1 A New View of Efficiency

How we analyze the efficiency of an algorithm depends on our *model of computation*.

- What can we do in constant time? (e.g. arithmetic)
- Which operations are important? (e.g. main- vs external-memory reads and writes)
- How many operations can occur at the same time?
- We’ll consider the last of these today.

We are going to define a new computational model to measure how “parallel” a computation is.

- Real-world processors offer many forms of parallelism.
- Multiple cores, SIMD vector instructions, FPGA systolic arrays, ...
- All have the property that they can execute different operations in an algorithm at the same time.
- We’d like to measure, in a general way, how much an algorithm can be parallelized.
- But just as big-O notation is agnostic about the (sequential) machine on which an algorithm executes, we’d like a measure of parallelism that is not closely tied to any one hardware platform.
- Instead, we will define a simple model of computation that lets programs with more parallelism run faster.
- Assume we have a computer with an *unbounded* number of processors – as many as we want for any given algorithm and input.
- All processors can execute independently but share the same storage (memory).
- Any step of an algorithm can run equally fast on any processor.
- (We will ignore the cost of any communication needed to assign work to each processor.)

How does this model help measure parallelism?
• We can view any computation as a directed, acyclic graph (DAG) of constant-time operations.

• The DAG has a single input node that starts the computation and a single output node that returns the result.

• For example, consider the following simple computation:

\[
F(x, y) \\
z \leftarrow x + y \\
w \leftarrow 2 \times z \\
v \leftarrow z + 7 \\
\text{return } w \times v
\]

• We can render \( F \) as a DAG of operations like this:

• Each edge of the DAG is a dataflow dependency from the operation that computes a value to the operation that uses it.

• An operation cannot execute until its predecessors in the DAG have executed; e.g. we can’t compute \( v \) or \( w \) until we’ve computed \( z \).

• But if we have two operations, neither of which is a predecessor of the other, we can execute them on two processors in parallel.

• In this case, \( v \) and \( w \) can be computed in parallel from \( z \).

We’re now ready to define some measures of efficiency.

• Suppose we build a DAG that describes the operations of an algorithm \( A \).

• The work of \( A \), denoted \( T_1 \), is the total number of operations in the DAG.

• “Work” is just the familiar running time of \( A \) on a single processor, which can execute only one operation at a time.

• The span of \( A \), denoted \( T_\infty \), is the minimum number of steps needed to finish all the work in the DAG, given an unbounded number of processors that can execute operations in parallel.
• This is just the length of a longest path in the DAG.

• Indeed, all operations in any one path from the algorithm’s input to its output must happen one after the other, so cannot run in parallel.

• Moreover, a machine with an unbounded number of processors can always execute A in time $T_\infty$: at time step $i$, execute in parallel all operations whose longest path from the input has length $i$.

• (Both $T_1$ and $T_n$ are functions of A’s input size $n$.)

• Finally, the parallelism of A is defined as $T_1/T_\infty$.

• Intuitively, algorithms with high parallelism permit many operations to execute concurrently and so can take advantage of highly parallel processing hardware.

• Conversely, an algorithm with low parallelism is unlikely to be able to exploit such hardware.

2 Parallelism of Divide-and-Conquer Algorithms

• Conventionally, we analyze the work of D&C algorithms using a recurrence, which describes a recursion tree.

• For example, consider our naive recursive algorithm for multiplying two $n \times n$ matrices:

• We accounted the total cost of multiplication with this recursion tree:

To find an algorithm’s work and span, we need to think about its DAG.

• At a high level, the DAG of a D&C algorithm looks like two copies of its recursion tree stuck together.
• The “top half” does the “divide” operations, while the “bottom half” does the “combine” operations. They meet at the base cases.

• For matrix multiply, the divide operations copy the input matrices into eight submatrices, in time $O(n^2)$ for an $n \times n$ input.

• The combine operations add the products of submatrices together, again in time $O(n^2)$ for an $n \times n$ result.

• Hence, the DAG looks something like this:

• The work $T_1$ is just the sum of work at each node – $\Theta(n^3)$ by our earlier analysis.

• *If we assume that the work within one node is done sequentially*, the span is the work done along any one path through the DAG (all paths do the same work):

$$T_\infty(n) = \sum_{i=0}^{\log_2 n-1} \Theta((n/2^i)^2) + O(1) + \sum_{i=\log_2 n-1}^{0} \Theta((n/2^i)^2)$$

$$= \Theta(n^2).$$

• Hence, the parallelism of matrix multiply so far is $\Theta(n^3)/\Theta(n^2) = \Theta(n)$.

• But in fact, we need not do the work of a single node sequentially!

• We can copy all $(n/2^i)^2$ elements of a matrix into submatrices in parallel, and we can do all $\Theta((n/2^i)^2)$ element-wise adds required by combine in parallel.

• Hence, we can replace each node above by a small DAG with span $O(1)$:
Since each of these little DAGs has height $O(1)$, the span of the fully parallelized multiply is just

$$T_\infty(n) = \sum_{i=0}^{\log_2 n - 1} O(1) + O(1) + \sum_{i=\log_2 n}^{0} O(1)$$

$$= \Theta(\log n).$$

The true parallelism of matrix multiply is therefore $\Theta(n^3 / \log n)$.

Note that the same span analysis can be applied to Strassen’s algorithm, whose parallelism is therefore $\Theta(n^{2.81} / \log n)$.

3 Improving Parallelism: Parallel MergeSort

We’ll now look at an example where we can modify a D&C algorithm in a non-obvious way to improve its parallelism.

- Consider the warm-and-fuzzy MergeSort algorithm:

  ```plaintext
  MERGESORT(A[1..n])
  if n = 1
      return A
  B ← MERGESORT(A[1..n/2])
  C ← MERGESORT(A[n/2 + 1..n])
  D ← []
  while B, C not both empty do
      move min of B[1], C[1] to end of D
  return D
  ```

- The “divide” part of the algorithm forms a tree of $\Theta(n \log n)$ total nodes.
- Each node copies its input array into two halves, doing linear work.
- But if all elements are copied in parallel, the span of a copy is only $O(1)$.

- Hence, the divide work and base-case work alone have span $O(\log n)$, the height of the tree.
- But every divide has a corresponding combine operation!
- Moreover, combine as described here is sequential – we cannot move an item to $D$ until we’ve moved the previous item.
• Hence, each node of the “upside-down” combine tree has linear work and linear span in its input size.

• Conclude that the span of this algorithm is

\[ \Theta(\log n) + \sum_{i=1}^{\log n} c \times 2^i = \Theta(n) \]

and so the algorithm’s parallelism is only \( \Theta(\log n) \).

Can we do better?

• We will introduce a replacement merge operation that is more parallelizable, i.e. has similar work but a lower span.

• Let \( |B| = m \) and \( |C| = \ell \).

• This method merges \( B, C \) into an array \( D \) of size \( m + \ell \).

• Suppose WLOG that \( m \geq \ell \) (if not, swap them).

• If \( m = 1 \), then just write \( B, C \) into \( D \) time \( O(1) \).

• Otherwise, use binary search to find \( s \) s.t. \( C[s] \leq B[m/2] < C[s+1] \).

• Recursively merge \( B[1..m/2] \) with \( C[1..s] \), storing the result into \( D[1..s+m/2] \).

• Recursively merge \( B[m/2+1..m] \) with \( C[s+1..\ell] \), storing the result into \( D[s+m/2+1..m+\ell] \).

OK, what’s the work and span of this new merge operation on inputs of size \( m \) and \( \ell \)?

• First, how big are the two recursive calls?

• Each recursive call gets \( m/2 \) elements from \( B \) and up to \( \ell \leq m \) elements from \( C \).

• As a lower bound, we have

\[ m/2 + \ell \geq m/4 + \ell/4 = (m + \ell)/4. \]

• Moreover, as an upper bound,

\[ m/2 + \ell \leq m/2 + \ell/2 + m/4 + \ell/4 = 3(m + \ell)/4. \]

• Hence, each recursive call is of some size between \((m + \ell)/4\) and \(3(m + \ell)/4\), and the two calls combined are of total size \( m + \ell \).
Moreover, it takes one binary search, in time $O(\log \ell)$, to divide the input into the two subproblems, and no time at all to combine (since we write into $D$ directly in the base case).

Now let’s discuss the span. Suppose $n = m + \ell$.

Each recursive call for an input of total size $n$ has size at most $3n/4$.

Moreover, an input of size $n$ requires one (sequential) binary search of span $\Theta(\log n)$.

Conclude that

$$T_\infty(n) = T_\infty(3n/4) + C \log n.$$ 

By the Master Theorem, we have that for one merge, $T_\infty(n) = \Theta(\log^2 n)$.

The work is somewhat harder to compute.

In general, we have

$$T_1(n) = T_1(\alpha n) + T_1((1 - \alpha)n) + C \log n$$

where $\alpha \in [1/4, 3/4]$.

Our usual tricks don’t work to solve this recurrence, but we can guess a solution and verify it.

In this case, we can prove inductively that $T_1(n) = \Theta(n)$ (Proof omitted due to dullness.)

What does this imply for the parallelism of MergeSort, if we replace each sequential merge with a divide-and-conquer merge?

On an input of size $n$, recall that the divide portion of the sort does work $\Theta(n)$ at each level to copy the input array.

Hence, we have that for the entire MergeSort (divide plus combine),

$$T_1(n) = 2T_1(n/2) + \Theta(n) + \Theta(n)$$

and so, by the Master Theorem, the work remains $\Theta(n \log n)$.

Moreover, we have that

$$T_\infty(n) = T_\infty(n/2) + O(1) + \Theta(\log^2 n)$$

and so, again by the Master Theorem, the span is $\Theta(\log^3 n)$.

Conclude that the parallelism of our modified algorithm is a whopping $\Theta(n/ \log^2 n)$, which is asymptotically much bigger than with sequential merge!