Lecture 1 — Overview, Algorithmic Techniques and Cost Models

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Margaret Reid-Miller base on notes by Guy Blelloch — 17 January 2012

1 Administrivia

Welcome to 15-210 Parallel and Sequential Data Structures and Algorithms. This course covers methods for designing, analyzing, and programming sequential and parallel algorithms and data structures, with an emphasis on fundamental concepts that will be applicable across a wide variety of problem domains, and transferable across a reasonably broad set of programming languages and computer architectures. *There is no textbook for the class.* We will (electronically) distribute course notes and supplemental reading materials as we go along. Each set of

The course web site is located at

`http://www.cs.cmu.edu/~15210`

Please take time to look around the website. While you're at it, we *strongly* encourage you to read and understand the collaboration policy.

Instead of spamming you with mass emails, we will post announcements, clarifications, corrections, hints, etc. on the course web site and on the class bboards—please check them on a regular basis. The bboards are `academic.cs.15-210.announce` for announcements from the course staff and `academic.cs.15-210.discuss` for general discussions and clarification questions.

There will be approximately weekly assignments due at 11:59pm, 2 midterm exams, and a final exam.

Since this is only the second incarnation of 15-210 we would appreciate feedback any time. Please come talk to us if you have suggestions and/or concerns.

2 Course Overview

This is a data structures and algorithms course, but it differs from a traditional course in data structures and algorithms in many ways. In particular, this course centers around the following themes:

- defining precise problem and data abstractions
- designing and programming correct and efficient algorithms and data structures for given problems and data abstractions

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

We will be looking at the following relationship matrix:

	Abstraction	**Implementation**
Functions	Problem	Algorithm
Data	Abstract Data Type	Data Structure

A *problem* specifies precisely the problem statement and the intended input/output behavior in an abstract form. It is an abstract (precise) definition of the problem but does not describe how it is solved. Whereas an *algorithm* is what allows us to solve a problem; it is an implementation that meets the intended specification. Typically, a problem will have many algorithmic solutions. For example, sorting is a problem—it specifies what the input is (e.g., a sequence of numbers) and the intended output (e.g., an ordered sequence of numbers)—but `quicksort` is an algorithm that solves the sorting problem and insertion sort is another algorithm. The distinction between problems vs. algorithms is standard in literature.

Similarly, an *abstract data type* (ADT) specifies precisely an interface for accessing data in an abstract form without specifying how the data is structured, whereas a *data structure* is a particular way of organizing the data to support the interface. For an ADT, the interface is specified in terms of a set of operations on the type. For example, a priority queue is an ADT with operations that might include `insert`, `findMin`, and `isEmpty?`. Various data structures can be used to implement a priority queue, including binary heaps, arrays, and balanced binary trees. The terminology ADTs vs. data structures is not as widely used as problems vs. algorithms. In particular, sometimes the term data structure is used to refer to both the interface and the implementation. We will try to avoid such usage in this class.

The crucial differences between this course and a traditional course on data structures and algorithms lie in our focus on *parallelism* and our emphasis on *functional language* implementation. We'll also put heavy emphasis on helping you define precise and concise abstractions.

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. The table below shows some example timings from a recent paper.

	Serial	**Parallel**		
		1-core	8-core	32h-core
Sorting 10 million strings	2.9	2.9	.4	.095
Remove duplicates 10M strings	.66	1.0	.14	.038
Min spanning tree 10M edges	1.6	2.5	.42	.14
Breadth first search 10M edges	.82	1.2	.2	.046

32h stands for 32 cores with hyperthreading.

In this table, the sorting algorithm used in sequential is not the same as the algorithm used in parallel. Notice that going from 1 core to 8 core is not quite 8 times faster, as can be expected

with overheads associated with parallelization. The magic is going from 8 core to 32 cores, which is more than four times as fast. The reason for this extra speedup is that the 32-core machine uses hyperthreading, which allows for 64 threads and provides the additional speedup. The other algorithms don't show quite the same speedup.

It is unlikely that you will get similar speedup using Standard ML. But maximizing speedup by highly tuning an implementation is not the goal of this course. That is an aim of 15-213. Functional languages, however, are great for studying parallel algorithms—they're safe for parallelism because they avoid mutable data. They also generally provide a clear distinction between abstraction and implementations, and are arguably easier to build "interesting" applications quickly.

3 Algorithmic Techniques

In this class we are going to cover many algorithmic techniques/approaches for solving problems. In the context of the shortest superstring (SS) problem we already mentioned three techniques: brute force, reducing one problem to another, and the greedy approach. In the next lecture we will discuss divide-and-conquer and in previous classes I'm sure you have seen other techniques. To give a preview of what else we will be covering in this course, here is a list of techniques we will cover. All these techniques are useful for both sequential and parallel algorithms, however some, such as divide-and-conquer, play a even larger role in parallel algorithms.

Brute Force: The brute force approach typically involves trying all possibilities. In SS problem, for example, we argued that every solution has to correspond to a permutation of the inputs with overlaps removed. The brute force approach therefore tried all permutations and picked the best. Since there are $n!$ permutations, this solution is not "tractable" for large problems. In many other problems there are only polynomially many possibilities. For example in the stock market problem on the first homework you need only try all pairs of elements from the input sequence. There are only $O(n^2)$ such pairs. However, even $O(n^2)$ is not good, since as you will work out in the assignment there are solutions that require only $O(n)$ work. One place the brute force approach can be very useful is when writing a test routine to check the correctness of more efficient algorithms. Even if inefficient for large n the brute force approach could work well for testing small inputs. The brute force approach is often the simplest solution to a problem.

Reducing to another problem: Sometimes the easiest thing to do is just reduce the problem to another problem for which known algorithms exist. In the case of the SS problem we reduced it to what would seem superficially to be a very different problem, the Traveling Salesperson problem. The reduction was pretty straightforward, but how would anyone even think of doing such a translation. When you get more experienced with algorithms you will start recognizing similarities between problems that on the surface seem very different.

Inductive techniques: The idea behind inductive techniques is to solve one or more smaller problems that can be used to solve the original problem. The technique most often uses recursion to solve the sub problems and can be proved correct using (strong) induction. Common techniques that fall in this category include:

- *Divide-and-conquer.* Divide the problem on size n into $k > 1$ subproblems on sizes $n_1, n_2, \ldots n_k$, solve the problem recursively on each, and combine the solutions to get the solution to the original problem.
- *Greedy.* For a problem on size n use some greedy approach to pull out one element leaving a problem of size $n - 1$. Solve the smaller problem.
- *Contraction.* For a problem of size n generate a significantly smaller (contracted) instance (e.g. of size $n/2$), solve the smaller instance, and then use the result to solve the original problem. This only differs from divide and conquer in that we make one recursive call instead of multiple.
- *Dynamic Programming.* Like divide and conquer, dynamic programming divides the problem into smaller problems and then combines the solutions to the subproblems to find a solution to the original problem. The difference, though, is that the solutions to subproblems are reused multiple times. It is therefore important to store the solutions for reuse either using memoization or by building up a table of solutions.

Collection Types: Some techniques make heavy use of the operations on abstract data types representing collections of values. Abstract collection types that we will cover in this course include: Sequences, Sets, Priority Queues, Graphs, and Sparse Matrices.

Randomization: Randomization in is a powerful technique for getting simple solutions to problems. We will cover a couple of examples in this course. Formal cost analysis for many randomized algorithms, however, requires probability theory beyond the level of this course.

Once you have defined the problem, you can look into your bag of techniques and, with practice, you will find a good solutions to the problem. When we say a *good solution* we mean:

1. **Correct:** Clearly correctness is most important.
2. **Low cost:** Out of the correct solutions, we would prefer the one with the lowest cost. This brings us to our next topic:

4 Cost Models

When we analyze the cost of an algorithm formally, we need to be reasonably precise in what model we are performing the analysis. Typically when analyzing algorithms the purpose of the model is not to calculate exact running times (this is too much to ask), but rather just to analyze asymptotic costs (*i.e.*, big-O). These costs can then be used to compare algorithms in terms of how they scale to large inputs. For example, as you know, some sorting algorithms use $O(n \log n)$ work and others $O(n^2)$. Clearly the $O(n \log n)$ algorithm scales better, but perhaps the $O(n^2)$ 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. Since you have seen big-O, big-Theta, and big-Omega in 15-122, 15-150 and 15-251 we will not be covering it here but would be happy to review it in recitation.

There are two important ways to define cost models, one based on machines and the other based more directly on programming constructs. Both types can be applied to analyzing either sequential and parallel computations. Traditionally machine models have been used, but in this course, as in 15-150, we will use a model that abstract to the programming constructs. We first review the traditional machine model.

4.1 The RAM model for sequential computation:

Traditionally, algorithms have been analyzed in the Random Access Machine (RAM)[1] model. 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 execute by the machine, and is referred to as *time*.

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 what this model is. One reason for its success is that there is an easy mapping from algorithmic pseudocode and sequential languages such as C and C++ to the model and so it is reasonably easy to reason about the cost of algorithms and code. 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.

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 and everything has to be divided onto 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.

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.

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

4.2 The Parallel Model Used in this Course

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. The goal of the course is to get you to "think parallel" and we believe the the model we use makes it much easier to separate the high-level concepts of parallelism from low-level machine-specific details. 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. 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 dependences. Although you have seen work and span in 15-150, we will review the definition here in 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 can then specify rules for composing the costs across sub expressions. 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! We do, however, have to specify when things are composed sequentially and when 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. So for example, 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 an imperative language we have to be much more careful. Indeed 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. e.g. For example in C, in the expression:

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

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 and will be covered further in more advanced courses.

In this course, as we will use only purely functional constructs, it is always safe to run expressions in parallel. To make it clear whether expressions are evaluated sequentially or in parallel, in

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

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

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.

the pseudocode we write we will use the notation (e_1, e_2) to mean that the two expressions run sequentially (even when they could run in parallel), and $e_1 \mid\mid e_2$ to mean that the two expressions run in parallel. Both constructs return the pair of results of the two expressions. For example `fib`(6) || `fib`(7) would return the pair (8, 13). In addition to the || construct we assume the set-like notation we use in pseudocode $\{f(x) : x \in A\}$ 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.

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 `parallel(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:

```
parallel (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 `parallel`. 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:

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

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

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

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

For example for a mergesort with work $\theta(n \log n)$ and span $\theta(\log^2 n)$ the parallelism would be $\theta(n/\log n)$. Parallelism represents roughly 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.

For example, 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.

Goals: In parallel algorithm design, we would like to keep the 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
2. Second priority: keep parallelism as high as possible (and hence the span as low as possible).

In this course we will mostly cover 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.

Under the hood: In the parallel model we will be using a program can generate tasks on the fly and can generate a huge amount of parallelism, typically much more than the number of processors available when running. It therefore might not be clear how this maps onto a fixed number of processors. That is the job of a scheduler. The scheduler will take all of these tasks, which are generated dynamically as the program runs, and assign them to processors. If only one processor is available, for example, then all tasks will run on that one processor.

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

Definition 4.1. 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).

This is actually a very powerful statement. 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 better than S clock cycles since S represents the longest chain of sequential dependences. 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 $P \gg p$ (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).

Truth in advertising. 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.