

# Homework 1: Work, Span and Algorithm Design

15-897: Parallel Computing: Theory and Practice

Spring 2016

TA: Stefan Muller (smuller@cs.cmu.edu)

Out: 1/21/16

Due: 2/4/16

## 1 Logistics

- Make sure you sign up for the course on Piazza. This is how corrections and clarifications to the assignment, as well as other important announcements, will be made.
- The first few lectures have been a whirlwind of notation and languages: asymptotic notation, functional programming, C++, etc. Because this is a graduate course, students come from all different backgrounds and it's perfectly understandable if you're not familiar with all of this. If that's the case, feel totally free to go to Umut or Stefan, who will be happy to help you get up to speed.

## 2 Submission

Submit your work by sending Stefan (smuller@cs.cmu.edu) an email with:

- A pdf containing your answers to the theory questions. L<sup>A</sup>T<sub>E</sub>X is preferable (and the handout contains a template for you to use), but a scan of *neatly* handwritten answers is OK as well.
- `tasks.hpp` modified to include your code for the coding questions.
- The `plots.pdf` file generated by `pplot` with your speedup curve.

## 3 Theory

### 3.1 Be greedy

In class, we saw a version of Brent's theorem that states that for a dag with work  $W$  and span  $S$ , *any* greedy  $P$ -processor schedule has length at most  $\frac{W}{P} + S \frac{P-1}{P}$ .

**Task 1** [15] *Show that this bound is tight by describing an algorithm which takes a span  $S \geq 2$  and a number of processors  $P \geq 2$  and constructs a dag for which there exists a greedy schedule of length exactly  $\frac{W}{P} + S \frac{P-1}{P}$ . (In other words, you will be giving a family of dags, one for each  $S$  and  $P$ .) Using diagrams to help explain your algorithm will likely be helpful.*

**Describe the schedule** which takes time  $\frac{W}{P} + S \frac{P-1}{P}$  (see hint below).

**Hint:** Note carefully the quantifiers here. For a given dag, there may be many possible greedy schedules of substantially different lengths. Your dag may admit a shorter greedy schedule, but as long as there is a way of scheduling it in a greedy fashion which takes the required amount of time, the bound is tight.

**Relaxation:** In the context of parallel computations, we generally require that dags be connected and have one source node (node of in-degree zero) and one sink node (of out-degree zero). For the purposes of *this question only*, you may ignore these requirements, e.g. by showing just the "middle portion" of a computation which already contains some number of disconnected sub-dags. (This could be turned into a proper dag by joining the disconnected sub-dags with balanced trees at the beginning and end; the work and span that this adds could be made negligible if the overall work and span of the computation are large enough.)

## 3.2 Toward practical schedulers

One simple type of schedule we discussed in class was a level-by-level schedule, which executes all of the vertices of a dag at level  $i$  ( $P$  at a time) before proceeding to level  $i + 1$ . Brent's Theorem shows that this schedule is within a factor of 2 of optimal, so it may seem like this hints at the design for an ideal *scheduler*, which we could implement by keeping two global containers of ready nodes: one for level  $i$  and one for level  $i + 1$ . The scheduler takes  $P$  vertices from the  $i$  container (if  $P$  are available) and executes them. Any other vertices they enable are added to the  $i + 1$  container. When the  $i$  container is empty, the roles of the two containers are switched.

**Task 2** [5] Suppose we use this scheduler to compute `fib(42)` (the 42<sup>nd</sup> Fibonacci number) using the simple, naturally parallel and recursive exponential-work algorithm. How many vertices are there in the dag at level 20 (count the initial vertex as level 0), assuming we do not do any granularity control? That is, how many vertices will need to be stored in the container at this level? Approximately how does this number scale at lower levels of the dag? What does this mean for our scheduler's space usage?

**Task 3** [5] Describe another likely problem (not having to do with space usage) with our scheduler. Note that the description of the implementation was deliberately vague. You may make whatever (reasonable) assumptions you'd like about the exact implementation or the underlying system, but please document whatever important and non-obvious assumptions you make so we know how to interpret your answer.

**Task 4** [5] Suggest an improvement to the scheduler which has better space usage. You do not need to describe its implementation in detail, or to prove any properties of it. This is just to get you thinking about scheduling approaches. Your scheduler also need not solve the problem you posed in the previous task (though it would be great if it did!)

## 4 Practice

### 4.1 Setting up PASL

PASL is already installed on Andrew UNIX machines, so you don't need to do this yourself. To get started, ssh to `linux.andrew.cmu.edu` and log in with your Andrew ID and password (this should work on cluster machines as well, but you'd rather be antisocial and work from the comfort of your office or home, right?). Run the following command to add the location of PASL to your PATH:

```
setenv PATH /opt/rh/devtoolset-3/root/usr/bin:$PATH
```

(You may also want to add this to your path in your shell configuration file so that you don't have to do this each time.) Look through Chapters 8.3-8.7 to get familiar with the tools we'll be using in this assignment, including the Makefile and pbench. Note that pbench is already public in AFS, and you were given a symlink to it in your handout, so you can run `prun` (and `pplot`) with

```
/shared/pbench/prun ...
```

where ... is the arguments you want to pass to `prun` or `pplot`.

### 4.2 Sequence Library

In the `support/lib` folder of your assignment handout, there's a file `sparrray.hpp`, which defines a type `sparrray` of sequences (of the kind we've been discussing in classes) and many operations over them (including `map`, `reduce` and `scan`). Familiarize yourself with the functions provided by this file.

### 4.3 Merge

In this task, you will write a function which merges two **sorted** sequences (call them  $A$  and  $B$ ) into one sorted sequence with  $O(n)$  work and  $O(\log^2 n)$  span, where  $n$  is the length of the longer list. This can be done using a simple divide-and-conquer algorithm.

To avoid excessive allocation of sequences, the signature of `merge` indicates that a sequence should be passed in, into which the results will be placed. To make recursion easy, it also includes arguments to indicate which subsequence of  $A$  and  $B$  should be merged, and where in `res` the function should start storing the result.

```
void merge(const sparray& xs, const sparray& ys, sparray& res,
          long reslo, long Alo, long Ahi, long Blo, long Bhi)
```

This function should merge the subsequence of `xs` from indices `Alo` (inclusive) to `Ahi` (exclusive) with the subsequence of `ys` from indices `Blo` (inclusive) to `Bhi` (exclusive). The result should be stored in `res` starting at index `reslo` (no corresponding “hi” index is needed since the result will contain exactly `Ahi - Alo + Blo - Bhi` elements).

**Task 5** [5] First, implement a sequential version `merge_seq` (with the same signature) by filling in the skeleton in `tasks.hpp`. This will be used as the sequential baseline and by PASL’s automatic granularity control.

**Task 6** [15] Implement the parallel version, `merge_par` in `tasks.hpp`. Your implementation should have  $O(n)$  work and  $O(\log^2 n)$  span, where  $n$  is the length of the longer list. Make sure to use PASL’s granularity control features to revert to the sequential version when necessary.

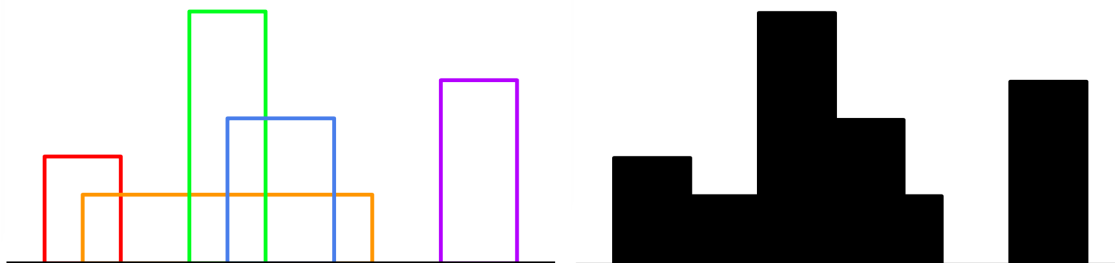
#### Hints:

- Assume, without loss of generality, that  $A$  is longer than  $B$  (if it’s not, you can call `merge_par` recursively with the arguments switched).
- To make this a divide-and-conquer algorithm, we want to split  $A$  in half (call the two halves  $A_1$  and  $A_2$ ) and recur. However, to make joining the results easy, we also want to split  $B$  into  $B_1$  and  $B_2$  such that all elements of  $B_1$  are smaller than all elements in  $A_2$  (and similarly for  $A_1$  and  $B_2$ ). How can you do this in  $O(\log n)$  work and span?

### 4.4 The Pittsburgh Skyline Problem (with thanks and apologies to 15-210)



Feast your eyes upon downtown Pittsburgh from Mt. Washington, a vista named the second most beautiful in America by USA Weekend in 2003 <sup>1</sup>. From our perspective, most of the buildings are roughly rectangular (the exceptions being those like PPG Place), so let's assume from now on that all buildings are rectangles. We'll also assume that the ground is perfectly flat (this assumption holds as long as you don't actually go for walk, bike ride or drive in Pittsburgh) so that all the rectangles rest on the same line. The *skyline* of these rectangles is basically their silhouette.



#### 4.4.1 Buildings

Under the assumptions given, each building can be represented by a triple  $(\ell, h, r)$  which describes a rectangle with corners  $(\ell, 0)$ ,  $(\ell, h)$ ,  $(r, h)$ , and  $(r, 0)$ . We will assume  $\ell < r$  so that  $\ell$  and  $r$  are the left and right sides of the building, and  $h$  is the height of the building. The buildings will be passed in as three sequences of the same length,  $L$ ,  $H$  and  $R$ . Building  $i$  consists of the triple  $(L[i], H[i], R[i])$ .



#### 4.4.2 Definition

The skyline of a set of buildings given by sequences  $L$ ,  $H$  and  $R$  is a pair of sequences  $(X, Y)$  where  $X$  is a *sorted* sequence consisting of all of the  $x$ -coordinates in  $L$  and  $R$ , and  $Y[i]$  is the height of the skyline (the height of the largest building) at  $X[i]$ :

$$H[i] = \max_{\{j | L[j] \leq X[i] < R[j]\}} H[j]$$

Note that this definition may include **redundant** points: points whose height is unchanged from the nearest point to the left. (You can think about how to remove redundant points while remaining within the cost bounds, and even try it out if you'd like, but this is not required.)

#### 4.4.3 Example

In the diagrams below, skyline points are marked with . Redundant points are marked in the first diagram with .

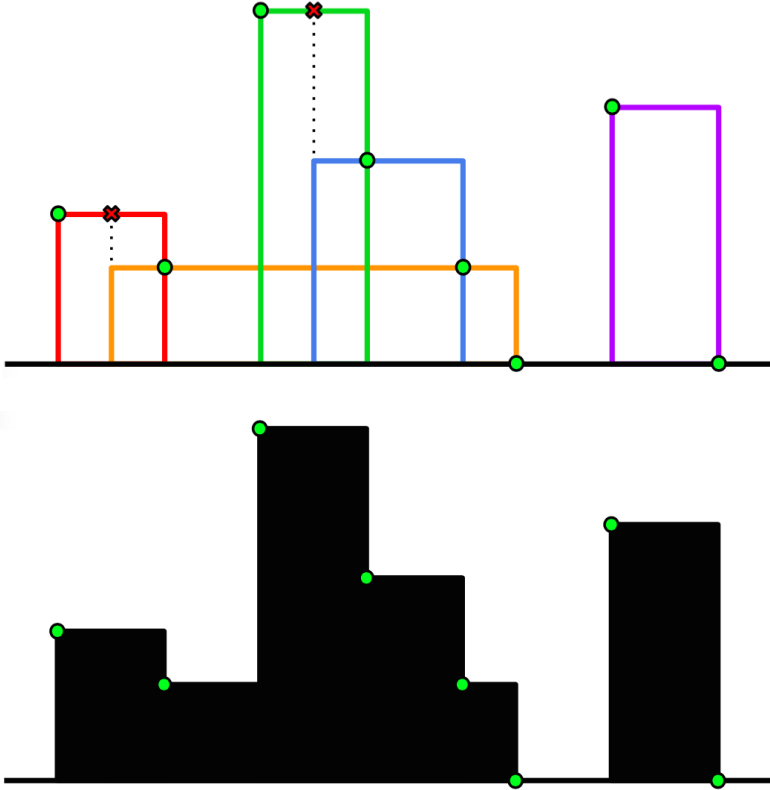
---

<sup>1</sup>Seriously: <http://www.usaweekend.com/article/99999999/LIVING01/91015001/The-10-Most-Beautiful-Places-America>

Buildings:		
$\ell$	$h$	$r$
2	3	4
3	2	11
6	7	8
7	4	10
13	5	15

Skyline:

$x$	$y$
2	3
3	3
4	2
6	7
7	7
8	4
10	2
11	0
13	5
15	0



Your implementation should be divide-and-conquer over the sequences of buildings (which are given in no particular order). This means you need to think about two things: how to generate a skyline from one building and how to merge two valid skylines.

**Task 7** [40] In *tasks.hpp*, fill in the function

```
void skyl(const sparray &lefts, const sparray &heights,
         const sparray &rights, sparray &xres, sparray &yres,
         long lo, long hi)
```

**Important notes (these are part of the task):**

- As with `merge`, the arguments include `lo` and `hi`, which are the (inclusive and exclusive, respectively) indices over which to work with the sequences `lefts`, `heights` and `rights`. Unlike with `merge`, `xres` and `yres` will be exactly large enough to hold the  $x$  and  $y$  coordinates of the resulting skyline, so you will start storing results in these sequences from index 0. This means that, when you call `skyl` recursively, you will need to allocate new sequences to store the recursive results.
- Your implementation should have work  $O(n \log n)$  and span  $O(\log^3 n)$ . **Correction 2/2/16:** The span bound was previously  $\log^2 n$ . You can meet the work bound if your divide-and-conquer algorithm satisfies the recurrence

$$W_{\text{skyline}}(n) = 2W_{\text{skyline}}(n/2) + O(n)$$

where the  $O(n)$  comes from the work of combining the two skylines.

- Make sure to make use of PASL's automatic granularity control! You don't need to write a sequential version of your divide-and-conquer function, but you should use a `cstmt` to sequentialize the parallel calls at small enough instances. You should do the same (revert to a sequential baseline or an elision, as appropriate, at small instances) in any helper functions you write.

**Hint:** You may find it useful to extend your `merge` function to define a function

```
void merge_assoc(const sparray& xs, const sparray& ys, sparray& res,
               const sparray& xas, const sparray& yas,
               sparray& xares, sparray &yares,
               long reslo, long Alo, long Ahi, long Blo, long Bhi)
```

which merges `xs` and `ys`, and also sets `xares[i]` to `xas[i]` if `res[i]` was taken from `xs` or to `-1` if `res[i]` was taken from `ys` (and similar for `yares[i]`).

You may also want to use `scan_excl` (whose semantics match the scan given in class) and/or `scan_incl`, which *does not* include the identity as the first element of the resulting sequence and includes the final result as the last element of the sequence instead of as a separate value (so the sequence returned by `scan_incl` is still the same length as the input sequence). The work and span are the same as those of `scan`,  $O(n)$  and  $O(\log n)$ , respectively.

## 4.5 Testing

In `sandbox.cpp`, you can fill in `merge_sandbox` and `skyline_sandbox` with whatever testing code you'd like. We've included some to start you off.

When testing, you'll probably want to compile with debug on:

```
make sandbox.dbg
```

You can then run your sandbox code with:

```
./sandbox.dbg -sandbox FUNCTION
```

where `FUNCTION` is either `merge` or `skyline`.

## 4.6 Evaluation

When you're done testing, compile and the sequential baseline and parallel version, following the instructions in Chapters 8.4 and 8.6 of TAPP (with `sandbox` instead of `bench`).

**Task 8** [5] *Following the instructions in Chapter 8.7 of TAPP, use `prun` and `pplot` to run your `skyline` code on (at least) 1, 5, 10, 15 and 20 processors, along with the baseline, and generate a speedup curve. Marvel at how simple the `pbench` tool makes this.*

It's OK if your speedup curve doesn't look great; the point is to explore parallel algorithms, not to spend a long time optimizing.

**Task 9** [5] *Briefly explain what the speedup curve says about the parallelism of your `skyline` code. If your curve doesn't look as good as you had hoped, suggest why this might be the case.*