processors execute the same instruction on each step. We will not be using this model since we find it awkward to work with, especially because everything has to be divided between the p processors. However, most of the ideas presented in this course also work with the PRAM, and many of them were originally developed in the context of the PRAM.

4 Work-Span Model

Instead of using a machine model, in this course, as with 15-150, we will define a model more directly tied to programming constructs without worrying how it is mapped onto a machine. We believe that the model we use makes it much easier to separate the high-level concepts of parallelism from low-level machine-specific details. It is therefore more amenable to get you to "think parallel", one goal of this course. As it turns out, there is a way to map the costs we derive onto costs for specific machines.

To formally define a cost model in terms of programming constructs requires a so-called "operational semantics". We won't define a complete semantics, but will give a partial semantics to give a sense of how it works.

Work and Span We will measure complexity in terms of two costs: work and span. Roughly speaking the *work* corresponds to the total number of operations we perform, and *span* to the longest chain of dependencies. Although you have seen work and span in 15-150, we will review the definition here and go into some more detail.

We define work and span in terms of simple compositional rules over expressions in the language. For an expression e we will use $W(e)$ to indicate the work needed to evaluate that expression and $S(e)$ to indicate the span.

We define work and span in terms of composing the costs across sub expressions.

Question 4.1. *Can you think of different ways of composing two subexpressions e_1 and e_2 of an expression e?*

For our purposes, expressions are either composed sequentially (one after the other) or in parallel (they can run at the same time). When composed sequentially we add the work and we add the span, and when composed in parallel we add the work but take the maximum of the span. Basically that is it!

Question 4.2. *How can we know that two expressions are composed sequentially or in parallel?*

Determining whether two expressions are composed sequentially or in parallel can be very difficult or very easy. In an imperative language, it can be very hard to figure out when computations depend on each other and, therefore, when it is safe to put things together in parallel. In particular subexpressions can interact through shared state.

Example 4.3. *In C, consider the expression:*

```
foo(2) + foo(3)
```

Are `foo(2)` *and* `foo(3)` *parallel, i.e., can we evaluate them in parallel?*
No, it is not safe to make the two calls to `foo(x)` *in parallel since they might interact. Suppose*

```
int y = 0;
int foo(int x) { return y = y + x; }
```

With `y` *starting at* `0`*, the expression* `foo(2) + foo(3)` *could lead to several different outcomes depending on how the instructions are interleaved (scheduled) when run in parallel. This interaction is often called a race condition. You will sure to encounter it when working with imperative languages, for example in other courses.*
Note that in this simple example, it might be possible to determine that two subexpressions interact by inspecting the code. In general, however, this is impossible a priori because it depends on run-time values.

Question 4.4. *In a functional language, can we know that two expressions are composed sequentially or in parallel?*

In a functional language, as long as the output for one expression is not required for the input of another, it is safe to run the two expressions in parallel.

Example 4.5. *In the expression $e_1 + e_2$ where e_1 and e_2 are themselves other expressions (e.g. function calls) we could run the two expressions in parallel giving the rule*

$$S(e_1 + e_2) = 1 + \max(S(e_1), S(e_2)).$$

This rule says the two subexpressions run in parallel so that we take the max of the span of each subexpression. But the addition operation has to wait for both subexpressions to be done. It therefore has to be done sequentially after the two parallel subexpressions and hence the reason why there is a plus 1 in the expression $1 + \max(S(e_1), S(e_2))$.

In this course, as we will use only purely functional constructs, it is always safe to run expressions in parallel.

Notation. To make it clear whether expressions are evaluated sequentially or in parallel, in the pseudo code we use the notation (e_1, e_2) to mean that the two expressions run sequentially (even when they could run in parallel), and $e_1 \parallel e_2$ to mean that the two expressions run in parallel. Both constructs return the pair of results of the two expressions.

Example 4.6. *The expression* `fib(6)` || `fib(7)` *returns the pair* $(8, 13)$.

In addition to the || construct we assume the set-like notation we use in pseudo code $\{f(x) : x \in A\}$ also runs in parallel, i.e., all calls to $f(x)$ run in parallel. These rules for composing work and span are outlined in Figure 1. Note that the rules are the same for work and span except for the two parallel constructs we just mentioned.

Remark 4.7. As there is no || construct in the ML, in your assignments you will need to specify in comments when two calls run in parallel. We will also supply an ML function `par (f1,f2)` with type `(unit -> α) × (unit -> β) -> α × β`. This function executes the two functions that are passed in as arguments in parallel and returns their results as a pair. For example:

```
par (fn => fib(6), fn => fib(7))
```

returns the pair $(8, 13)$. We need to wrap the expressions in functions in ML so that we can make the actual implementation run them in parallel. If they were not wrapped both arguments would be evaluated sequentially before they are passed to the function `par`. Also in the ML code you do not have the set notation $\{f(x) : x \in A\}$, but as mentioned before, it is basically equivalent to a `map`. Therefore, for ML code you can use the rules:

$$W(\texttt{map } f\ \langle s_0, \ldots, s_{n-1}\rangle) = 1 + \sum_{i=0}^{n-1} W(f(s_i))$$

$$S(\texttt{map } f\ \langle s_0, \ldots, s_{n-1}\rangle) = 1 + \max_{i=0}^{n-1} S(f(s_i))$$

$$
\begin{aligned}
W(c) &= 1 \\
W(\text{op } e) &= 1 \\
W((e_1, e_2)) &= 1 + W(e_1) + W(e_2) \\
W(e_1 \mid\mid e_2) &= 1 + W(e_1) + W(e_2) \\
W(\texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}) &= 1 + W(e_1) + W(e_2[\text{Eval}(e_1)/x])) \\
W(\{f(x) : x \in A\}) &= 1 + \sum_{x \in A} W(f(x))
\end{aligned}
$$

$$
\begin{aligned}
S(c) &= 1 \\
S(\text{op } e) &= 1 \\
S((e_1, e_2)) &= 1 + S(e_1) + S(e_2) \\
S((e_1 \mid\mid e_2)) &= 1 + \max(S(e_1), S(e_2)) \\
S(\texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}) &= 1 + S(e_1) + S(e_2[\text{Eval}(e_1)/x])) \\
S(\{f(x) : x \in A\}) &= 1 + \max_{x \in A} S(f(x))
\end{aligned}
$$

Figure 1: Composing work and span costs. In the first rule c is any constant value (e.g. 3). In the second rule op is a primitive operator such as op.+, op.$*$, op. , ... The next rule, the pair (e_1, e_2) evaluates the two expressions sequentially, whereas the rule $(e_1 \mid\mid e_2)$ evaluates the two expressions in parallel. Both return the results as a pair. In the rule for `let` the notation Eval(e) evaluates the expression e and returns the result, and the notation $e[v/x]$ indicates that all free (unbound) occurrences of the variable x in the expression e are replaced with the value v. These rules are representative of all rules of the language. Notice that all the rules for span are the same as for work except for parallel application indicated by $(e_1 \mid\mid e_2)$ and the parallel map indicated by $\{f(x) : x \in A\}$. The expression e inside $W(e)$ and $S(e)$ have to be closed. Note, however, that in the rules such as for `let` we replace all the free occurrences of x in the expression e_2 with their values before applying W.

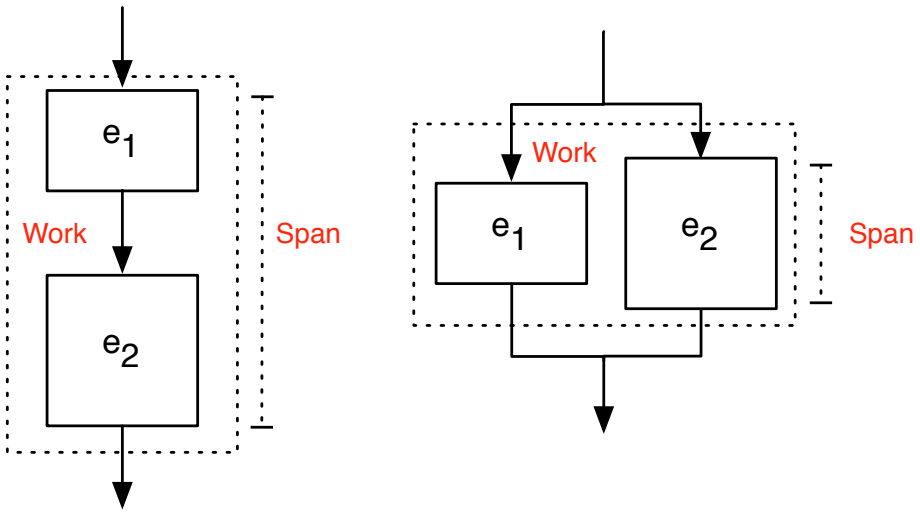


Figure 2: Sequential (left) and parallel (right) compositions of two evaluated expressions e_1 and e_2.

Figure 3: DAGs for sequential (left) and parallel (right) composition of two evaluated expressions e_1 and e_2 where G_1 and G_2 correspond to the DAGs of e_1 and e_2 respectively.

Representing parallel computations as DAG's. We can represent a parallel computation with a DAG (directed acylic graph) by recursively composing the DAG's of subcomputations either in sequential or in parallel form. See the illustration in Figure 3 for an example. Note that the DAG is not "static" but rather can only be constructed by observing the expressions evaluated.

Example 4.8. *Consider applying a function* `f` *to each element of a set.*

- $W(\texttt{map f } \mathcal{S}) = 1 + \sum_{s \in \mathcal{S}} W(f(s))$.
- $S(\texttt{map f } \mathcal{S}) = 1 + \max_{s \in \mathcal{S}} S(f(s))$.

Example 4.9. *Let expressions compose sequentially.*

- $W(\texttt{let a = f x in g a end}) = 1 + W(f\ x) + W(g\ a)$
- $S(\texttt{let a = f x in g a end}) = 1 + S(f\ x) + S(g\ a)$

Exercise 1. *What is the work and span of* `reduce v f` S*?*
Is work $\Theta(n)$*?*

Finally there are two additional related concepts that are useful at this point:

- *Upper bound*: The maximum asymptotic work (and span) that a given algorithm needs as a function of the size of the input. Any input of size n can use at most this amount of work (and span).
- *Lower bound*: The minimum asymptotic work (and span) that *any* algorithm for a problem needs as a function of the size of the input. Any algorithm for the problem needs to use *at least* this much work and span for any input of size n.

The important point here is that we speak of an upper bound for work and span for specific algorithm for a problem, while we speak of a lower bound for a problem – independent of the algorithm used.

Parallelism: An additional notion of cost that is important in comparing algorithms is the *parallelism* of an algorithm. Parallelism, sometimes called *average parallelism*, is simply defined as the work over the span:

$$\mathbb{P} = \frac{W}{S}$$

Parallelism informs us approximately how many processors we can use efficiently. As you saw in the processor scheduling example in 15-150, $\mathbb{P}$ is the most parallelism you can get. That is, it measures the limit on the performance that can be gained when executed in parallel.

Example 4.10. *For a mergesort with work* $\theta(n \log n)$ *and span* $\theta(\log^2 n)$ *the parallelism would be* $\theta(n/\log n)$.
Suppose $n = 10,000$ *and if* $W(n) = \theta(n^3) \approx 10^{12}$ *and* $S(n) = \theta(n \log n) \approx 10^5$ *then* $\mathbb{P}(n) \approx 10^7$*, which is a lot of parallelism. But, if* $W(n) = \theta(n^2) \approx 10^8$ *then* $\mathbb{P}(n) \approx 10^3$*, which is much less parallelism. The decrease in parallelism is not because of the span was large, but because the work was reduced.*

Question 4.11. *What are ways in which we can increase parallelism?*

We can increase parallelism by decreasing span and increasing work. Increasing work is not desirable because it leads to an inefficient algorithm. Increasing work does not necessarily lead to an improvement in completion time, unless span can also be reduced.

Work efficiency. We say that a parallel algorithm is *work efficient* if it performs asymptotically the same work as the best known sequential algorithm for that problem.

Example 4.12. *A (comparison-based) parallel sorting algorithm with $\Theta(n \log n)$ work is work efficient; one with $\Theta(n^2)$ is not, because we can sort sequentially with $\Theta(n \log n)$ work.*

Designing parallel algorithms. In parallel-algorithm design, we aim to keep parallelism as high as possible but without increasing work. In general the goals in designing efficient algorithms are

1. first priority: to keep work as low as possible, and
2. second priority: keep parallelism as high as possible (and hence the span as low as possible).

In this course we will mostly cover work-efficient algorithms where the work is the same or close to the same as the best sequential time. Indeed this will be our goal throughout the course. Now among the algorithm that have the same work as the best sequential time we will try to achieve the greatest parallelism.

5 Scheduling

An important advantage of the work-depth model is that is allows us to design parallel algorithms without having to worry about the details of how they are executed on an actual parallel machine. In other words, we never have to worry about mapping of the parallel computation to processors, i.e., *scheduling*.

Question 5.1. *Is scheduling a challenging task? Why?*

Scheduling can be challenging because a parallel algorithm generates tasks on the fly as it runs, and it can generate a massive number of them, typically much more than the number of processors available when running.

Example 5.2. *A parallel algorithm with $\Theta(n/\log n)$ parallelism can generate as many as 10^5 parallel subcomputations or tasks at the same time, even when running on a multicore computer with, for example, 10 cores.*

Scheduler. Mapping parallel tasks to available processor so that each processor remains busy as much as possible is the task of a scheduler. The scheduler works by taking all parallel tasks, which are generated dynamically as the algorithm evaluates, and assign them to processors. If only one processor is available, for example, then all tasks will run on that one processor. If two processor are available, the task will be divided between the two.

Question 5.3. *Can you think of a scheduling algorithm?*

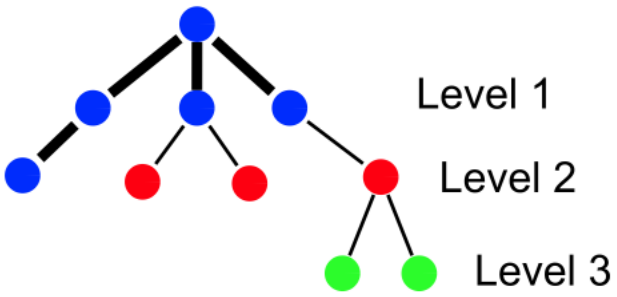
Greedy scheduling. We say that a scheduler is *greedy* if whenever there is a processor available and a task ready to execute, then it assigns the task to the processor and start running it immediately. Greedy schedulers have a very nice property that is summarized by the following:

Definition 5.4. The *greedy scheduling principle* says that if a computation is run on p processors using a greedy scheduler, then the total time (clock cycles) for running the computation is bounded by

$$T_p < \frac{W}{p} + S \tag{1}$$

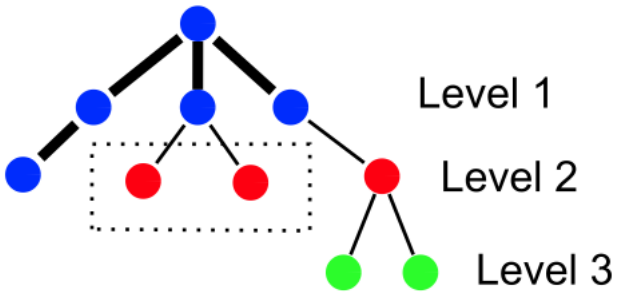
where W is the work of the computation, and S is the span of the computation (both measured in units of clock cycles).

We can motivate this upper bound with the following observation.[2] The following diagram illustrates a breadth-first tree of the work dependence graph. Blue vertices symbolize completed computations. Red vertices have their dependencies satisfied, and are ready to be completed. Green nodes need their dependencies to be satisfied before being computed. This diagram requires $W = 10$ computations. Suppose there are $P = 2$ processors.

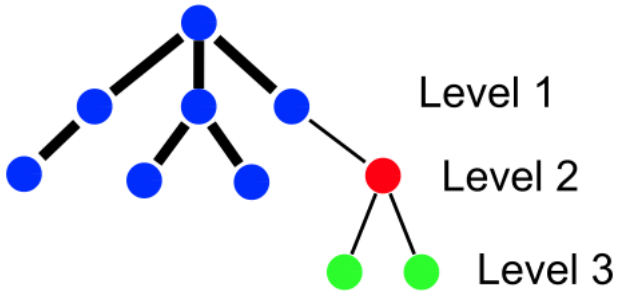


The next step would involve the following two computations:

[2] Summarized from 15-492 – Parallel Algorithms course notes by Guy Blelloch, Sept 6, 2007.



Since there are more available computations than processors, no work is wasted. After the computation is completed, there is only 1 computation left with dependencies satisfied. Time will be wasted on the following step since not every processor would be busy.



So, this diagram illustrates how time can only be wasted on the completion of a level. Since there are S levels (span is S), and $\frac{W}{P}$ time is required if no time is wasted, the latter can be increased once at each level.

The time to execute the computation cannot be any better than $\frac{W}{p}$ clock cycles since we have a total of W clock cycles of work to do and the best we can possibly do is divide it evenly among the processors. Also note that the time to execute the computation cannot be any less than S clock cycles since S represents the longest chain of sequential dependencies. Therefore the very best we could do is:

$$T_p \geq \max\left(\frac{W}{p}, S\right)$$

We therefore see that a greedy scheduler does reasonably close to the best possible. In particular $\frac{W}{p} + S$ is never more than twice $\max(\frac{W}{p}, S)$ and when $\frac{W}{p} \gg S$ the difference between the two is very small. Indeed we can rewrite equation 1 above in terms of the parallelism $\mathbb{P} = W/S$ as follows:

$$\begin{aligned} T_p &< \frac{W}{p} + S \\ &= \frac{W}{p} + \frac{W}{\mathbb{P}} \\ &= \frac{W}{p}\left(1 + \frac{p}{\mathbb{P}}\right) \end{aligned}$$

Therefore as long as $\mathbb{P} \gg p$ (the parallelism is much greater than the number of processors) then we get near perfect speedup. (Speedup is W/T_p and perfect speedup would be p).

Remark 5.5. No real schedulers are fully greedy. This is because there is overhead in scheduling the job. Therefore there will surely be some delay from when a job becomes ready until when it starts up. In practice, therefore, the efficiency of a scheduler is quite important to achieving good efficiency. Also the bounds we give do not account for memory affects. By moving a job we might have to move data along with it. Because of these affects the greedy scheduling principle should only be viewed as a rough estimate in much the same way that the RAM model or any other computational model should be just viewed as an estimate of real time.

6 Analysis of Shortest Superstring Algorithms

As examples of how to use our cost model we will analyze a couple of the algorithms we described for the shortest superstring problem: the brute force algorithm and the greedy algorithm.

6.1 The Brute Force Shortest Superstring Algorithm

Recall that the idea of the brute force algorithm for the SS problem is to try all permutations of the input strings and for each permutation to determine the maximal overlap between adjacent strings and remove them. We then pick whichever remaining string is shortest, if there is a tie, we pick any of the shortest. We can calculate the overlap between all pairs of strings in a preprocessing phase. Let n be the size of the input S and m be the total number of characters across all strings in S, i.e.,

$$m = \sum_{s \in S} |s|.$$

Note that $n \leq m$. The preprocessing step can be done in $O(m^2)$ work and $O(\log n)$ span (see analysis below). This is a low order term compared to the other work, as we will see, so we can ignore it.

Now to calculate the length of a given permutation of the strings with overlaps removed we can look at adjacent pairs and look up their overlap in the precomputed table. Since there are n strings and each lookup takes constant work, this requires $O(n)$ work. Since all lookups can be done in parallel, it will require only $O(1)$ span. Finally we have to sum up the overlaps and subtract it from m. The summing can be done with a **reduce** in $O(n)$ work and $O(\log n)$ span. Therefore the total cost is $O(n)$ work and $O(\log n)$ span.

As we discussed in the last lecture the total number of permutations is $n!$, each of which we have to check for the length. Therefore the total work is $O(nn!) = O((n+1)!)$. What about the span? Well we can run all the tests in parallel, but we first have to generate the permutations. One simple way is to start by picking in parallel each string as the first string, and then for each of these picking in parallel another string as the second, and so forth. The pseudo code looks something like this:

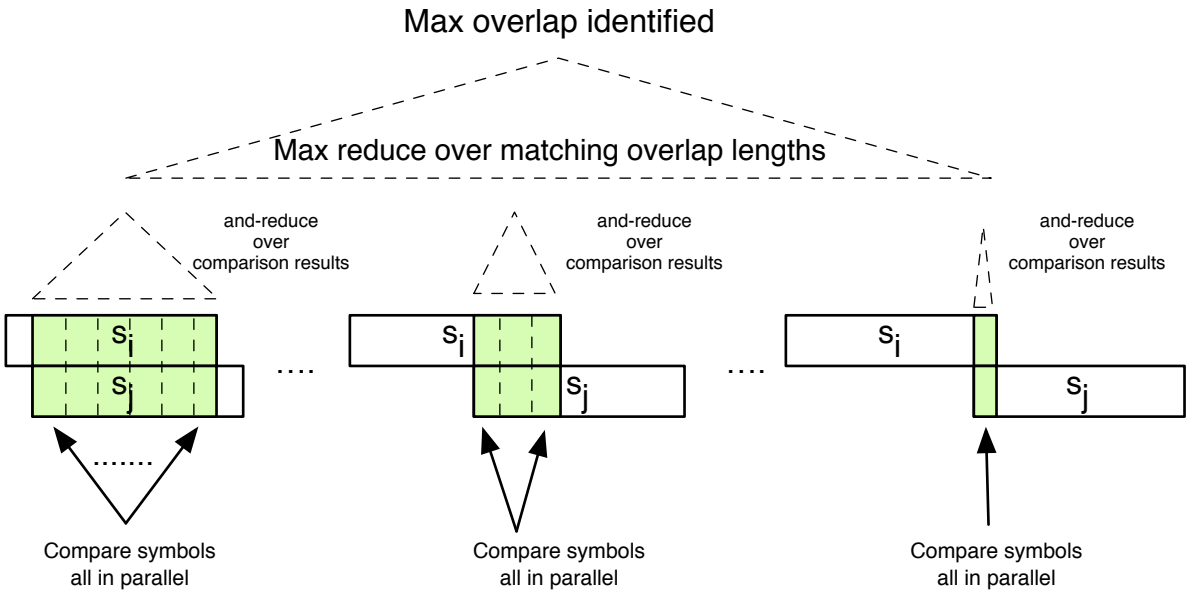


Figure 4: Preprocessing a pair of strings s_i and s_j in parallel. All $n(n-1)$ pairs can be preprocessed in parallel.

```
func permutations S =
   if |S| = 1 then {S}
   else
      {append([s], p) :  s in S, p in permutations(S/s)}
```

What is the span of this code?

6.2 The Greedy Shortest Superstring Algorithm

We'll consider a straightforward implementation, although the analysis is a little tricky since the strings can vary in length. First we note that calculating `overlap`(s_1, s_2) and `join`(s_1, s_2) can be done in $O(|s_1||s_2|)$ work and $O(\log(|s_1| + |s_2|))$ span.This is simply by trying all overlap positions between the two strings, seeing which ones match, and picking the largest. The logarithmic span is needed for picking the largest matching overlap using a reduce. See Figure 4 for how this happens for one pair of strings; all pairs can be processed in parallel.

Let W_{ov} and S_{ov} be the work and span for calculating all pairs of overlaps (the line {(`overlap` $(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}$).

We have

$$\begin{aligned}
W_{ov} &\leq \sum_{i=1}^{n}\sum_{j=1}^{n} W(\texttt{overlap}(s_i, s_j))) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} O(|s_i||s_j|) \\
&\leq \sum_{i=1}^{n}\sum_{j=1}^{n} (k_1 + k_2|s_i||s_j|) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} k_1 + \sum_{i=1}^{n}\sum_{j=1}^{n} (k_2|s_i||s_j|) \\
&= k_1 n^2 + k_2 \sum_{j=1}^{n} |s_j| (\sum_{i=1}^{n} |s_i|) \\
&= k_1 n^2 + k_2 m^2 \\
&\in O(m^2)
\end{aligned}$$

and since all pairs can be done in parallel,

$$\begin{aligned}
S_{ov} &\leq \max_{i=1}^{n} \max_{j=1}^{n} S(\texttt{overlap}(s_i, s_j)) \\
&\in O(\log m)
\end{aligned}$$

The `maxval` can be computed in $O(m^2)$ work and $O(\log m)$ span using a simple reduce. The other steps cost no more than computing `maxval`. Therefore, not including the recursive call each call to `greedyApproxSS` costs $O(m^2)$ work and $O(\log m)$ span.

Finally, we observe that each call to `greedyApproxSS` creates S' with one fewer element than S, so there are at most n calls to `greedyApproxSS`. These calls are inherently sequential because one call must complete before the next call can take place. Hence, the total cost for the algorithm is $O(nm^2)$ work and $O(n \log m)$ span, which is highly parallel.

Exercise 2. *Can you come up with a more efficient way of implementing the greedy method.*