Outline of this lecture:

1. Parallelizing mergesort
2. Work stealing

1 Mergesort

You will work on two different implementations of parallel mergesort in project 1, one using Cilk Plus and one using pthreads. We will begin our study of the problem in this lecture. Pseudocode will follow in the project description. When doing the project, it is also important to think about design choices, such as in-place sort versus returning a different array from the input array. Please include a discussion in the writeup.

An Initial Approach

Here is an initial approach we could take to mergesort, which closely resembles a serial implementation like what would be covered in an introductory data structures and algorithms class such as CSE 241:

\[
\text{Mergesort}(A, \text{begin, end})
\]

1. if \( \text{begin} < \text{end} \)
2. \hspace{1em} then \( \text{mid} \leftarrow \left\lceil \frac{\text{end} + \text{begin}}{2} \right\rceil \)
3. \hspace{1em} \( A_1 = \text{spawn} \) Mergesort(\( A \), begin, mid)
4. \hspace{1em} \( A_2 = \text{Mergesort}(A, \text{mid, end}) \)
5. \hspace{1em} sync
6. \hspace{1em} \( A' = \text{Merge}(A_1, \text{begin, mid, } A_2, \text{mid, end}) \)
7. \hspace{1em} return \( A' \)
8. \hspace{1em} else // base case: one element
9. \hspace{1em} return \( A \)

In this case, of course, merge() is a helper function that merges two sorted arrays. We want to merge \( A_1 \) and \( A_2 \), which are the sorted first (begin to mid) and second (mid to end) halves of the input array \( A \). As with serial mergesort, we can do this by having a pointer that points to the smallest non-added elements in each of the two arrays to be merged, adding the smaller of the two and advancing the respective pointer.
What is the time complexity of this implementation? We can calculate the work and span using the Master Theorem (see an introductory course in data structures and algorithms such as CSE 241 for details):

\[
T_1(n) = 2T_1\left(\frac{n}{2}\right) + \Theta(n), \text{ which solves to } \Theta(n \lg n) \quad \text{(Work)}
\]

\[
T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + \Theta(n), \text{ which solves to } \Theta(n) \quad \text{(Span)}
\]

Thus, we see that the parallelism is \(\Theta\left(\frac{n \lg n}{n}\right) = \Theta(\lg n)\).

**Getting More Parallelism**

In order to get more parallelism, we would like to decrease the span. We describe a new way of implementing mergesort that yields more parallelism intuitively at a high level; pseudocode will follow in the project description. Specifically, we are going to focus on the merge() routine; how should we combine two sorted arrays into one to reduce the span and increase the parallelism?

Our basic problem: given two sorted arrays \(A_1\) and \(A_2\), where without loss of generality \(A_1\) is the size of \(A_2\) or bigger, we want to merge the elements of these arrays into one combined and still sorted array.

Our high-level approach: we will partition both arrays around an element \(x\), such that elements in the left portions of \(A_1\) and \(A_2\) (denote them by \(A_{1L}\) and \(A_{2L}\)) are at most \(x\), and the elements in the right portions of \(A_1\) and \(A_2\) (denote them by \(A_{1R}\) and \(A_{2R}\)) are at least \(x\). Why is this? We then want to merge \(A_{1L}\) and \(A_{2L}\), which gives us a sorted array \(A_L\) of elements that are all at most \(x\), and \(A_{1R}\) and \(A_{2R}\), which gives us a sorted array \(A_R\) of elements that are all at least \(x\). This means that all the elements in \(A_L\) are less than or equal to all the elements in \(A_R\), and since these arrays are already sorted, we can just append them together to get the complete sorted array. This process can be done recursively, breaking an input array into portions around a partition element \(x\), recursively breaking those portions into subportions, etc., down to a base case that consists of a single element (which is of course already sorted).

Now we must decide how to choose \(x\). We choose \(x\) to be the midpoint of our larger array, which we said without loss of generality was \(A_1\). Then \(A_{1L}\) is essentially the left half of the array, and \(A_{1R}\) is the right half of the array. When we partition \(A_2\) around \(x\), we divide \(A_2\) into portions \(A_{2L}\) and \(A_{2R}\). Note that we don’t know how big these portions are. For instance, if the numbers in \(A_2\) were all much larger than the numbers in \(A_1\), \(A_{2L}\) might be an empty array if none of its elements were smaller than \(x\), but the opposite might also be true, or anything in between.

From looking at the size of the portions of \(A_1\) alone, however, we can achieve some important bounds. If there are \(n\) elements total to be in our sorted array that consists of \(A_1 \cup A_2\), then there must be at least \(\frac{n}{2}\) elements in \(A_1\). Thus, there are at least \(\frac{n}{4}\) elements in \(A_L\) and \(A_R\), respectively, because one of their input, \(A_{1L}\) and \(A_{1R}\) respectively, contains at least \(\frac{n}{4}\) elements. With this approach, then, the resulting smaller array has at least \(\frac{n}{4}\) elements and the resulting larger array has at most \(\frac{3n}{4}\) elements.

Analyzing the complexity of this merge routine, we let \(\alpha\) be the fraction of the total number of elements that are in one of the two arrays to be merged. The important point, which we will use in
bounding the span, is that by the previous argument \( \alpha \leq \frac{3}{4} \). Thus, we get the work and the span of merge() as we have parallelized it above:

\[
T_1(n) = T_1(\alpha n) + T_1((1 - \alpha)n) + \Theta(\lg n), \text{ which solves to } \Theta(n) \quad \text{(Work)}
\]

\[
T_\infty(n) \leq T_\infty \left( \frac{3n}{4} \right) + \Theta(\lg n), \text{ which solves to } \Theta(\lg^2 n) \quad \text{(Span)}
\]

So the complexity of the overall mergesort routine is:

\[
T_1(n) = 2T_1 \left( \frac{n}{2} \right) + \Theta(n), \text{ which solves to } \Theta(n \lg n) \quad \text{(Work)}
\]

\[
T_\infty(n) = T_\infty \left( \frac{n}{2} \right) + \Theta(\lg^2 n), \text{ which solves to } \Theta(\lg^3 n) \quad \text{(Span)}
\]

The parallelism is thus \( \Theta \left( \frac{n \lg n}{\lg^3 n} \right) = \Theta \left( \frac{n}{\log^2 n} \right) \), which for large \( n \) is greater than the log \( n \). Thus, for large \( n \) (which of course is the case for problems that we would consider large enough to take the trouble to parallelize), this represents an increased parallelism over our initial approach.

As mentioned before, pseudocode corresponding to this high-level intuitive sketch will be coming out in Project 1, where you will have to implement this approach with both pthreads and Cilk Plus. Try to get as much parallelism as you can with the pthread implementation, but with Cilk Plus you should try to get the maximum amount of parallelism.

2 Work Stealing

Recall that the Cilk command “spawn” may or may not actually launch cause things to be executed in parallel. It simply makes that option available, but the decision of when to execute things in parallel is made by a scheduler at runtime. Here we analyze the means by which a scheduler tries to schedule tasks as efficiently as possible. Note that an efficient scheduling mechanism is important: if the computation has ample parallelism, you are still not going to see good speedup if you use an inefficient scheduler. An efficient scheduler will load balance the computation across available threads as evenly as possible while incurring small scheduling overhead.

In the previous lecture, we analyzed the behavior of a greedy scheduler, and we showed that a greedy scheduler guarantees the following time bound. For a computation with \( T_1 \) work and \( T_\infty \) span, a greedy scheduler can execute the computation on \( P \) processors in time

\[
T_P \leq \frac{T_1}{P} + T_\infty
\]

But note that in this bound, we don’t assume any scheduling overhead. The greedy scheduler somehow has the information of which nodes are ready and can be executed, and assign them to processors without overhead. We are now going to look at a more sophisticated scheduling algorithm called work stealing, and we will analyze it accounting scheduling overhead. In particular, a
work-stealing scheduler can execute a computation with $T_1$ work and $T_\infty$ span on $P$ processors in expected time:

$$T_P \leq \frac{T_1}{P} + O(T_\infty)$$

and uses space:

$$S_P \leq S_1 \cdot P$$

where $S_1$ is the stack space used during serial execution (or, execution of its serial counterpart).

At system startup, the runtime creates $P$ threads, called the workers, and the runtime system will load balance the computation across these $P$ workers. Each worker employs a data structure called deque, or double-ended queue, to keep track of what work items / tasks it is responsible for.

Whenever a worker executes a function $A$ that spawns a function $B$, the worker continues to execute the spawned child, $B$, and pushes the continuation of the parent, $A$, onto the bottom of the deque. When the worker finishes executing $B$, it pops what’s on the bottom of the deque and resumes that (in this case, the continuation of $A$).

For the most part, a worker pushes and pops frames off the deque like a stack. When a worker runs out of work (i.e., its deque is empty), however, it turns into a thief and steals work items / tasks from another worker. This means it randomly chooses a victim (another worker) from whom to steal a task, and a thief always steals from the top of a victim’s deque. If the steal is successful, the thief takes over the responsibility of executing the stolen task — it resumes the execution of that task. Note that, because the victim is chosen randomly, the steal could fail, for instance, if the victim’s deque is empty, in which case, the thief will choose a different victim randomly and try again.

Thus, the steps a worker in a work-stealing scheduler may perform are as follows.

1. **Spawn:** say $A$ spawns $B$; the worker pushes $A$ onto the bottom of its deque and continues to execute $B$.

2. **Return:** say $B$ is returning back to its parent $A$.

   There are a few possible scenarios:

   (a) The worker pops $A$ off the bottom of its deque and resumes $A$.

   (b) $A$ is stolen, in which case the deque is empty. Note that, if $A$ is stolen, everything above $A$ must have been stolen as well. Because a thief always steals from the top of a victim’s deque, tasks above $A$ in the deque must have been stolen before $A$ can be stolen.

   If $A$ is stolen, the worker tries to see if $A$ can be resumed.

   - If yes, $B$ is the last child returning, so it takes $A$ back and resumes execution of $A$

   - If no, $A$ is being executed still by some other worker, so this worker turns into a thief and performs random work stealing to find more work to do.

3. **Sync:** the worker executes a sync statement in function $A$. 

4
• If there is any outstanding child of $A$ executing on another worker (i.e., $A$ has been stolen before), the sync fails, so the worker suspends $A$. (When $A$ is suspended, it is put aside in limbo, i.e., not on anyone’s deque. It goes onto some worker’s deque when it’s ready to be resumed.)

• Otherwise, sync succeeds, and the worker resumes $A$.

4. **Steal**: as described above.

    In class, we worked out simple examples to illustrate these processes.

    A few things to note. First, when a thief successfully steals a function $A$ from a victim, that’s when parallelism is realized — because the thief subsequently resumes the continuation of $A$ while the victim continues to executes the spawned child that causes $A$ to be pushed onto the deque. Second, this style of execution is called **work-first**, where a spawn statement causes the worker to execute the spawned child first, while pushing the continuation of the parent onto the deque, making it available to be stolen. One could imagine a different approach of scheduling, called **help-first**, where the worker executes the continuation, but packages the spawned child as an work item to push onto the deque, allowing the spawned child to be stolen.

    Cilk’s work-stealing scheduler chooses the work-first scheduling, due to the following advantages:

    1. Serial semantics: if you execute the program with work-first scheduling on a single worker, you complete the tasks in the same order as you would with the execution of its serial counterpart (i.e., a serial program resulted from removing all parallel keywords).

    2. Space bound: we can get a lower space bound for work-first scheduling than help-first scheduling. Imagine a for loop that spawns off a thousand tasks. A help-first strategy executing on one worker will create a thousand tasks and push them onto the deque before executing any of them.

    3. Locality: with work-first scheduling and a private cache, the cache complexity is similar to that of serial execution, except when a worker steal. Thus, only additional cache misses are incurred whenever successful steals occur. We won’t really get into this and mention it only in passing, but it’s good to be aware of.