Chapter 15

Cost Models

Any algorithmic analysis must assume a ***cost model*** that defines the resource costs required by a computation. There are two broadly accepted ways of defining cost models: machine-based and language-based cost models.

1 Machine-Based Cost Models

Definition 15.1 (Machine-Based Cost Model)**.** A ***machine-based (cost) model*** takes a machine model and defines the cost of each instruction that can be executed by the machine—often unit cost per instruction. When using a machine-based model for analyzing an algorithm, we translate the algorithm so that it can be executed on the machine and then analyze the cost of the machine instructions used by the algorithm.

Remark. Machine-based models are suitable for deriving asymptotic bounds (i.e., using big-O, big-Theta and big-Omega) but not for predicting exact runtimes. The 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.

1.1 RAM Model

The classic machine-based model for analyzing sequential algorithms is the ***Random Access Machine*** or ***RAM***. In this model, a machine consists of a single processor that can access unbounded memory; the memory is indexed by the natural numbers. 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. Each instruction takes unit time to execute, including those that access memory. The execution-time, or simply ***time*** of a computation is measured in terms of the number of instructions executed by the machine. Because the model is sequential (there is only one processor) time and work are the same.

Critique of the RAM Model. Most research and development of sequential algorithms has used the RAM model for analyzing time and space costs. One reason for the RAM model's success is that it is relatively easy to reason about the costs because algorithmic pseudo code can usually be translated to the model. Similarly, code in low-level sequential languages such as C can also be translated (compiled) to the RAM Model relatively easily. When using higher level languages, the translation from the algorithm to machine instructions becomes more complex and we find ourselves making strong, possibly unrealistic assumptions about costs, even sometimes without being aware of the assumptions.

More broadly, the RAM model becomes difficult to justify in modern languages. For example, in object- oriented languages certain operations may require substantially more time than others. Likewise, features of modern programming languages such as automatic memory management can be difficult to account for in analysis. Functional features such as higher-order functions are even more difficult to reason about in the RAM model because their behavior depends on other functions that are used as arguments. Such functional features, which are the mainstay of "advanced" languages such as the ML family and Haskell, are now being adopted by more mainstream languages such as Python, Scala, and even for more primitive (closer to the machine) languages such as C++. All in all, it requires significant expertise to understand how an algorithm implemented in modern languages today may be translated to the RAM model.

Remark. One aspect of the RAM model is the assumption that accessing all memory locations has the same uniform cost. On real machines this is not the case. In fact, there can be a factor of 100 or more difference between the time for accessing different locations in memory. For example, all machines today have caches and accessing the first-level cache is usually two orders of magnitude faster than accessting main memory.

Various extensions to the RAM model have been developed to account for this non-uniform cost of memory access. One variant assumes that the cost for accessing the i^{th} memory location is $f(i)$ for some function f, e.g. $f(i) = \log(i)$. Fortunately, however, most algorithms that are good in these more detailed models are also good in the RAM model. Therefore analyzing algorithms in the simpler RAM model is often a reasonable approximation to analyzing in the more refined models. Hence the RAM has served quite well despite not fully accounting for non-uniform memory costs.

The model we use in this book also does not account for non-uniform memory costs, but as with the RAM the model can be refined to account for it.

1.2 PRAM: Parallel Random Access Machine

The RAM model is sequential but can be extended to use multiple processors which share the same memory. The extended model is called the Parallel Random Access Machine.

PRAM Model. A ***Parallel Random Access Machine***, or ***PRAM***, consist of p sequential random access machines (RAMs) sharing the same memory. The number of processors, p, is a parameter of the machine, and each processor has a unique index in $\{0, \ldots, p-1\}$, called the ***processor id***. Processors in the PRAM operate under the control of a common clock and execute one instruction at each time step. The PRAM model is most usually used as a *synchronous* model, where all processors execute the same algorithm and operate on the same data structures. Because they have distinct ids, however, different processors can do different computations.

Example 15.1. We can specify a PRAM algorithm for adding one to each element of an integer array with p elements as shown below. In the algorithm, each processor updates certain elements of the array as determined by its processor id, id.

```
(* Input:  integer array A. *)
$arrayAdd = A[id] \leftarrow A[id] + 1$
```

If the array is larger than p, then the algorithm would have to divide the array up and into parts, each of which is updated by one processor.

SIMD Model. Because in the PRAM all processors execute the same algorithm, this typically leads to computations where each processor executes the same instruction but possibly on different data. PRAM algorithms therefore typically fit into ***single instruction multiple data***, or ***SIMD***, programming model. Example 15.1 shows an example SIMD program.

Critique of PRAM. Since the model is synchronous, and it requires the algorithm designer to map or schedule computation to a fixed number of processors, the PRAM model can be awkward to work with. Adding a value to each element of an array is easy if the array length is equal to the number of processors, but messier if not, which is typically the case. For computations with nested parallelism, such as divide-and-conquer algorithms, the mapping becomes much more complicated.

We therefore don't use the PRAM model in this book. Most of the algorithms presented in this book, however, also work with the PRAM (with more complicated analysis), and many of them were originally developed using the PRAM model.

2 Language Based Models

Language-Based Cost-Models. A ***language-based model*** takes a language as the starting point and defines cost as a function mapping the expressions of the language to their cost. Such a cost function is usually defined as a recursive function over the different forms of expressions in the language. To analyze an algorithm by using a language-based model, we apply the cost function to the algorithm written in the language. In this book, we use a language-based cost model, called the work-span model.

2.1 The Work-Span Model

Our language-based cost model is based on two cost metrics: work and span. Roughly speaking, the ***work*** of a computation corresponds to the total number of operations it per-forms, and the ***span*** corresponds to the longest chain of dependencies in the computation. The work and span functions can be defined for essentially any language ranging from low-level languages such as C to higher level languages such as the ML family. In this book, we use the SPARC language.

Notation for Work and Span. For an expressions e, or an algorithm written in a language, we write $W(e)$ for the work of e and $S(e)$ for the span of e.

Example 15.2. For example, the notation

$$W(7+3)$$

denotes the work of adding 7 and 3.

The notation

$$S(\mathit{fib}(11))$$

denotes the span for calculating the 11^{th} Fibonacci number using some particular code for fib.

The notation

$$W(\mathit{mySort}(a))$$

denotes the work for mySort applied to the sequence a.

Using Input Size. Note that in the third example the sequence a is not defined within the expression. Therefore we cannot say in general what the work is as a fixed value. However, we might be able to use asymptotic analysis to write a cost in terms of the length of a, and in particular if `mySort` is a good sorting algorithm we would have:

$$W(\texttt{mySort}(a)) = O(|a| \log |a|).$$

Often instead of writing $|a|$ to indicate the size of the input, we use n or m as shorthand. Also if the cost is for a particular algorithm, we use a subscript to indicate the algorithm. This leads to the following notation

$$W_{\texttt{mySort}}(n) = O(n \log n).$$

where n is the size of the input of `mysort`. When obvious from the context (e.g. when in a section on analyzing `mySort`) we typically drop the subscript, writing $W(n) = O(n \log n)$.

Definition 15.2 (SPARC Cost Model)**.** The work and span of SPARC expressions are defined below. In the definition and throughout this book, we write $W(e)$ for the work of the expression and $S(e)$ for its span. Both work and span are cost functions that map an expression to a integer cost. As common in language-based models, the definition follows the definition of expressions for SPARC (Sparc Chapter). We make one simplifying assumption in the presentation: instead of considering general bindings, we only consider the case where a single variable is bound to the value of the expression.

In the definition, the notation Eval(e) evaluates the expression e and returns the result, and the notation $[v/x]\ e$ indicates that all free (unbound) occurrences of the variable x in the expression e are replaced with the value v.

$$\begin{array}{lcl}
W(v) & = & 1 \\
W(\texttt{lambda}\ p\,.\,e) & = & 1 \\
W(e_1\ e_2) & = & W(e_1) + W(e_2) + W([\text{Eval}(e_2)/x]\ e_3) + 1 \\
 & & \text{where Eval}(e_1) = \texttt{lambda}\ x\,.\,e_3 \\
W(e_1\ \texttt{op}\ e_2) & = & W(e_1) + W(e_2) + 1 \\
W(e_1\ ,\ e_2) & = & W(e_1) + W(e_2) + 1 \\
W(e_1\ ||\ e_2) & = & W(e_1) + W(e_2) + 1 \\
W\left(\begin{array}{l}\texttt{if}\ e_1 \\ \texttt{then}\ e_2 \\ \texttt{else}\ e_3\end{array}\right) & = & \left\{\begin{array}{ll} W(e_1) + W(e_2) + 1 & \text{if Eval}(e_1) = \texttt{true} \\ W(e_1) + W(e_3) + 1 & \text{otherwise}\end{array}\right. \\
W\left(\begin{array}{l}\texttt{let}\ x = e_1 \\ \texttt{in}\ e_2\ \texttt{end}\end{array}\right) & = & W(e_1) + W([\text{Eval}(e_1)/x]\ e_2) + 1 \\
W((e)) & = & W(e)
\end{array}$$

$$\begin{array}{lcl}
S(v) & = & 1 \\
S(\texttt{lambda}\ p\,.\,e) & = & 1 \\
S(e_1\ e_2) & = & S(e_1) + S(e_2) + S([\text{Eval}(e_2)/x]\ e_3) + 1 \\
 & & \text{where Eval}(e_1) = \texttt{lambda}\ x\,.\,e_3 \\
S(e_1\ \text{op}\ e_2) & = & S(e_1) + S(e_2) + 1 \\
S(e_1\ ,\ e_2) & = & S(e_1) + S(e_2) + 1 \\
S(e_1\ ||\ e_2) & = & \max\left(S(e_1), S(e_2)\right) + 1 \\
S\left(\begin{array}{l}\texttt{if}\ e_1 \\ \texttt{then}\ e_2 \\ \texttt{else}\ e_3\end{array}\right) & = & \left\{\begin{array}{ll} S(e_1) + S(e_2) + 1 & \text{Eval}(e_1) = \texttt{true} \\ S(e_1) + S(e_3) + 1 & \text{otherwise}\end{array}\right. \\
S\left(\begin{array}{l}\texttt{let}\ x = e_1 \\ \texttt{in}\ e_2\ \texttt{end}\end{array}\right) & = & S(e_1) + S([\text{Eval}(e_1)/x]\ e_2) + 1 \\
S((e)) & = & S(e)
\end{array}$$

Example 15.3. Consider the expression $e_1 + e_2$ where e_1 and e_2 are themselves other expressions (e.g., function application). Note that this is an instance of the rule e_1 op e_2, where op is a plus operation. In SPARC, we evaluate this expressions by first evaluating e_1 and then e_2 and then computing the sum. The work of the expressions is therefore

$$W(e_1 + e_2) = W(e_1) + W(e_2) + 1.$$

The additional 1 accounts for computation of the sum.

For the `let` expression, we first evaluate e_1 and assign it to x before we can evaluate e_2. Hence the fact that the span is composed sequentially, i.e., by adding the spans.

Example 15.4. In SPARC, `let` expressions compose sequentially.

$$\begin{array}{lcl}
W(\texttt{let}\ y = f(x)\ \texttt{in}\ g(y)\ \texttt{end}) & = & 1 + W(f(x)) + W(g(y)) \\
S(\texttt{let}\ y = f(x)\ \texttt{in}\ g(y)\ \texttt{end}) & = & 1 + S(f(x)) + S(g(y))
\end{array}$$

Note that all the rules for work and span have the same form except for parallel application, i.e., $(e_1 \,||\, e_2)$. Recall that parallel application indicates that the two expressions can be evaluated in parallel, and the result is a pair of values containing the two results. In this case we use maximum for combining the span since we have to wait for the longer of the two. In all other cases we sum both the work and span. Later we will also add a parallel construct for working with sequences.

Example 15.5. The expression $(\texttt{fib}(6) \,||\, \texttt{fib}(7))$ runs the two calls to `fib` in parallel and returns the pair $(8, 13)$. It does work

$$1 + W(\texttt{fib}(6)) + W(\texttt{fib}(7))$$

and span

$$1 + \max\left(S(\texttt{fib}(6)), S(\texttt{fib}(7))\right).$$

If we know that the span of `fib` grows with the input size, then the span can be simplified to $1 + S(\texttt{fib}(7))$.

Remark. When assuming purely functional programs, it is always safe to run things in parallel if there is no explicit sequencing. For the expression $e_1 + e_2$, for example, it is safe to evaluate the two expressions in parallel, which would give the rule

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

i.e., we wait for the later of the two expression fo finish, and then spend one additional unit to do the addition. However, in this book we use the convention that parallelism has to be stated explicitly using $||$.

Definition 15.3 (Average Parallelism)**.** Parallelism, sometimes called ***average parallelism***, is defined as the work over the span:

$$\overline{P} = \frac{W}{S}.$$

Parallelism informs us approximately how many processors we can use efficiently.

Example 15.6. For a mergesort with work $\Theta(n \log n)$ and span $\Theta(\log^2 n)$ the parallelism would be $\Theta(n/\log n)$.

Example 15.7. Consider an algorithm with work $W(n) = \Theta(n^3)$ and span $S(n) = \Theta(n \log n)$. For $n = 10,000$, $\overline{P}(n) \approx 10^7$, which is a lot of parallelism. But, if $W(n) = \Theta(n^2) \approx 10^8$ then $\overline{P}(n) \approx 10^3$, which is much less parallelism. Note that the decrease in parallelism is not because of an increase in span but because of a reduction in work.

Designing Parallel Algorithms. In parallel-algorithm design, we aim to keep parallelism as high as possible. Since parallelism is defined as the amount of work per unit of span, we can do this by decreasing span. We can increase parallelism by increasing work also, but this is usually not desirable. In designing parallel algorithms our goals are:

1. to keep work as low as possible, and

2. to keep span as low as possible.

Except in cases of extreme parallelism, where for example, we may have thousands or more processors available to use, the first priority is usually to keep work low, even if it comes at the cost of increasing span.

Definition 15.4 (Work efficiency)**.** We say that a parallel algorithm is *work efficient* if it perform asymptotically the same work as the best known sequential algorithm for that problem.

Example 15.8. 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.

Note. In this book, we will mostly cover work-efficient algorithms where the work is the same or close to the same as the best sequential time. Among the algorithm that have the same work as the best sequential time, our goal will be to achieve the greatest parallelism.

2.2 Scheduling

Scheduling involves executing a parallel program by mapping the computation over the processors in such a way to minimize the completion time and possibly, the use of other resources such as space and energy. There are many forms of scheduling. This section describes the scheduling problem and briefly reviews one particular technique called greedy scheduling.

2.2.1 Scheduling Problem

An important advantage of the work-span model is that it 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.

Scheduling can be challenging, because a parallel algorithm generates independently executable ***tasks*** on the fly as it runs, and it can generate a large number of them, typically many more than the number of processors.

Example 15.9. A parallel algorithm with $\Theta(n/\lg n)$ parallelism can easily generate millions of parallel subcomptutations or task at the same time, even when running on a multicore computer with 10 cores. For example, for $n = 10^8$, the algorithm may generate millions of independent tasks.

Definition 15.5 (Scheduler)**.** A ***scheduling algorithm*** or a ***scheduler*** is an algorithm for mapping parallel tasks to available processors. The scheduler works by taking all parallel tasks, which are generated dynamically as the algorithm evaluates, and assigning them to processors. If only one processor is available, for example, then all tasks will run on that one processor. If two processors are available, the task will be divided between the two.

Schedulers are typically designed to minimize the execution time of a parallel computation, but minimizing space usage is also important.

2.2.2 Greedy Scheduling

Definition 15.6 (Greedy Scheduler)**.** 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 starts running it immediately. Greedy schedulers have an important property that is summarized by the greedy scheduling principle.

Definition 15.7 (Greedy Scheduling Principle)**.** The ***greedy scheduling principle*** postulates 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

$$T_P < \tfrac{W}{P} + S$$

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

Optimality of Greedy Schedulers. This simple statement is powerful.

Firstly, the time to execute the computation cannot be less 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.

Secondly, the time to execute the computation cannot be any less than S clock cycles, because S represents the longest chain of sequential dependencies. Therefore we have

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

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 $\max(\frac{W}{P}, S)$.

Furthermore, greedy scheduling is particularly good for algorithms with abundant parallellism. To see this, let's rewrite the inequality of the Greedy Principle in terms of the parallelism $\overline{P} = W/S$:

$$\begin{aligned} T_P &< \tfrac{W}{P} + S \\ &= \tfrac{W}{P} + \tfrac{W}{\overline{P}} \\ &= \tfrac{W}{P}\left(1 + \tfrac{P}{\overline{P}}\right). \end{aligned}$$

Therefore, if $\overline{P} \gg P$, i.e., the parallelism is much greater than the number of processors, then the parallel time T_P is close to W/P, the best possible. In this sense, we can view parallelism as a measure of the number of processors that can be used effectively.

Definition 15.8 (Speedup)**.** The ***speedup*** S_P of a P-processor parallel execution over a sequential one is defined as

$$S_P = T_s/T_P,$$

where T_S denotes the sequential time. We use the term ***perfect speedup*** to refer to a speedup that is equal to P.

When assessing speedups, it is important to select the best sequential algorithm that solves the same problem (as the parallel one).

Exercise 15.1. Describe the conditions under which a parallel algorithm would obtain near perfect speedups.

Remark. Greedy Scheduling Principle does not account for the time it requires to compute the (greedy) schedule, assuming instead that such a schedule can be created instantaneously and at no cost. This is of course unrealistic and there has been much work on algorithms that attempt to match the Greedy Scheduling Principle. No real schedulers can match it exactly, because scheduling itself requires work. For example, there will surely be some delay from when a task becomes ready for execution and when it actually starts executing. In practice, therefore, the efficiency of a scheduler is quite important to achieving good efficiency. Because of this, the greedy scheduling principle should only be viewed as an asymptotic cost estimate in much the same way that the RAM model or any other computational model should be just viewed as an asymptotic estimate of real time.