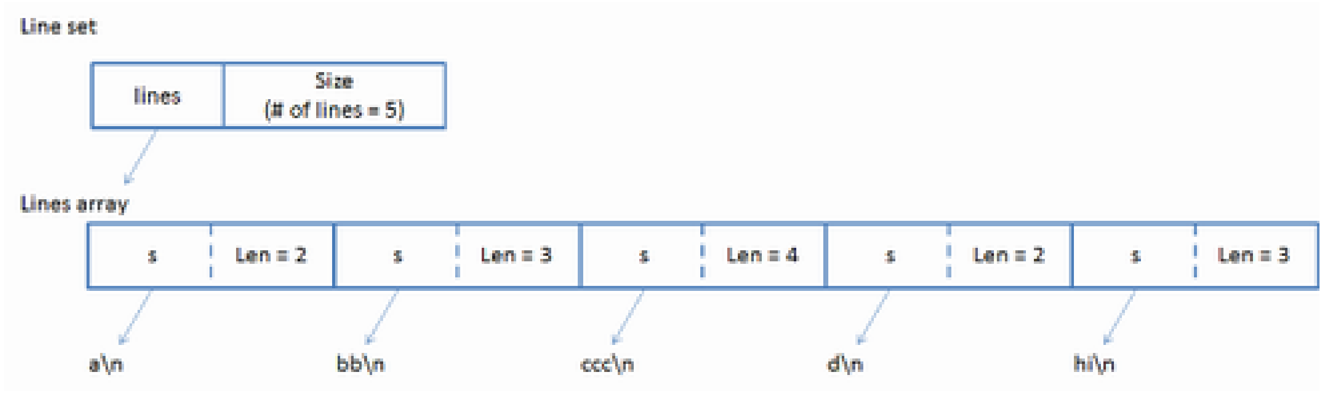
Julia Carvalho, Neil Gat, Y. Jeanette Park
Tuesday, December 4, 2012

CS 61 Notes- Lecture 25

Topics:
- Concurrency
- Parallelism speedup
- Pthread mutexes & locks

Lecture repository: 25

> Lineset: an array of lines, where each line is a pointer to some data (e.g. sequence of characters with a new-line “\n” between them; included with this sequence is the length of the sequence of characters)



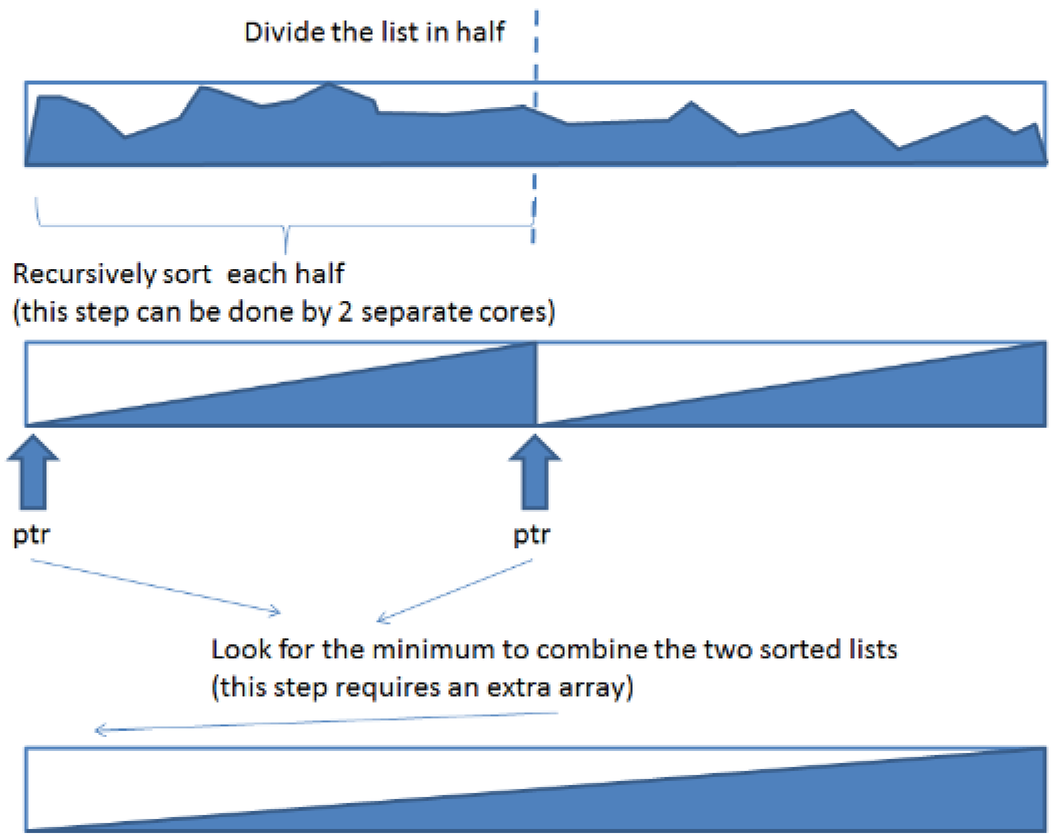
> sort01, using Qsort in the C standard library, takes ~4s; Q-sort generally assumed to be fastest (nlog(n) in the best/average case, n^2 in the worst case) sorting algorithm on a single core
> system sort takes ~2s

> Q: Why is system sort faster? hint: over 200% CPU usage when running system sort
> > A: system sort uses multiple cores; sort01 uses only 1 core

interesting sidenote about the time unix function:
1st number (x.xxu): time spent by user application .
2nd number (x.xxs): the time it took in the kernel, ie to read in the file
3rd number: 1st number + 2nd number (clock time)
% number: CPU usage. If over 100%, it’s using multiple cores.

Lightening quick review of mergesort:
> Array of things to sort. Mergesort: divide and conquer.

1. Divide array into two arrays (x[100] = x_1[0-49], x_2[50-100])
2 Recursively sort both halves. (which can be done in parallel)
3. Merge by copying with comparisons (x[0] = min(x_1[0],x_2[50]))


Divide the list in half
Recursively sort each half
(this step can be done by 2 separate cores)
ptr
ptr
Look for the minimum to combine the two sorted lists
(this step requires an extra array)

Problems? Step 3 requires a spare array- e.g., requires o(N) extra space (Why? Imagine that all of the elements in the first half (x_1) are larger than the elts in x_2)

> Why we like merge sort: can easily parallelize: just do the two halfs in different threads!
> > pseudo-code for merge sort:

```
1   split array;
2   sort left;
3   sort right
4   merge;
```

Lines 2 and 3 can be done in parallel

>Sort02.c: uses a merge sort algorithm (not parallelized): does pretty well

> pthread_create(thread_id, NULL (thread attributes; ignore), thread_function, arg_to_thread_function)

Try to run, get seg fault: what happened?
Synchronization problem: multiple threads running in parallel -> race condition. Threads executing in an unexpected order. Why? Merging lines that haven't been sorted yet. When child thread exits, THEN we know that the child has finished. So use pthread_join with child threads.

What is pthread_join? Glad you asked:
pthread_join:

- -blocks until thread exits
- -returns thread exit value

e.g. (in pseudo-code)

```
{
left = pthread_create(blah)
right = pthread_create(blah2)
merge(left, right)
```

} → this code will break because we need to wait till left and right finish before merging. Correct version is:

```
{
left = pthread_create(blah)
right = pthread_create(blah2)
pthread_join(right, left)
pthread_join(left, right)
merge(left, right)
```

} → this code is still a problem:

Our sort (using threads instead of recursion): 4.04 seconds. Still not doing so hot.

Hm. How parallel is it? How many threads? Depends on threshold value and input size. Proportional to the size of the input. The main thread that creates two more just blocks until the other threads go. This is just busy-waiting!

Let's fix this:
Create one child thread for the right half and just reuse the parent one for the left thread. This will cut the number of threads in half. What does this do for performance?
Time: 3.9s -- better by .1s

Let's track the number of threads... Eddie's hypothesis is that this is running a tremendous number of threads and that's why it's running slower

How to track number of threads? Create global variables: the current number of threads, the total number of threads we've ever seen, the maximum number of threads that we've seen running at the same time.

Critical section: find the shared variables that are getting modified by more than one thread (e.g the counter variables).
Solution: add locks (read about locks on the man page)
Anytime you're about to enter a critical section, get the lock. Anytime you're about to leave, release the lock.

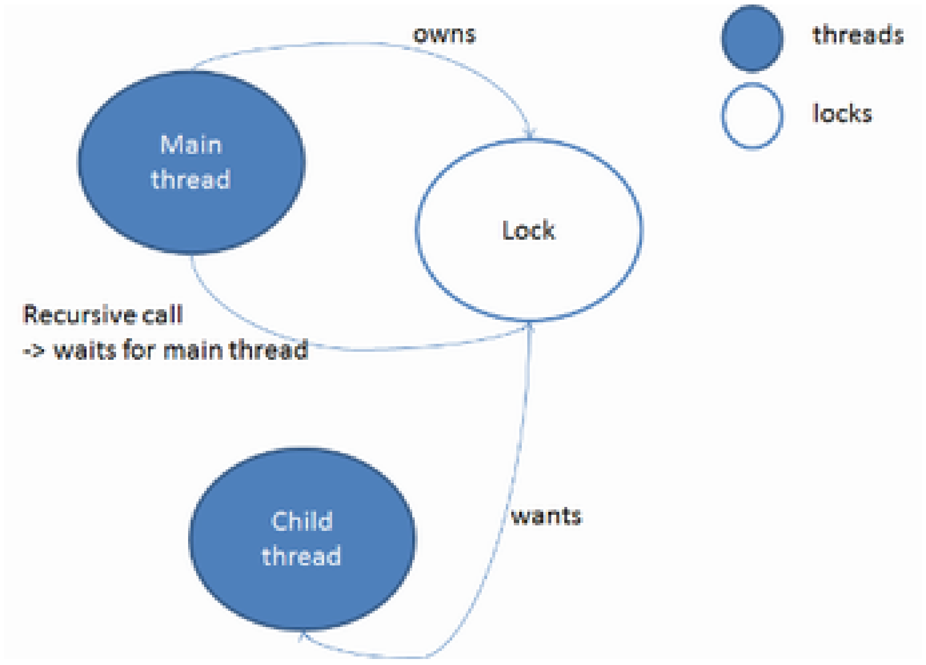
To use the lock on a critical section:
“pthread_mutex_lock(&lock)” (before entering the critical section)
“pthread_mutex_unlock(&lock)” (after the critical section)

→ this didn’t help on the speed very much

Challenge of parallel programming: programming in such a way to prevent “heisen-bugs” : bugs (like race conditions) that can not appear for millions and millions of runs and slip by your debugging

you can try to make sure you don’t miss anything by making the critical section really big (e.g., lock at beginning and unlock at end of function)... but this makes program run super slow (/ infinitely) because the main thread dominates the lock (called a circular wait)

Circular wait: when the main thread owns the lock and is waiting on a child thread to return, but the child thread won’t complete until it gets the locked: **deadlock**! **dun dun dun...** this is a risk of having too large of a ‘critical section’ (remember, critical section: the mutual section that you surround with locks)

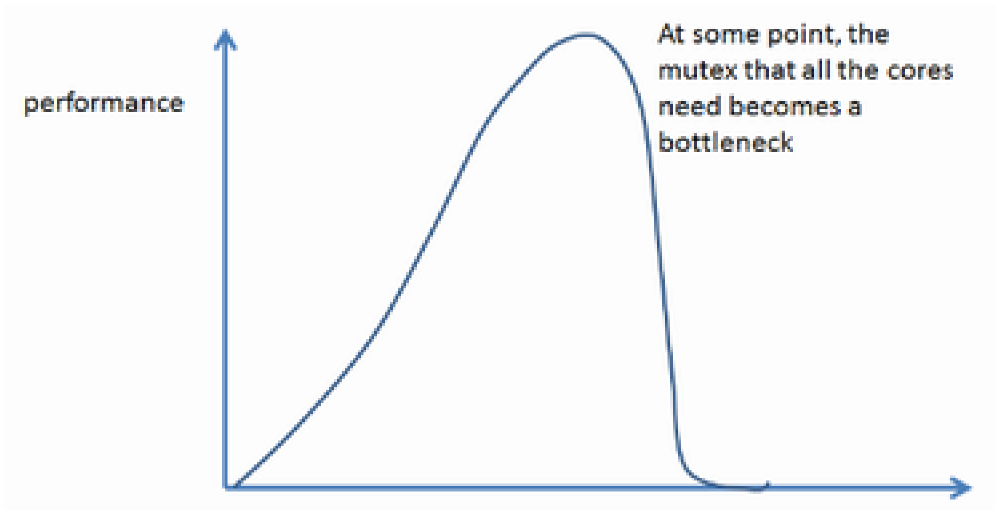


We currently have 4095 threads over 48 cores -- that’s a lot. Computer doesn’t know how to prioritize among these many threads and switches many times... costly. Let’s limit it to 48 threads.

>sort08.c

Time with max of 68 threads: 2.9s
Time with max of 48 threads: 2.88s. Very close to the system's 'sort' function! Got there by not overloading each core

Possible bottlenecks in code:
>Contention (waiting on locks): too many threads trying to acquire the same exclusive (mutex) lock. Each thread waits proportional to the number of threads.



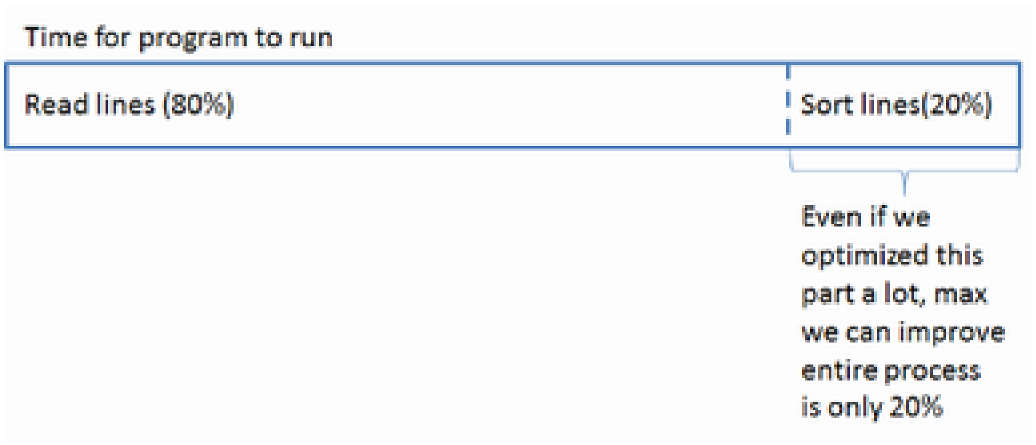
After a certain point, adding cores causes the performance to crash. (lots of time to go from one core to the next because the mutex that all the cores need becomes bottleneck
Solution to contention problems: Try to remove the locks (try to work without having to use locks) or minimize the execution time of the code in critical sections.

In our code, critical section executes quickly and we cap the number of threads, so contention is likely not the problem

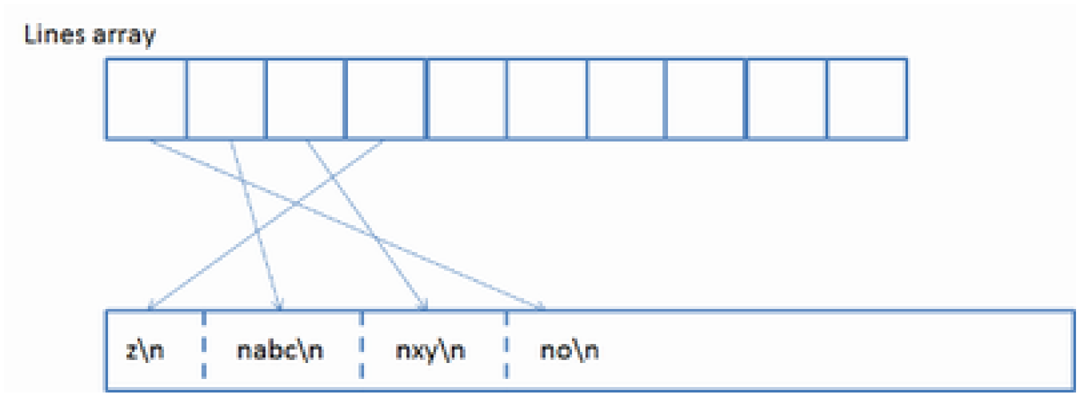
>Imbalance: One thread (or a small number of threads) takes much longer to complete; NOT likely to be the problem here because we separate problem into ROUGHLY same-sized sub-problems (max difference in size is factor of 2)

What else could be going slowly in our program if Imbalance and Contention are likely NOT issues? Perhaps some other function we call is slow? split_lines might be slow. Which brings us to:

Amdahl's law: (see book for equational form)
The performance benefit of an optimization to part of a program is limited by the overall performance of the rest of the program.


Time for program to run
Read lines (80%)
Sort lines(20%)
Even if we optimized this part a lot, max we can improve entire process is only 20%

Another potential constraint: cache use? specifically, a line points to a word. These words are basically random access, so the cache can't handle it well. Otherwise, this pattern is more or less sequential.



You can do this better by putting the first N letters of each word in a hash table?

What comes next after CS61? Systems are important. Try enrolling in:

CS161 -- Operating Systems "it has a certain reputation...(class nervously giggles)"

CS 205, 207: parallelism related

Review next week. That's it!