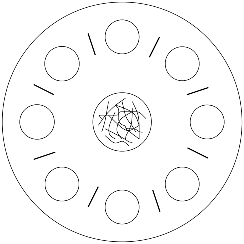
Another old and famous problem: the *Dining Philosophers*

Another old and famous problem: the *Dining Philosophers*

Often used to illustrate problems of resource contention in operating systems, it can be used to help understand problems in concurrency, too



Dining Philosophers

We have five philosophers wanting to eat spaghetti, but there are only five chopsticks to go round

The life of a philosopher is

- think
- sit
- take chopsticks
- eat
- drop chopsticks
- leave
- repeat

A philosopher sits at any free position, but can only use the two neighbouring chopsticks

A philosopher sits at any free position, but can only use the two neighbouring chopsticks

They require two chopsticks to be able to eat!

Parallel Algorithms

Dining Philosophers

A philosopher sits at any free position, but can only use the two neighbouring chopsticks

They require two chopsticks to be able to eat!

If a chopstick is already in use, the philosopher must wait until it is free

This problem shows

Parallel Algorithms

Dining Philosophers

This problem shows

- mutual exclusion of the chopsticks

This problem shows

- mutual exclusion of the chopsticks
- deadlock if all the philosophers sit down simultaneously and grab the left chopstick: they will all then have to wait on their right chopstick

This problem shows

- mutual exclusion of the chopsticks
- deadlock if all the philosophers sit down simultaneously and grab the left chopstick: they will all then have to wait on their right chopstick
- starvation, as four of the philosophers might conspire to keep out the fifth

Mutual exclusion of the chopsticks is easily provided by having a mutex for each chopstick

```
lock chopstick[5];
```

Mutual exclusion of the chopsticks is easily provided by having a mutex for each chopstick

```
lock chopstick[5];
```

Then philosopher *i* grabbing and dropping the chopsticks is

Mutual exclusion of the chopsticks is easily provided by having a mutex for each chopstick

```
lock chopstick[5];
```

Then philosopher *i* grabbing and dropping the chopsticks is

```
lock(chopstick[i]);
lock(chopstick[(i+1)%5]);
eat();
unlock(chopstick[(i+1)%5]);
unlock(chopstick[i]);
```

But, as we know, this can deadlock if all philosophers grab (say) the left chopstick simultaneously

But, as we know, this can deadlock if all philosophers grab (say) the left chopstick simultaneously

Simply alternating left-then-right grab with right-then-left grab won't fix it; neither will picking a random chopstick first

Parallel Algorithms

Dining Philosophers

But, as we know, this can deadlock if all philosophers grab (say) the left chopstick simultaneously

Simply alternating left-then-right grab with right-then-left grab won't fix it; neither will picking a random chopstick first

The classical solution is to have a counting semaphore, initialised to 4, to limit the number of simultaneously sitting philosophers

Parallel Algorithms

Dining Philosophers

```
lock chopstick[5];
place = make_counting_semaphore(4);
...
philosopher(int i) {
  while (1) {
    think();
    wait(place);
    lock(chopstick[i]);
    lock(chopstick[(i+1)%5]);
    eat();
    unlock(chopstick[(i+1)%5]);
    unlock(chopstick[i]);}
    signal(place);
  }
}
```

Exercise Prove this cannot deadlock

Exercise Think about fixing starvation

Exercise Solve the Dining Philosophers using monitors

Exercise Solve the Dining Philosophers using GCD

Parallel Algorithms

Sorting

We now turn to some concrete examples of parallel algorithms, beginning with sorting

We now turn to some concrete examples of parallel algorithms, beginning with sorting

Clearly, a merge sort is amenable to divide and conquer

We now turn to some concrete examples of parallel algorithms, beginning with sorting

Clearly, a merge sort is amenable to divide and conquer

- divide data into two equal chunks
- recursively merge sort each half in parallel
- merge the two sorted lists together

For example, $n = 8$. The division is trivial, so we concentrate on the merge:

								t	p
3	1	4	1	5	9	2	6		

t is the time to merge sort that line; p the number of processors

For example, $n = 8$. The division is trivial, so we concentrate on the merge:

								t	p
3	1	4	1	5	9	2	6		
1	3	1	4	5	9	2	6	**2**	4

t is the time to merge sort that line; p the number of processors

For example, $n = 8$. The division is trivial, so we concentrate on the merge:

								t	p
3	1	4	1	5	9	2	6		
1	3	1	4	5	9	2	6	**2**	4
1	1	3	4	2	5	6	9	**4**	2

t is the time to merge sort that line; p the number of processors

For example, $n = 8$. The division is trivial, so we concentrate on the merge:

								t	p
3	1	4	1	5	9	2	6		
1	3	1	4	5	9	2	6	**2**	4
1	1	3	4	2	5	6	9	**4**	2
1	1	2	3	4	5	6	9	**8**	1

t is the time to merge sort that line; p the number of processors

Parallel Algorithms
Sorting

For example, $n = 8$. The division is trivial, so we concentrate on the merge:

								t	p
3	1	4	1	5	9	2	6		
1	3	1	4	5	9	2	6	**2**	4
1	1	3	4	2	5	6	9	**4**	2
1	1	2	3	4	5	6	9	**8**	1
							Total:	**14**	

t is the time to merge sort that line; p the number of processors

It is easy to calculate the time this takes on n values (PRAM: assume we have enough processors and ignore communications costs)

It is easy to calculate the time this takes on n values (PRAM: assume we have enough processors and ignore communications costs)

- The last merge takes time n

It is easy to calculate the time this takes on n values (PRAM: assume we have enough processors and ignore communications costs)

- The last merge takes time n
- The step before takes time $n/2$ (twice, in parallel)

It is easy to calculate the time this takes on n values (PRAM: assume we have enough processors and ignore communications costs)

- The last merge takes time n
- The step before takes time $n/2$ (twice, in parallel)
- The step before takes time $n/4$ (four times, in parallel)

It is easy to calculate the time this takes on n values (PRAM: assume we have enough processors and ignore communications costs)

- The last merge takes time n
- The step before takes time $n/2$ (twice, in parallel)
- The step before takes time $n/4$ (four times, in parallel)
- etc.

It is easy to calculate the time this takes on n values (PRAM: assume we have enough processors and ignore communications costs)

- The last merge takes time n
- The step before takes time $n/2$ (twice, in parallel)
- The step before takes time $n/4$ (four times, in parallel)
- etc.

Total time is $T(n) = n + n/2 + n/4 + \cdots + 2 = 2n - 2 = O(n)$

The sequential merge sort takes time $O(n \log n)$, giving a speedup of

$$S = O(n \log n / n) = O(\log n)$$

using $O(n)$ processors ($n/2$ in this case)

The sequential merge sort takes time $O(n \log n)$, giving a speedup of

$$S = O(n \log n / n) = O(\log n)$$

using $O(n)$ processors ($n/2$ in this case)

This increases with n, but not very quickly, and is a lot smaller than n

The sequential merge sort takes time $O(n \log n)$, giving a speedup of

$$S = O(n \log n / n) = O(\log n)$$

using $O(n)$ processors ($n/2$ in this case)

This increases with n, but not very quickly, and is a lot smaller than n

It uses $O(n)$ processors, for an efficiency of

$$E = O(\log n / n)$$

The sequential merge sort takes time $O(n \log n)$, giving a speedup of

$$S = O(n \log n/n) = O(\log n)$$

using $O(n)$ processors ($n/2$ in this case)

This increases with n, but not very quickly, and is a lot smaller than n

It uses $O(n)$ processors, for an efficiency of

$$E = O(\log n/n)$$

The efficiency drops to 0 as n gets large

If we have just p processors, this becomes

$$T_p(n) = O\left(n + \frac{n}{p}\log\frac{n}{p}\right)$$

as we have sequential merge sorts of p chunks of size n/p, plus $(n/p)O(p) = O(n)$ steps to merge them in parallel

If we have just p processors, this becomes

$$T_p(n) = O\left(n + \frac{n}{p}\log\frac{n}{p}\right)$$

as we have sequential merge sorts of p chunks of size n/p, plus $(n/p)O(p) = O(n)$ steps to merge them in parallel

We get

$$S_p(n) \approx p$$
$$E_p(n) \approx 1$$

for large n and fixed p

If we have just p processors, this becomes

$$T_p(n) = O\left(n + \frac{n}{p}\log\frac{n}{p}\right)$$

as we have sequential merge sorts of p chunks of size n/p, plus $(n/p)O(p) = O(n)$ steps to merge them in parallel

We get

$$S_p(n) \approx p$$
$$E_p(n) \approx 1$$

for large n and fixed p

Exercise Work this example through for yourself

So: for a fixed number of processors we can get good a speedup, but if we let the number of processors get large our relative speedup gets quite poor

Parallel Algorithms

Sorting

So: for a fixed number of processors we can get good a speedup, but if we let the number of processors get large our relative speedup gets quite poor

Seems counterintuitive until you think about it, but it means we have to have lots of data relative to the number of processors to get a good speedup

So: for a fixed number of processors we can get good a speedup, but if we let the number of processors get large our relative speedup gets quite poor

Seems counterintuitive until you think about it, but it means we have to have lots of data relative to the number of processors to get a good speedup

Alternatively: if we have a lot of processors, most of them are going to be idle most of the time: we only use all of them in the first step; and even fewer in subsequent steps

So: for a fixed number of processors we can get good a speedup, but if we let the number of processors get large our relative speedup gets quite poor

Seems counterintuitive until you think about it, but it means we have to have lots of data relative to the number of processors to get a good speedup

Alternatively: if we have a lot of processors, most of them are going to be idle most of the time: we only use all of them in the first step; and even fewer in subsequent steps

Exercise Think about this result in the context of Amdahl and Gustafson

The most famous sequential sort (after bubble) is *quicksort*

The most famous sequential sort (after bubble) is *quicksort*

Similar to mergesort, in that it is a divide and conquer method, but different in how it divides

Parallel Algorithms

Sorting

The most famous sequential sort (after bubble) is *quicksort*

Similar to mergesort, in that it is a divide and conquer method, but different in how it divides

- pick a value, the *pivot*, from the data
- partition the data into two chunks: values bigger than the pivot; values less than the pivot
- recursively quicksort the two chunks
- return the sorted lower chunk; the pivot; the sorted higher chunk

The partition phase is a bit fiddly to parallelise, but the recursive sorts are clearly parallelisable

The partition phase is a bit fiddly to parallelise, but the recursive sorts are clearly parallelisable

It works well with manager/worker: as each sub-partition is created it becomes a new task

The partition phase is a bit fiddly to parallelise, but the recursive sorts are clearly parallelisable

It works well with manager/worker: as each sub-partition is created it becomes a new task

Also, the tasks are entirely independent with no communications between them once created

The partition phase is a bit fiddly to parallelise, but the recursive sorts are clearly parallelisable

It works well with manager/worker: as each sub-partition is created it becomes a new task

Also, the tasks are entirely independent with no communications between them once created

Though we do need to join the sorted partitions back together

Parallel quicksort is very similar in time complexity to mergesort: it takes time $O(n)$ with $O(n)$ processors in the average case

Parallel quicksort is very similar in time complexity to mergesort: it takes time $O(n)$ with $O(n)$ processors in the average case

And time $O(n + (n/p)\log(n/p))$ with p processors

Parallel quicksort is very similar in time complexity to mergesort: it takes time $O(n)$ with $O(n)$ processors in the average case

And time $O(n + (n/p)\log(n/p))$ with p processors

As usual, quicksort relies on decent pivots: this translates directly to the need to get good load balancing of the sub-tasks

Heapsort: another $O(n \log n)$ (sequential) sort, is valued as it has very stable behaviour: no bad cases

Heapsort: another $O(n \log n)$ (sequential) sort, is valued as it has very stable behaviour: no bad cases

But there doesn't seem to be a good way of parallelising it as the swaps in the heap creations and destructions need to pass in unpredictable ways through the entire dataset

Bucket sort parallelises well: this splits the data into several buckets, then recursively sorts the buckets

Bucket sort parallelises well: this splits the data into several buckets, then recursively sorts the buckets

Example. Sorting CDs. Have one bucket per letter of the alphabet. It is quick to put CDs in the correct buckets

Bucket sort parallelises well: this splits the data into several buckets, then recursively sorts the buckets

Example. Sorting CDs. Have one bucket per letter of the alphabet. It is quick to put CDs in the correct buckets

Clearly, an extension of the merge sort, it has very similar properties

Parallel sorting algorithms exist that take parallel time $O(\log n)$, but require $O(n^2 / \log n)$ processors: very inefficient

Parallel sorting algorithms exist that take parallel time $O(\log n)$, but require $O(n^2/\log n)$ processors: very inefficient

Other sorts exist that take time $O(\log n)$ time and $O(n)$ processors: sounds better?

Parallel Algorithms

Sorting

Parallel sorting algorithms exist that take parallel time $O(\log n)$, but require $O(n^2/\log n)$ processors: very inefficient

Other sorts exist that take time $O(\log n)$ time and $O(n)$ processors: sounds better?

Some of these you need to be sorting upwards of 10^{22} items to be faster than simpler sorts with apparently worse complexities, like the *bitonic* sort, with time $O(\log^2 n)$

Parallel Algorithms

Sorting

The bitonic sort, a divide and conquer method somewhat related to merge sort and shell sort, takes time $O(\log^2 n)$ on $O(n)$ processors

The bitonic sort, a divide and conquer method somewhat related to merge sort and shell sort, takes time $O(\log^2 n)$ on $O(n)$ processors

It takes $O(n \log^2 n)$ sequentially, so having a speedup of $O(n)$

The bitonic sort, a divide and conquer method somewhat related to merge sort and shell sort, takes time $O(\log^2 n)$ on $O(n)$ processors

It takes $O(n \log^2 n)$ sequentially, so having a speedup of $O(n)$

This sounds good, until you realise this is a parallelisation of a slightly sub-optimal sequential sort

Parallel Algorithms

Sorting

The bitonic sort, a divide and conquer method somewhat related to merge sort and shell sort, takes time $O(\log^2 n)$ on $O(n)$ processors

It takes $O(n \log^2 n)$ sequentially, so having a speedup of $O(n)$

This sounds good, until you realise this is a parallelisation of a slightly sub-optimal sequential sort

Comparing against a $O(n \log n)$ fast sort, we see bitonic has speedup $O(n/\log n)$; still not too bad

The bitonic sort, a divide and conquer method somewhat related to merge sort and shell sort, takes time $O(\log^2 n)$ on $O(n)$ processors

It takes $O(n \log^2 n)$ sequentially, so having a speedup of $O(n)$

This sounds good, until you realise this is a parallelisation of a slightly sub-optimal sequential sort

Comparing against a $O(n \log n)$ fast sort, we see bitonic has speedup $O(n/\log n)$; still not too bad

But the important thing is that it is practical for realistic sizes of n

Parallel Algorithms

Sorting

The bitonic sort, a divide and conquer method somewhat related to merge sort and shell sort, takes time $O(\log^2 n)$ on $O(n)$ processors

It takes $O(n \log^2 n)$ sequentially, so having a speedup of $O(n)$

This sounds good, until you realise this is a parallelisation of a slightly sub-optimal sequential sort

Comparing against a $O(n \log n)$ fast sort, we see bitonic has speedup $O(n/\log n)$; still not too bad

But the important thing is that it is practical for realistic sizes of n

Exercise Go and read up on bitonic sort

And there are many other sorts

And there are many other sorts

The literature for parallel sorts is huge, as it is a problem that is easy to understand, but hard to solve

And there are many other sorts

The literature for parallel sorts is huge, as it is a problem that is easy to understand, but hard to solve

Particularly when you start to factor communications costs into your time complexities

Exercise It has been claimed that `MapReduce` can sort “a petabyte of data in a few hours”. Find out about how it does this

Exercise Related to sorting is the problem of finding the maximum value in a dataset. Discuss how this might be parallelised and its time complexity

Exercise Then find the middle value in a dataset

Parallel Algorithms

Searching

The other classical problem is searching

Parallel Algorithms

Searching

The other classical problem is searching

This is very datastructure dependent, but can parallelise very well

The other classical problem is searching

This is very datastructure dependent, but can parallelise very well

For example, if the data are spread over many machines, searching for an item is as simple as getting each machine to search its chunk

Parallel Algorithms

Searching

The other classical problem is searching

This is very datastructure dependent, but can parallelise very well

For example, if the data are spread over many machines, searching for an item is as simple as getting each machine to search its chunk

When any machine finds the item, they can all stop

The other classical problem is searching

This is very datastructure dependent, but can parallelise very well

For example, if the data are spread over many machines, searching for an item is as simple as getting each machine to search its chunk

When any machine finds the item, they can all stop

Or, if multiple results are wanted, there can be a reduce step

If the data is distributed sensibly over p processors, the chunks will be of size n/p and take n/p time to search for a naïve linear search

If the data is distributed sensibly over p processors, the chunks will be of size n/p and take n/p time to search for a naïve linear search

Thus parallel searching can give perfect speedup $n/(n/p) = p$

If the data is distributed sensibly over p processors, the chunks will be of size n/p and take n/p time to search for a naïve linear search

Thus parallel searching can give perfect speedup $n/(n/p) = p$

But linear search is far from a good sequential search

If the data is distributed sensibly over p processors, the chunks will be of size n/p and take n/p time to search for a naïve linear search

Thus parallel searching can give perfect speedup $n/(n/p) = p$

But linear search is far from a good sequential search

Again, we get a good speedup since we start from a poor place

Searching in a tree takes time $O(\log n)$, so if we can perfectly distribute sub-trees across p processors, we can search them in parallel time $O(\log(n/p))$ for a speedup $O(\log n/\log(n/p))$

Searching in a tree takes time $O(\log n)$, so if we can perfectly distribute sub-trees across p processors, we can search them in parallel time $O(\log(n/p))$ for a speedup $O(\log n/\log(n/p))$

Sounds good? Well, consider the speedup for large n:

$$\begin{aligned} O(\log n/\log(n/p)) &= O(\log n/(\log n - \log p)) \\ &= O(1/(1 - \log p/\log n)) \\ &\to 1 \text{ as } n \to \infty \end{aligned}$$

Parallel Algorithms

Searching

Searching in a tree takes time $O(\log n)$, so if we can perfectly distribute sub-trees across p processors, we can search them in parallel time $O(\log(n/p))$ for a speedup $O(\log n/\log(n/p))$

Sounds good? Well, consider the speedup for large n:

$$
\begin{aligned}
O(\log n/\log(n/p)) &= O(\log n/(\log n - \log p)) \\
&= O(1/(1 - \log p/\log n)) \\
&\to 1 \text{ as } n \to \infty
\end{aligned}
$$

Here the problem is that tree search is so good that the benefit you get from spreading it across p processors is small, and gets smaller as the dataset increases in size

And these algorithms rely on everything being nice and uniform and randomly accessible and ignoring communications costs

And these algorithms rely on everything being nice and uniform and randomly accessible and ignoring communications costs

For example, if the searches cluster around the data on a single machine, we could write a sequential search that takes advantage of that fact, and our parallel search would not be much faster

Also, the datastructure must be able to be evenly spread

Also, the datastructure must be able to be evenly spread

Lists and trees, that have restrictions on the order you access their elements, are harder to access in this random manner

Also, the datastructure must be able to be evenly spread

Lists and trees, that have restrictions on the order you access their elements, are harder to access in this random manner

Of course, Google does this in a big way, using MapReduce, showing that searching petabytes of data can be done in fractions of a second

Also, the datastructure must be able to be evenly spread

Lists and trees, that have restrictions on the order you access their elements, are harder to access in this random manner

Of course, Google does this in a big way, using MapReduce, showing that searching petabytes of data can be done in fractions of a second

Again, we find that parallelism allows us to go bigger, rather than faster

Next: parallel reduction

Next: parallel reduction

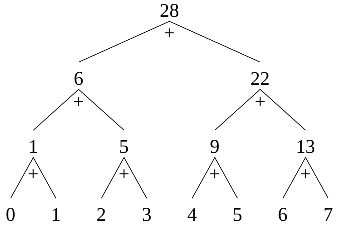
Reduction has a natural parallelisation using a tree

Parallel Algorithms

Reduction

Next: parallel reduction

Reduction has a natural parallelisation using a tree

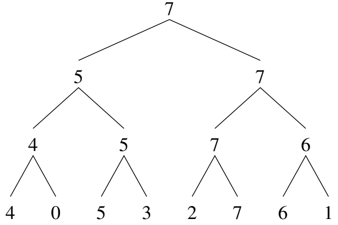


Tree reduction sum

Reducing a list of values using summation (read bottom up)

Next: parallel reduction

Reduction has a natural parallelisation using a tree



Tree reduction maximum

Reducing a list of values using maximum

This takes $O(\log n)$ steps to reduce n values, using $O(n)$ processors

This takes $O(\log n)$ steps to reduce n values, using $O(n)$ processors

Sequential time: $n - 1$ operations, giving speedup

$$S = O(n/\log n) \text{ using } O(n) \text{ processors}$$

This takes $O(\log n)$ steps to reduce n values, using $O(n)$ processors

Sequential time: $n-1$ operations, giving speedup

$$S = O(n/\log n) \text{ using } O(n) \text{ processors}$$

This is not much less than n, as $\log n$ grows only slowly with n

Efficiency

$$E = O(1/\log n)$$

which slowly drops as n increases

For p processors, divide the data into p chunks of size n/p

For p processors, divide the data into p chunks of size n/p

Time to reduce a chunk (sequential): $O(n/p)$
Time to reduce the chunks: $O(\log p)$

Parallel Algorithms

Reduction

For p processors, divide the data into p chunks of size n/p

Time to reduce a chunk (sequential): $O(n/p)$
Time to reduce the chunks: $O(\log p)$

Total

$$O\left(\frac{n}{p} + \log p\right)$$

Speedup

$$S_p = \frac{n}{n/p + \log p} = \frac{p}{1 + (p \log p)/n}$$

which approaches p as n gets large

Speedup

$$S_p = \frac{n}{n/p + \log p} = \frac{p}{1 + (p \log p)/n}$$

which approaches p as n gets large

Likewise, the efficiency approaches 1 for large n

Parallel Algorithms

Reduction

Speedup

$$S_p = \frac{n}{n/p + \log p} = \frac{p}{1 + (p \log p)/n}$$

which approaches p as n gets large

Likewise, the efficiency approaches 1 for large n

Similar to previous examples, if you allow yourself an indefinite number of processors, the speedup will be greater, but at a high cost, i.e., low efficiency

Speedup

$$S_p = \frac{n}{n/p + \log p} = \frac{p}{1 + (p \log p)/n}$$

which approaches p as n gets large

Likewise, the efficiency approaches 1 for large n

Similar to previous examples, if you allow yourself an indefinite number of processors, the speedup will be greater, but at a high cost, i.e., low efficiency

For a fixed number of processors, you get a fixed bound on the speedup, but you will be using the hardware very efficiently as the dataset get large