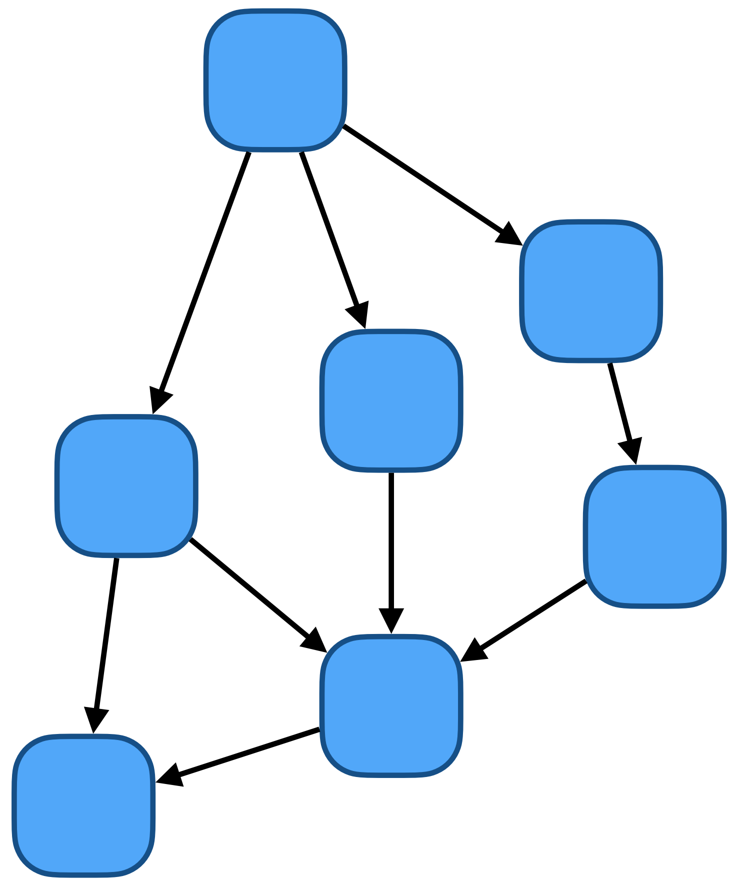
Utrecht University

B3CC: Concurrency

15: Work & Span

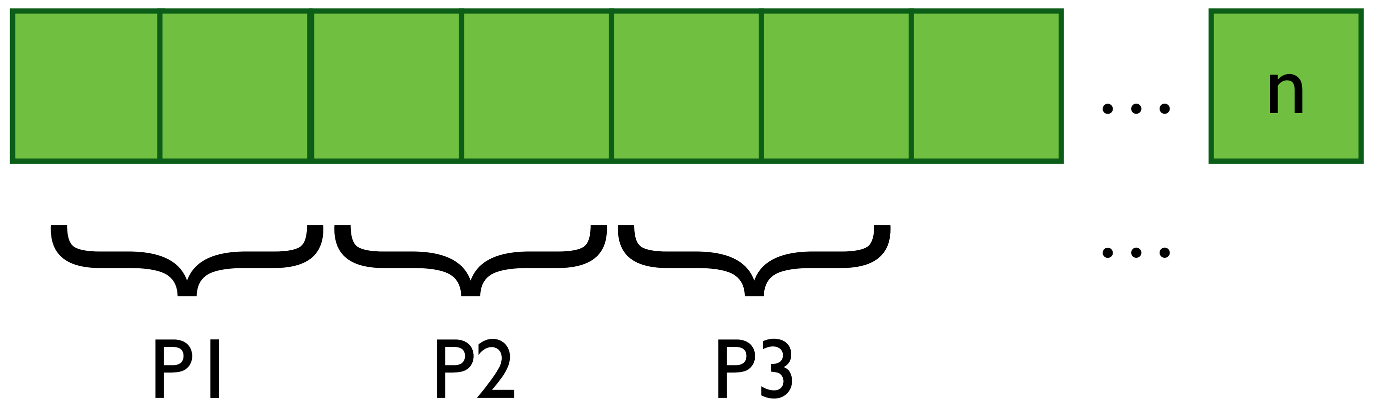
Ivo Gabe de Wolff

Previously…



Task parallelism

- Explicit threads
- Synchronise via locks, messages, or STM
- Modest parallelism
- Hard to program



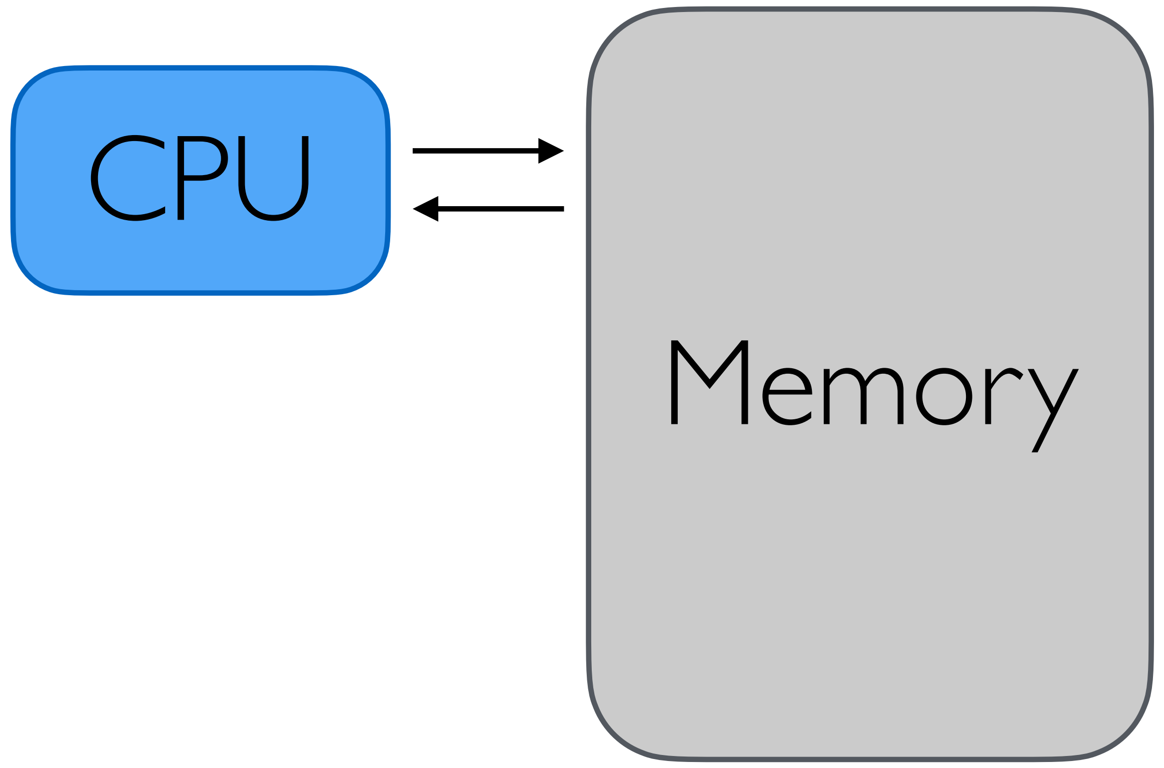
Data parallelism

- Operate simultaneously on bulk data
- Implicit synchronisation
- Massive parallelism
- Easy to program

Performance analysis

- We want to analyse the cost of a parallel algorithm
  - We will consider *asymptotic* costs, to compare algorithms in terms of:
    - How they scale to larger inputs
    - How they scale (parallelise) over more cores
  - Example: some sorting algorithms are $O(n \log n)$ and others $O(n^2)$ over the size of the input
  - Example: RTX 4090 Ti has 16384 "cores" distributed over 128 multiprocessors

RAM

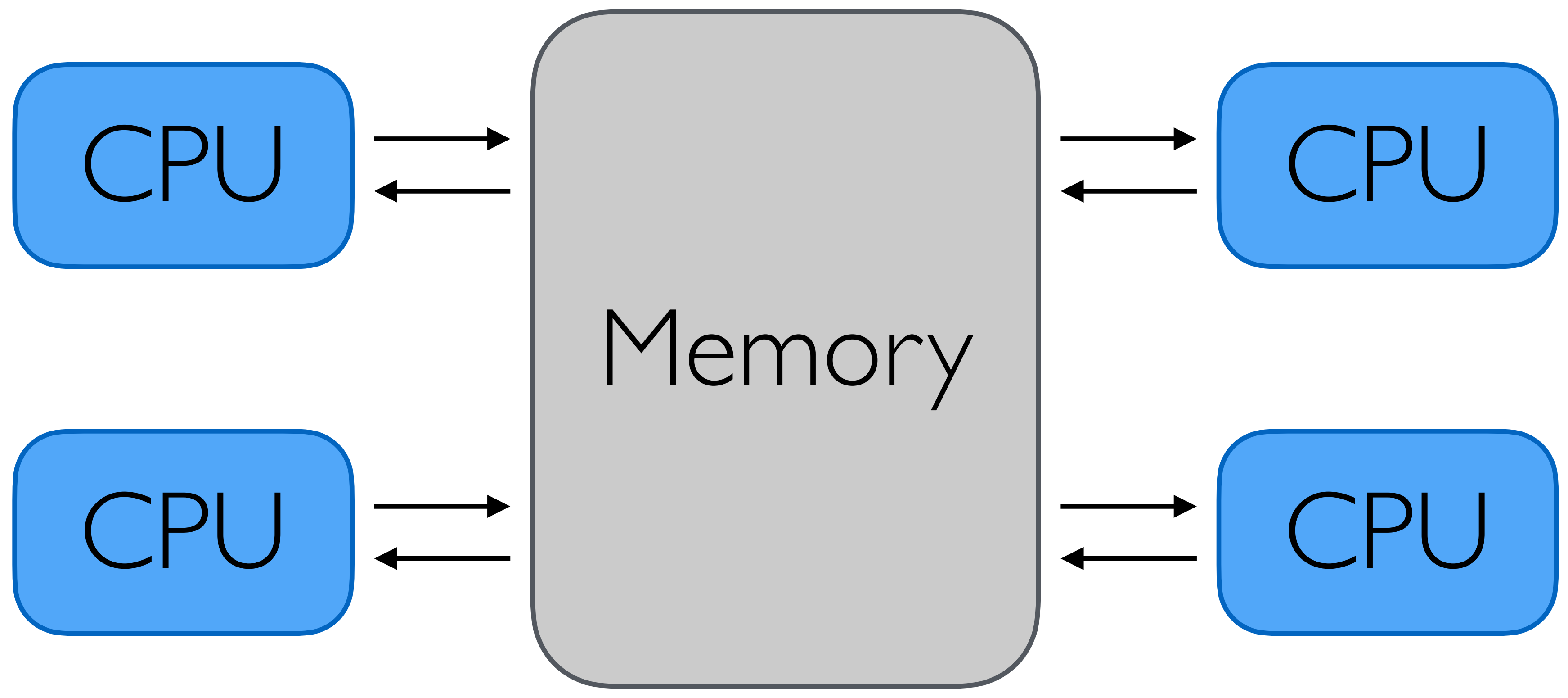


- When designing and analysing sequential algorithms, we use the random access machine (RAM) model
  - All locations in memory can be read from & written to in $O(1)$
  - Summing an array can be done in linear $\Theta(n)$ time

```
s = 0
for (i = 0..n)
  s := s + arr[i]
```

PRAM

- The parallel random access machine (PRAM) model is analogous for talking about parallel algorithms
  - Shared memory machine with multiple attached processors (cores)
  - Ignore details of synchronisation, communication, etc.
  - Question: can we sum an array in parallel using this algorithm?




```
s = 0
parallel_for (i = 0..n)
  s := s + arr[i]
```

Fold

- Binary tree reduction of an array

1. For even `i`:

   `arr[i] += arr[i+1]`

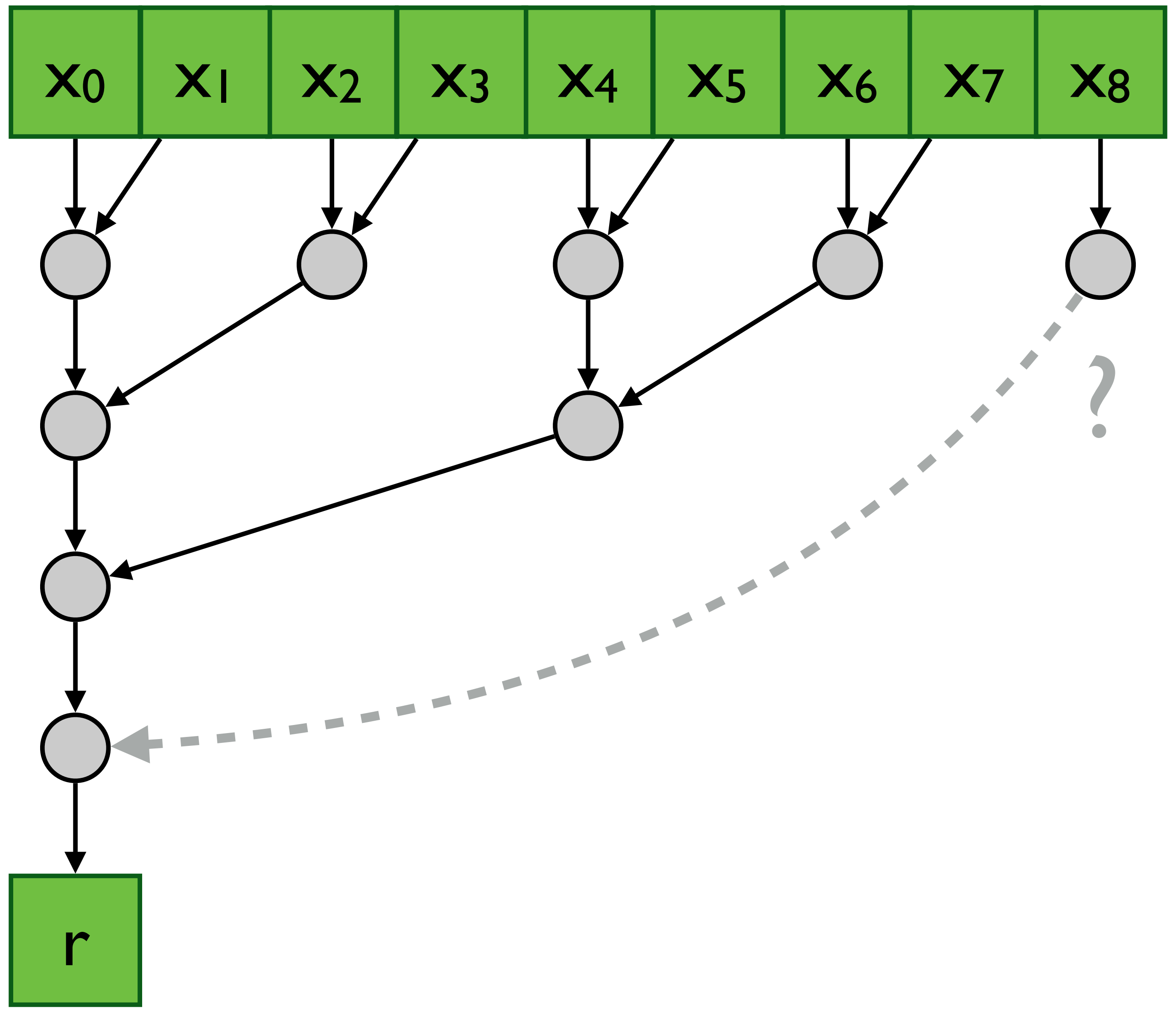
2. For `i` a multiple of 4:

   `arr[i] += arr[i+2]`

3. For `i` a multiple of 8:

   `arr[i] += arr[i+4]`

4. et cetera…



Fold

- Binary tree reduction of an array
  - To calculate step one instantly you need $n/2$ processors: $O(n)$ operations and the whole algorithm takes $O(\log n)$ time
  - The *hardware cost* is thus the number of processor P multiplied by how long you need them: $O(n \log n)$
  - So, we can go faster with parallelism but at a higher hardware cost. Can this be improved?
    1. Can we go faster than $O(\log n)$?
    2. Can we have less hardware cost than $O(n \log n)$?

Fold

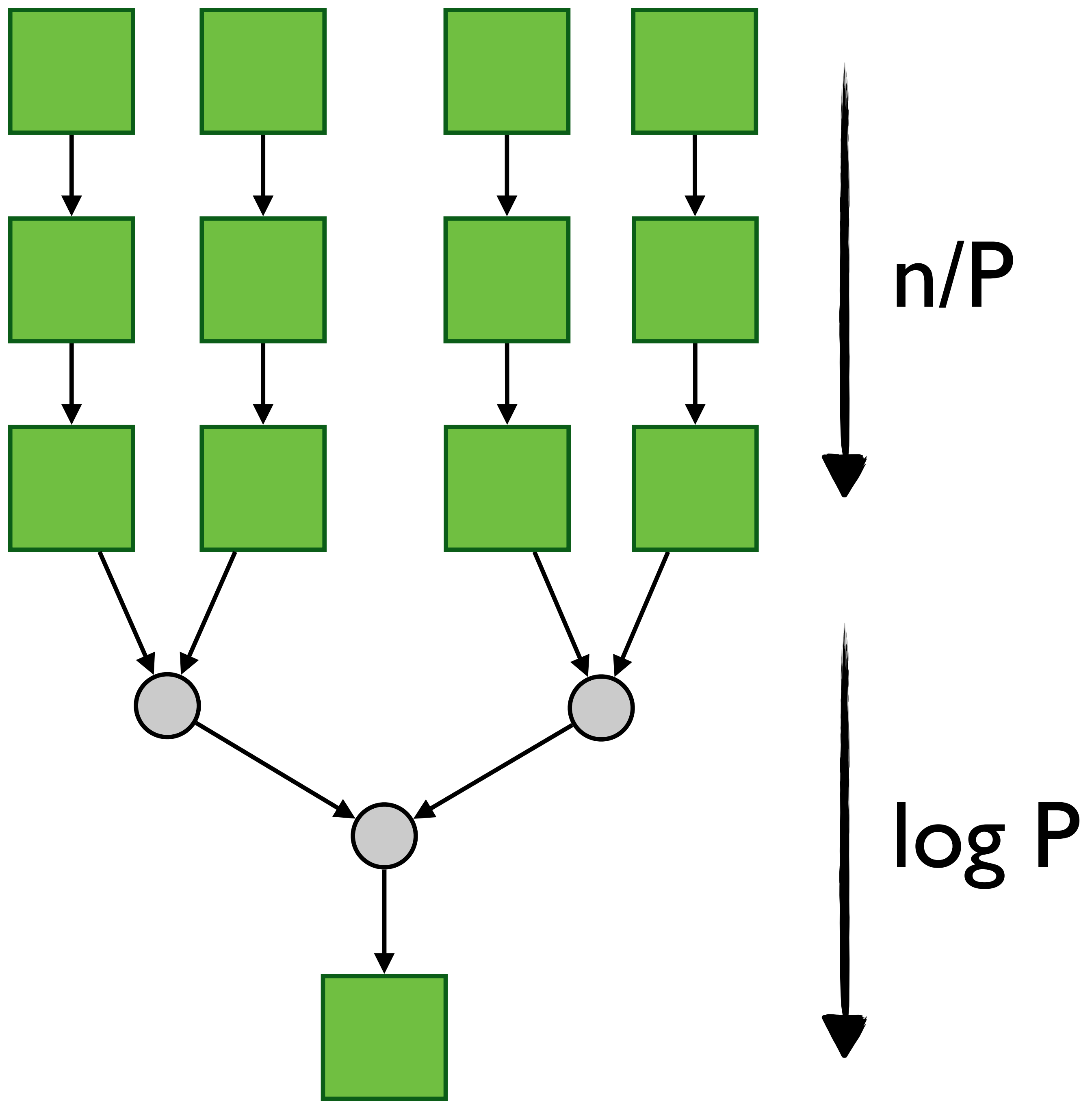
- Question 1: can we sum an array in sub-logarithmic time?
  - Addition is a binary operator
  - Parallel execution of binary operators can, after i rounds, produce values that depend on at most 2^i values
  - So, no matter what you do in parallel, you can not compute the full sum of n numbers in less than $O(\log n)$ time

Fold

- Question 1: proof by induction
  - Induction hypothesis (IH): after i rounds values can only depend on at most 2^i inputs
  - $i=0$: After zero rounds we haven't done anything, so a number only "depends" on itself, so on one number which is 2^0
  - $i+1$: In this round you can combine two inputs from round i, which according to the IH can only depend on at most $2^i + 2^i = 2^{(i+1)}$ inputs
  - Therefore, addition can not be done sub-logarithmically. This holds true for all binary operators, which is why (poly)logarithmic complexity $O(\log^c n)$ is the best possible outcome for parallel execution

Fold

- Question 2: can we reduce the hardware cost?
  - Split the problem into two steps
  - Phase 1: divide the input over the P processors in groups of length n/P
  - Phase 2: use a binary tree reduction to calculate the total from the P partial sums
  - Total time $T_p = n/p + \log p$
    - If $P \leq n / \log n$ then phase one is dominant
    - If $P \leq n / \log n$ then hardware cost is $O(n)$

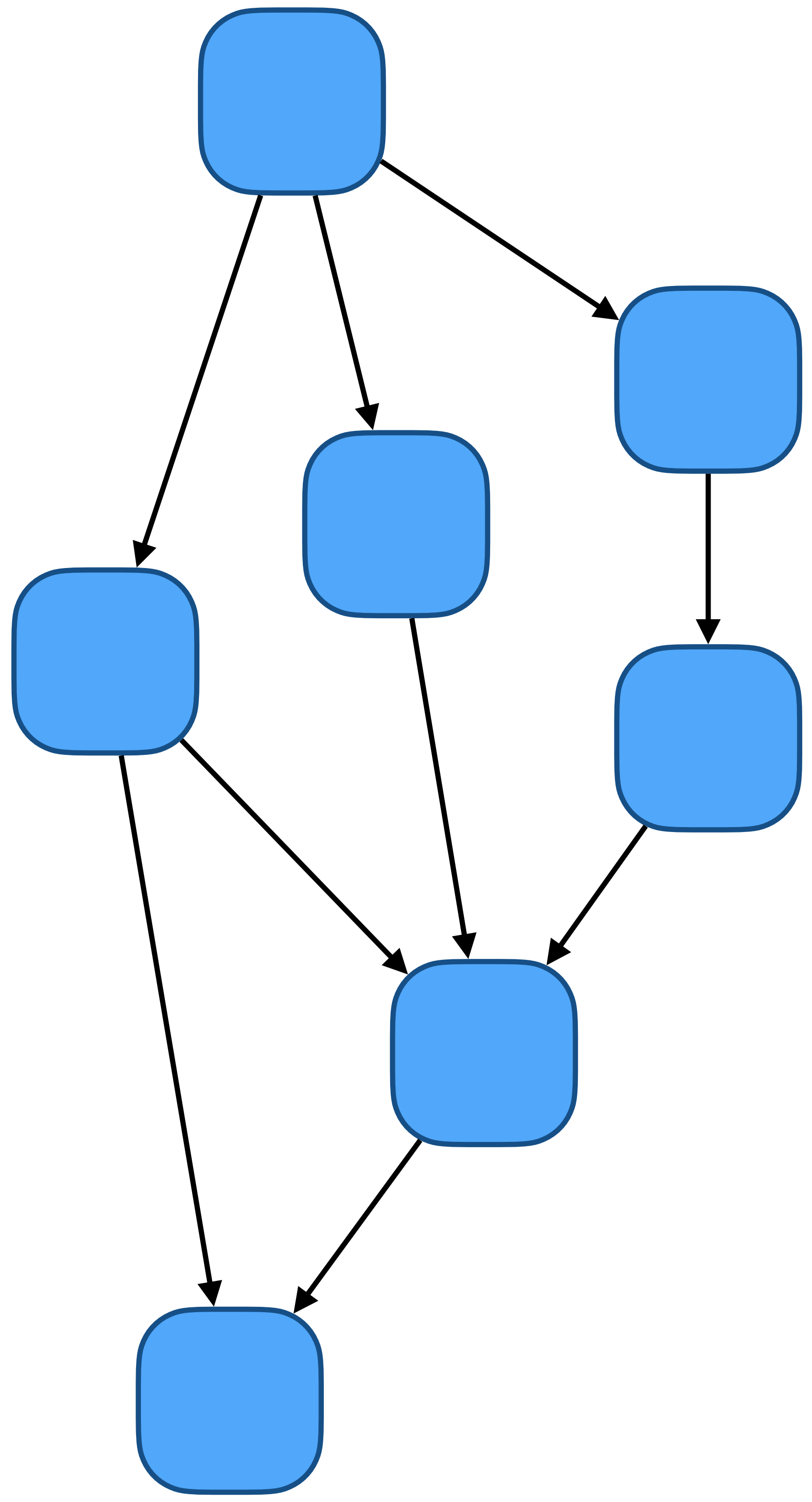


Work & Span

- We don't want a different optimal calculation when executing for a different number of cores
  - Use a description with two parameters, instead of just sequential time
  - Let T_p be the running time with P processors available
  - Then calculate two extremes: the *work* and *span*
- **Work = T_1:** How long to execute on a single processor
- **Span = T_∞:** How long to execute on an infinite number of processors
  - The longest dependence chain / critical path length / computational depth
  - Example: $O(\log n)$ for summing an array

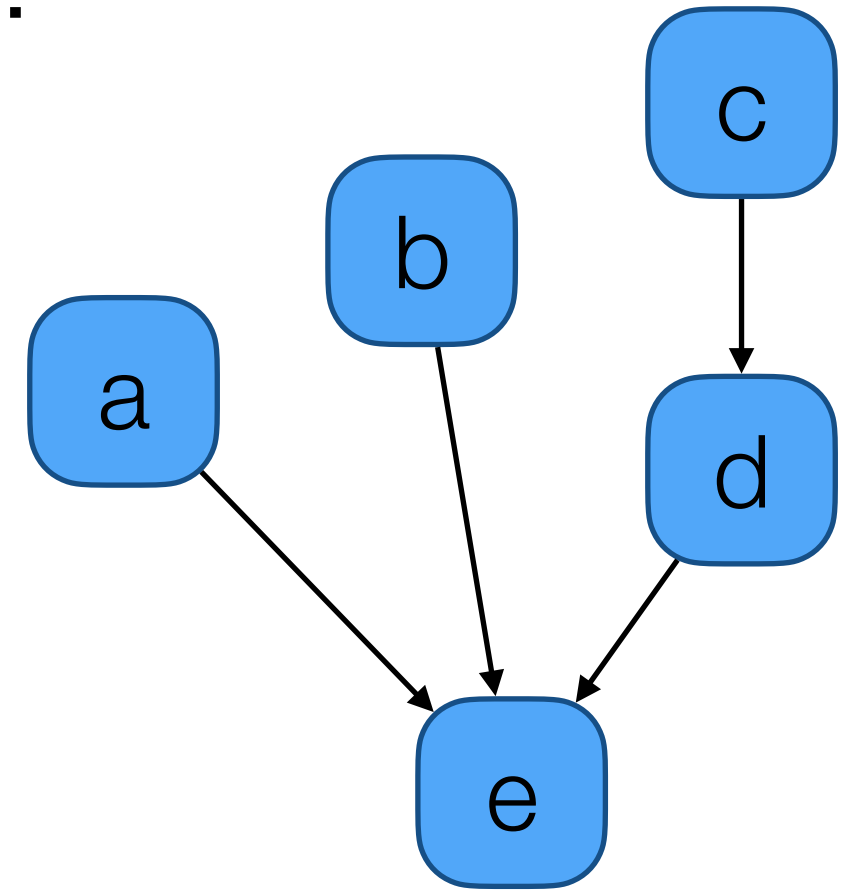
Work & Span

- Program can be seen as a dependency graph of the calculation steps
  - *Work* is the total number of nodes (calculations) in the whole graph
  - *Span* is the number of nodes on the longest path (height of the graph)



Work & Span

- If the work and span are known, you can estimate the time on P processors T_P with:
  - $\max(work/P, span) \leq T_P \leq work/P + span$
  - The latter is at most double the former, so:
    - $T_P = O(work/P + span)$
  - Question: what is the time to execute on 1, 2, or 3 cores?



Scheduling

- Brent proved that *greedy scheduling* is always two-optimal
  - We say a step is ready when all its predecessors (dependencies) have been computed in previous rounds
  - A greedy scheduler does as many steps in a round, but does not care which
  - This is two-optimal:
    Greedy scheduling takes at most twice as long as the optimal schedule
- Say T_P* is the time for the optimal schedule, then:
  - $T_P* \geq work/P$, because even the best schedule still has P cores available
  - $T_P* \geq span$, because all calculations on a path must be done sequentially

Scheduling

- Greedy scheduling
  - Full round: if there are P or more steps ready, do P steps this round; this happens at most $work/P$ times
  - Empty round: there are fewer than P steps ready; this happens at most $span$ times, because every round the span decreases by one
  - The length of the greedy schedule is:

$$
\begin{aligned}
T_P &= full + empty \\
&\leq work/P + span \\
&\leq T_P{*} + T_P{*} \\
&\leq 2T_P{*}
\end{aligned}
$$

Scheduling

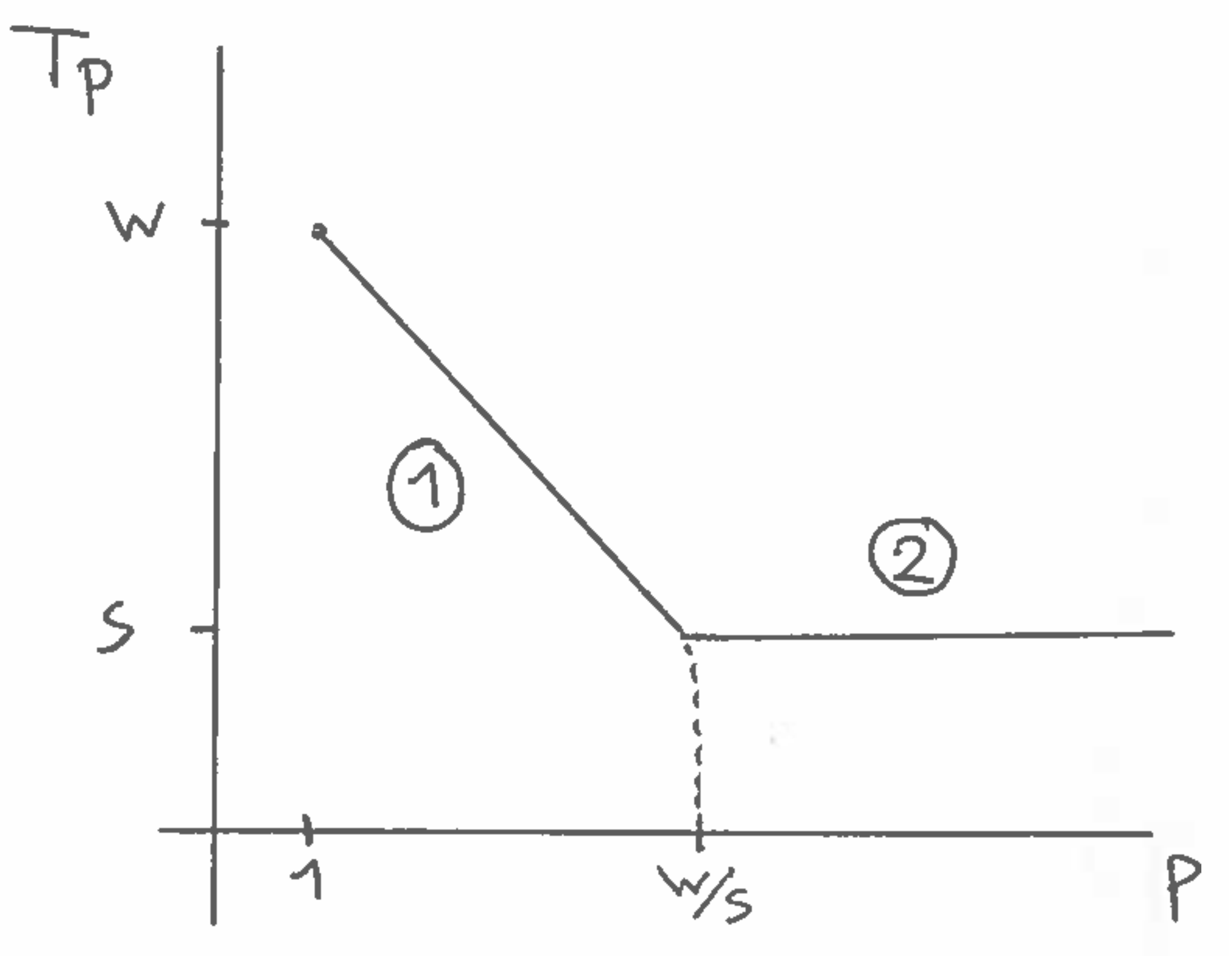
- Greedy scheduling
  - Greedy scheduling has length at most twice the length of the optimal one, so is asymptotically optimal
  - Because $work/P + span$ and $max(work/P, span)$ are asymptotically equal (differ by a factor of two), we can say that $T_P = max(work/P, span)$

Work & Span

- Greedy scheduling

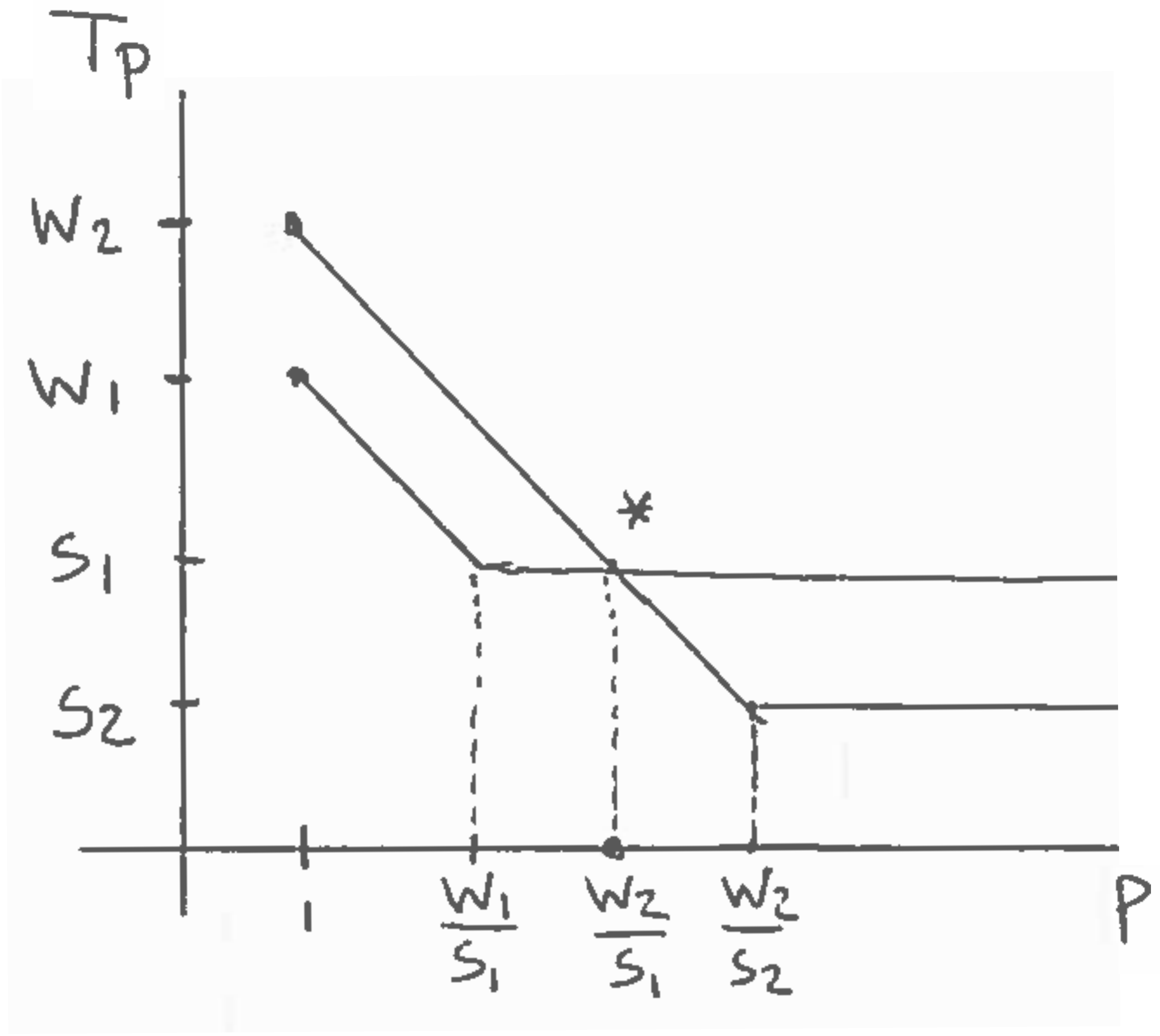
1. As long as $P \leq work/span$ the first term is dominant and the calculation can be shortened by adding more cores: *work bound phase*

2. If we have $P > work/span$ then the runtime will not get shorter by adding more cores: *span bound phase*



Work & Span

- When comparing algorithms, low work is better than high work, and low span is better than high span
  - What if algorithm one has better work complexity $w_1 < w_2$
  - But algorithm 2 has better span complexity $s_1 > s_2$
  - Low span is theoretically nice, but since we don't have infinite processors in practice, be careful not to lower span at the cost of too much extra work



Work & Span

- Calculating work and span is the same as computing the time of an algorithm, as learned in the course data structures
  - Count the number of instructions/operations
  - In the case of a loop, the cost of the body times the number of repetitions
  - For recursion, use the Master Theorem
- For the analysis of parallel algorithms:
  - You must do this process twice, once each for work and span
    - Work is done as you would for a sequential algorithm
    - Span takes the maximum of the branches which are performed in parallel

Example: zipWith

- Pair-wise multiply the elements of two arrays

```
1  parallel_for (i = 0..n)
2    r[i] := x[i] * y[i]
```

- Work analysis:
  - Doesn’t care about parallelism
  - Line one says that this is done n times, so costs $\Theta(n)$ steps
- Span analysis:
  - The maximum cost of all the branches which are done in parallel
  - Loop on line 1 is parallel, so take the longest path of steps: $\Theta(1)$

Example: fold (I)

- Add up all the numbers in an $n \times n$ matrix A, with subtotals per row

```
1 parallel_for (j = 0..n)
2   s[j] = 0
3   for (i = 0..n)
4     s[j] := s[j] + A[i,j]
5 t = 0
6 for (i = 0..n)
7   t := t + s[j]
```

- Work analysis:
  - Loop on line 3-4 costs $\Theta(n)$ steps
  - Line one says this will be done n times, so line 1-4 take $\Theta(n^2)$ steps
  - Line 6-7 take $\Theta(n)$ steps
  - Total is $\Theta(n^2)$ work

Example: fold (I)

- Add up all the numbers in an n x n matrix A, with subtotals per row

```
1  parallel_for (j = 0..n)
2    s[j] = 0
3    for (i = 0..n)
4      s[j] := s[j] + A[i,j]
5  t = 0
6  for (i = 0..n)
7    t := t + s[j]
```

- Span analysis:
  - Loop line 3-4 is sequential, $\Theta(n)$ steps
  - Loop line 1 is parallel, so we take the longest path of steps from line 1-4: $\Theta(n)$
  - Line 5-7 still have $\Theta(n)$ sequential steps
  - Total span is $\Theta(n)$ steps

Example: fold (2)

- Parallel algorithms can often use recursion effectively
  - We want a method `sum(A, p, q)` that calculates the sum of all numbers in A in the range $[p,q)$
  - Using recursion, pretend you already have a clever way to sum $n/2$ numbers, which you want to use to calculate the sum of n numbers

```
sum (A, p, q)
  parallel_for (i = 0..(q-p)/2)
    B[i] = A[p+2*i] + A[p+2*i+1]

  sum (B, 0, (q-p)/2)
```

  - Ignore possibility of uneven number of inputs, base case of recursion, etc…

Master Theorem

- The master theorem provides a solution to recurrence relations of the form
  - For constants $a \geq 1$ and $b > 1$ and f asymptotically positive

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- The master theorem has three cases:

Recursion dominates

If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$

for some $\epsilon > 0$,

then $T(n) = \Theta\left(n^{\log_b a}\right)$

Both contribute

If $f(n) = \Theta\left(n^{\log_b a}\right)$, then

$T(n) = \Theta\left(n^{\log_b a} \log n\right)$

***f* dominates**

If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$

for some $\epsilon > 0$, and

$af(n/b) \leq cf(n)$

for some $c < 1$

for all n sufficiently large,

then $T(n) = \Theta(f(n))$

https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

Recall: Master Theorem

- The master theorem provides a solution to recurrence relations of the form
  - For constants $a \geq 1$ and $b > 1$ and f asymptotically positive

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- Examples:
  - Merge sort: $T(n) = 2T(n/2) + n$
    Then case 2 gives (a=2, b=2): $T(n) = \Theta(n \log n)$
  - Traversing a binary tree: $T(n) = 2T(n/2) + O(1)$
    Then case 1 gives (a=2, b=2, ε=1): $T(n) = \Theta(n)$

Example: fold (2)

- Parallel algorithms can often use recursion effectively

```
1   sum (A, p, q)
2     parallel_for (i = 0..(q-p)/2)
3       B[i] = A[p+2*i] + A[p+2*i+1]
4
5     sum (B, 0, (q-p)/2)
```

- Work analysis:
  - Line 3 is $\Theta(1)$
  - Line 2 says it is done $n/2$ times, so $\Theta(n/2)$
  - Line 3 is a recursive call on $n/2$ inputs. Call the work $W(n)$ and we get $W(n) = W(n/2) + \Theta(n/2)$
  - Solve with the master theorem (a=1, b=2, ε=1, case 3): $W(n) = \Theta(n)$

Example: fold (2)

- Parallel algorithms can often use recursion effectively

```
1    sum (A, p, q)
2      parallel_for (i = 0..(q-p)/2)
3        B[i] = A[p+2*i] + A[p+2*i+1]
4
5      sum (B, 0, (q-p)/2)
```

- Span analysis:
  - Line 2-3 have constant span because they are done in parallel
  - This means the span $S(n) = S(n/2) + \Theta(1)$
  - Solve with the master theorem (a=1, b=2, case 2): $S(n) = \Theta(\log n)$
- *Conclusion*: we can sum n numbers in linear work and logarithmic span

Example: scan (I)

- Parallel implementation of prefix sum

```
   input: [3,4, 4, 4, 4, 3, 5, 4, 5]
expected: [3,7,11,15,19,22,27,31,36]
```

  - Split the data over two processors and perform a prefix sum individually on each part

```
             split: [3,4, 4, 4, 4] ¦ [3,5, 4, 5]
left/right result: [3,7,11,15,19] ¦ [3,8,12,17]
                          P1              P2
```

Example: scan (I)

- Example: recursive implementation of prefix sum:

```
prefix_sum (A, p, q)
  // base case

  m = (p+q)/2
  prefix_sum(A, p,   m)
  prefix_sum(A, m+1, q)
  parallel_for (i = m+1..q)
    A[i] = A[i] + A[m]
```

In parallel (lines `prefix_sum(A, p, m)` and `prefix_sum(A, m+1, q)`)

- Span (a=1, b=2, case 2): $S(n) = S(n/2) + 1 = \Theta(\log n)$
- Work (a=2, b=2, case 2): $W(n) = 2\,W(n/2) + n = \Theta(n \log n)$

Efficient & optimal

- The parallelisation *overhead* of an algorithm is its work divided by the cost of the best sequential algorithm
  - For this parallel scan we have to put $O(n \log n)$ work into something which can be done sequentially in linear $O(n)$ time: the overhead is logarithmic
  - A parallel algorithm is:
    - *Efficient* when the span is poly-logarithmic and the overhead is also poly-logarithmic
    - *Optimal* when the span is poly-logarithmic and the overhead is constant

Example: scan (2)

- Let's try a different approach to parallelising scan:

```
   input: [3,4, 4, 4, 4, 3, 5, 4, 5]
expected: [3,7,11,15,19,22,27,31,36]
```

  - Pair up neighbours at the even positions:

```
          [  7,    8,    7,    9   ]
```

  - Perform a prefix sum of these values:

```
          [  7,   15,   22,   31   ]
```

  - At the uneven positions add the input value at that position to the output of the previous step on the left:

```
          [3,7,11,15,19,22,27,31,36]
```

Example: scan (2)

- We can implement this recursively by keeping track of a hop distance

```
prefix_sum (A, d)
  parallel_for (i = even multiple of d)
    A[i] += A[i-d]
  prefix_sum(A, 2*d)
  parallel_for (i = uneven multiple of d)
    A[i] += A[i-d]
```

- Work:
  - Algorithm does $n-1$ additions and one half-size prefix sum
  - Master theorem ($a=1$, $b=2$, $\varepsilon=1$, case 3): $W(n) = W(n/2) + n = \Theta(n)$

Example: scan (2)

- We can implement this recursively by keeping track of a hop distance

```
prefix_sum (A, d)
  parallel_for (i = even multiple of d)
    A[i] += A[i-d]
  prefix_sum(A, 2*d)
  parallel_for (i = uneven multiple of d)
    A[i] += A[i-d]
```

- Span:
  - Additions are done in two (parallel) groups, before and after the prefix sum
  - Master theorem (a=1, b=2, case 2): $S(n) = 1 + S(n/2) + 1 = \Theta(\log n)$
  - Since the span is logarithmic and there is no overhead, this prefix sum is parallelised optimally

Summary

- Work and span are used to analyse and compare asymptotic behaviour of parallel algorithms
  - Work: total number of steps (computations)
  - Span: longest path of steps that need to be done sequentially (steps)
- The PRAM model ignores practical issues such as memory access latency
  - Assume uniform costs for all memory access
- Time to perform something on P cores: $T_P = \Theta(work/P + span)$
  - Compare to the formulation by Amdhal

Next time…

- Thursday: Revision lecture
  - This will consist of the last lectures presented simultaneously (it is up to you to parallelise your brain before then)
  - *Send me questions/topics to cover via Teams!*

tot ziens

Photo by Claudio Piccolo