CS5234: Combinatorial and Graph Algorithms Lecture 11–12

Parallelism

Lecturer: Seth Gilbert *October 27, 2016*

Abstract

Today, we will focus on parallel algorithms. We will look at two different models for thinking about parallel algorithms: the fork-join model, which works well for multicore/multithreaded machines, and the MapReduce model, which works well for clusters. We will look at how to analyze parallel algorithms (e.g., in terms of work, span, and parallelism). And we will several examples, including how to parallelize Breadth-First-Search and Bellman-Ford.

1 Introduction

Moore's Law. For years, everyone depended on Moore's Law to ensure that their programs would run faster. Every year we see more transistors packed onto a chip. More transistors used to mean faster CPUs. One (rough) proxy for the speed of a CPU is the clock rate, and again you can see clock rates increasing rapidly over the years, up until about 2006. However, since 2006, clock rate has been stable, around 3GHz. More transistors no longer leads inherently to faster CPUs. Instead, we have seen a significant increase in the available parallelism. One might divide the history of CPU design into the following eras:

- From 1970 until 1990, computers executed < 1 instruction per cycle. This was the era of in-order instruction issue.
- From 1990 until 2002, computers started to use deeper pipelines and out-of-order execution. This led to up to 5 instructions per cycle.
- In 2007, we were seeing up to 10 instructions per cycle.
- By 2008, we were seeing up to 20 instructions per cycle.
- By 2011, we were seeing up to 45 instructions per cycle.

Most of the increase from 2008 onwards was driven by multihreaded, multicore CPUs. So if you want a faster program, you need to be more parallel. Instructions per cycle continues to follow Moore's Law, even though clock rate does not!

The moral of the story is that if you want your program to run faster today, you need to take advantage of parallelism.

CS5234 motivation. So far this semester, we have looked at three main topics: sublinear time algorithms, sketches, and cache-efficient algorithms. We are concluding by looking at parallelism in part because it provides techniques that can integrate with the algorithms we have already looked at.

For example, many of the sublinear time algorithms we looked at are inherently parallel. These algorithms depend on sampling, and typically the sampling can be performed in parallel. Each processor or thread can take some of the samples, and the results can be merged. (Notice you can add n numbers in $O(n/p + \log n)$ time in parallel.)

Many of the sketching algorithms are also readily parallelizable. For example, you can divide up your data (e.g., your stream of graph updates) among processors and have each build a separate sketch. Then we can merge the sketches to produce a single summary sketch for the entire dataset. Of course, there may be tricky issues in parallelizing any algorithm that operates on the sketch, of course.

Finally, cache efficiency is equally important in parallel algorithm. One tricky part of parallel algorithms is the cost of memory access, and issues of caching (which we will mostly ignore today). For example, cores on the same CPU often share (some) cache, while different CPUs do not. And when two different cores/CPUs access the same piece of memory, that is expensive. There are many open research questions focusing on these issues.

Different types of parallelism. There are many different types of parallelism that you may encounter, and different styles of parallel algorithms that may best fit the different environments. For example, you might imagine the following different settings:

1. Multicore servers: many computational units on one chip.
2. Multisocket servers: many CPUs on one board, or a server with one or more GPUs.
3. Cluster computing: many boards connected together by a fast network (e.g., in a data center).
4. Distributed networking: many computers connected together by a distributed network.

In each of these settings, we need parallel algorithms. And in each of these settings, different parallel algorithms may be needed. I want to give two examples today: (i) a basic multicore example of BFS, specified in a fork-join parallel model; and (ii) a MapReduce algorithm for Bellman-Ford designed to run on a cluster.

2 Fork-Join Parallelism

Let's start with a simple example: assume that we want to find the sum of a set of integers in an array $A[1, .., n]$. We can do it sequentially in $O(n)$ time, simply enumerating the array and adding each number to the sum. Since n is very large, we want to use parallelism to do better.

2.1 Simple Example: Addition

If we have p processors (or p cores or p servers), a natural idea is to divide the array in p pieces, e.g., $A[1, .., n/p]$, $A[n/p+1, .., 2n/p]$, $\ldots$, $A[(p-1)n/p+1, .., n]$. Each processor individually performs sums its component of the array. This yields p partial sums: $s_1, s_2, \ldots, s_p$. It remains to sum these p values. We could simply have one processor perform this summation, yielding a total running time of $O(n/p + p)$.

If p is reasonably small (e.g., a 4-core machine), this may be entirely sufficient. However, as p gets large, it is natural to ask if we can do better.

Imagine organizing the p values as leaves in a complete binary tree. Each process is assigned to a node in the tree, where no process is assigned to more than one node per level of the tree. We can then compute the sum of the p values in parallel, level-by-level: each node computes the sum of its two children. In $\log p$ time, we have computed the root, which contains the final answer. Thus, the total running time of this algorithm is $O(n/p + \log p)$.

It is slightly clumsy to have to handle assigning each processor exactly which summations to perform. First, p may vary from system to system (or from execution to execution), and it would be preferably if the program continued to work. Second, as the algorithms get more complicated, the scheduling becomes increasingly complicated to manage. When designing an algorithm, we would prefer to relegate the scheduling problem to a separate scheduling component—instead, we want to focus on designing an efficient algorithm to solve the task at hand.

Consider the following alternative way to write the same parallel algorithm for summing the values in an array in the "fork-join" model: Notice this algorithm *almost* looks sequential. It divides the array in half and recursively calculates the sum on the two halves. It then adds these two half-sums together. The only aspect that is parallel is that we explicitly state that the two half-sums can be computed in parallel. That is, this procedure creates two subtasks

Algorithm 1: Sum($A, begin, end$)

```
if ($begin = end$) then
    return $A[begin]$
else
    $mid = (begin + end)/2$
    in parallel
        (1) $s_1 = Sum(A, begin, mid)$
        (2) $s_2 = Sum(A, mid + 1, end)$
    return $s_1 + s_2$
```

that can be executed in parallel. Eventually, there are n tasks in total, and these can be assigned efficiently to however many processors exist. We will assume that a scheduler does a good job of this.

If there are p processors, you can imagine that at some point, once there are p separate sub-arrays being summed, we simply assign one processor to each subarray and have it perform all the remaining subtasks for the component.

If you think about it, this yields essentially the same algorithm as before. The array is divided into p parts, each sub-part is summed, and then in parallel the individual sums are combined together again until we have a single final answer.

2.2 Metrics

Without knowing about the scheduler, however, how do we analyze this type of algorithm? There are basically two metrics that we can calculate, without knowing anything about p or the scheduler. These metrics calculate something about the inherent parallelism of the algorithm:

- Work W: how long would it take to run on one processor? Equivalently, this is how many steps of computation are needed, regardless of whether they are scheduled in parallel or sequentially. This metric captures something about the total cost of the algorithm. This can be analyzed by ignoring the parallel constructs and simply determining the cost of the algorithm in the usual way.
- Span S: how long would it take with an infinite number of processors? That is, if you had as much computation as possible, how long would it take? This is sometimes known as the "critical path" since it is the length of the longest sequential component of the computation. This can be analyzed by assuming that parallel computation happens in parallel, and hence we only need to think about the longer of the two jobs to complete.

Most algorithms have some trade-off between work and span. Sometimes you can reduce the span, but only at the cost of more total work. Other times, you might reduce the work, and end up with a larger span. Depending on the target application, you have to choose the best point in the trade-off space.

One interesting metric is the *parallelism*, defined as W/S. This tells you approximately how many processors you can usefully use. When the number of processors $p > W/S$, then you are not fully able to exploit your hardware: there is not enough parallelism in your algorithm to use all p processors efficiently. (In essence, you are constrained by the critical path, i.e., the span.)

Conversely, if $p < W/S$, then all p processors are being use, and there is more parallelism in the algorithm that you could exploit with a bigger parallel system. (In essence, you are constrained by the work, i.e., there is more work to do than processors available.)

It turns out that these two metrics, work and span, tell you almost everything you need to know about a parallel algorithm. One critical reason for that is that good schedulers exist that will execute this style of algorithm nearly optimally[1]. In particular, if you execute an algorithm with work W and span S, it will run in time $O(W/p + S)$ using a good scheduler. (Notice that it is pretty easy to see that any algorithm work W takes time at least $\Omega(W/p)$, and any algorithm with span S takes time at least $\Omega(S)$, and so this is roughly the best you can do.)

Going back to our example of summing an array, we can first calculate the work via a simple recurrence:

$$W(n) = 2W(n/2) + O(1) = O(n)$$

That is, the total work is $O(n)$. Similarly, we can calculate the span via a simple recurrence:

$$S(n) = S(n/2) + O(1) = O(\log n)$$

Notice that in calculating the span, we only have to look at the more expensive of the parallel recursive calls (since we are looking for the length of the critical path). In this case, the two recursive calls are both the same. Thus we get a span of $O(\log n)$.

Finally, using our (unproven) claim about good schedulers, we conclude that on a system with p processors, this summation algorithm will run in time $O(n/p + \log n)$, i.e., approximately the same result we got before from the more fine-grained analysis.

We also learn something else: this is a very parallel problem. We can useful exploit up to $W/S = n/\log n$ processors. If we had a very large system with $p = n/\log n$, the algorithm would run in time $O(\log n)$.

Notice that this model (like all models) is a simplification of the real world. It ignores the differing communication costs among processors. It ignores memory contention. It ignores caching. However, it provides a simple way to specify and analyze parallel algorithms in a way that can be readily translated to real multicore / multisocket machines.

3 Parallel Trees

One useful parallel data structure is a set, i.e., something that lets you insert and delete items, while at the same time merging two sets, finding the difference between two sets, taking the intersection, etc. That is, we would like to support the following operations (with the specified work W and span S):

- Insert / delete: $W = O(\log n)$, $S = O(\log n)$. These operations allow you to insert and delete items into the set.
- Divide: $W = O(\log n)$, $S = O(\log n)$. This operation allows you to divide a set into two (roughly) even pieces.
- Union: $W = O(n + m)$, $S = O(\log n)$. This operation allows you to merge two sets, where n is the size of the larger set and m is the size of the smaller set. Notice that merging two sets is more expensive, as it potentially requires eliminating duplicates.
- Difference: $W = O(n + m)$, $S = O(\log n)$. This operation finds the set difference between two sets, where n is the size of the larger set and m is the size of the smaller set.
- Intersection: $W = O(n + m)$, $S = O(\log n)$. This operation finds the set difference between two sets, where n is the size of the larger set and m is the size of the smaller set.

Notice that the latter (more expensive) operations run in time $O((n + m)/p + \log n)$ on a system with p processors.

One standard implementation of such a parallel structure is to begin with a standard balanced binary search tree, such as a red-black tree or a treap. In this case, insert/delete/search are implemented by a single processor in $O(\log n)$ time, as needed.

These binary search tree algorithms also support split and join primitives that act as follows:

[1]This claim depends on the precise parallel paradigm that you are working in.

- Split: takes a given key k and divides the tree into two new trees T_1 and T_2, where T_1 contains all the keys less than the input key k and tree T_2 contains all the keys greater than or equal to the input key k.
- Join: takes two trees T_1 and T_2 where all the keys in tree T_1 are less than all the keys in tree T_2; it produces a new balanced tree.

These operations are supported in red-black trees and treaps in $O(\log n)$ time. (See the earlier problem set exercise regarding supporting these operations in an (a, b)-tree. In fact, joining can be faster in some cases, depending on the relative heights of the trees being joined.) Using the split operation, it is easy to support the *divide* operation, simply splitting the tree based on its root.

The remaining operations are more difficult, as they require carefully comparing all the items in the two trees (to avoid duplicates, overlaps, etc.). Consider the following implementing of the *union* operation, using a divide-and-conquer strategy: Merge is implemented in a divide-and-conquer fashion, wherein you:

- Let k be the root key of tree T_1.
- Split tree T_2 based on k into t_1 and t_2.
- In parallel: recursively union the left subtree of tree T_1 with t_1 and the right subtree of tree T_1 with t_2.
- Now join the two trees (since every element in T_1 is $< k$ and every element in T_2 is $> k$) and re-insert k.

Notice that this provides an elegant parallel solution to the problem of finding the union of two trees, dividing up the problem into several pieces that can solved in parallel. It is a good example of the power of fork-join parallelism; it would not necessarily be easy to have explicitly divided the problem into pieces to be solved in parallel.

In order to analyze the performance, let us assume that the balanced binary tree guarantees that the tree is weight balanced, i.e., that the ratio of the number of nodes in the left and right subtrees is at most 10. (Notice that this is not true of a red-black tree; it is true in expectation for a treap.) We can then calculate the span via the following recurrence, where n is the number of nodes in tree T_1 and m is the number of nodes in tree T_2:

$$S(n, m) \leq S(9n/10, m) + O(\log n) + O(\log m) = O(\log^2 n + \log n \log m)$$

When $n \geq m$, this yields a span of $O(\log^2 n)$. With further optimization (outside the scope of this class) this can be reduced to $O(\log n)$.

Calculating the work is somewhat trickier, because we cannot ensure that tree T_2 is evenly split. In fact, the worst-case occurs when the tree is evenly split, but this requires some care to prove. Let n be the size of tree T_1, m be the size of tree T_2, let n_1 and n_2 be the sizes of the trees created by splitting tree T_1, and let m_1 and m_2 be the sizes of the trees created by splitting tree T_2:

$$W(n, m) \leq W(n_1, m_1) + W(n_2, m_2) + O(\log n) + O(\log m)$$

A simple analysis simply ignores tree T_2 (taking the worst-case where it divides into parts of size at most m) and yields:

$$W(n, m) \leq W(n_1, m) + W(n_2, m) + O(\log n) + O(\log m) = O(n \log m)$$

This follows because the tree T_1 always splits into two approximately equal parts, and so the recursion tree has $O(n)$ nodes, each of which has cost $O(\log m)$. (The additional $O(\log n)$ cost per node in the recursion tree sums to $O(n)$.) A tighter analysis yields cost $O(m \log(n/m))$ where n is the size of the bigger tree and m is the size of the smaller tree.

The other operations, i.e., set difference and intersection, can be implemented and analyzed in a similar fashion.

4 Breadth-First Search

We will consider the following basic paradigm for BFS: we maintain a frontier F, which is initially just the source node. At each step we mark all the nodes in F as visited, and we enumerate all the neighbors of nodes in F, adding all unvisited nodes to the new frontier F'. We continue this until the frontier is empty.

Algorithm 2: BFS$(G = (V, E), s, n, m)$

```
1  Initialize frontier set: $F \leftarrow \{s\}$.
2  $d = 0$
3  **repeat** until $F$ is empty:
4      Initialize new frontier: $F' \leftarrow \emptyset$.
5      **for each** node $u \in F$:
6          $distance[u] = d$
7          $visited[u] =$ **true**
8          **for each** neighbor $v$ of $u$:
9              **if** ($visited[v] =$ **false**) **then** $F'.add(v)$
10     Update $F$: $F \leftarrow F'$.
11     $d = d + 1$
```

4.1 Parallel Algorithm

To parallelize this algorithm, we want to process the frontier F in parallel. Imagine performing the following steps:

- Split F into p (almost) equal sized parts. This can accomplished by calling splitting F into two pieces $\log p$ times.
- Each processor individual produces a new fronter F', resulting in $F_1, F_2, \ldots, F_p$.
- Merge together the F_i to produce the new frontier F. This can be done via $\log p$ merges.

Let us re-write the algorithm in the fork-join model:

Algorithm 3: ParallelBFS$(G = (V, E), s, n, m)$

```
1  Initialize frontier set: $F \leftarrow \{s\}$.
2  Initialize complete set: $D \leftarrow \emptyset$.
3  $d = 1$
4  **repeat** until $F$ is empty:
5      Mark nodes in $F$ as done: $D = D \cup F$
6      $F = \text{ProcessFrontier}(F, d)$
7      $F = F \setminus D$
8      $d = d + 1$
```

4.2 Analysis

For ProcessFrontier: assume there are n' nodes in F and m' outgoing edges from nodes in F. Then we can calculate the work and span as follows:

$$W(n', m') = W(n_1, m_1) + W(n_2, m_2) + O(m' \log m') + O(\log n') + O(1)$$

Algorithm 4: ProcessFrontier(F, d)

```
1  if $|F| = 1$ then
2      Let $u$ be the unique node in $F$.
3      $distance[u] = d$
4      return $nbrs[u]$
5  Split $F$ into two parts: $\langle F_1, F_2 \rangle = F.split()$
6  in parallel
7      One:
8          $F_1 = \text{PROCESSFRONTIER}(F_1, d)$
9      Two:
10         $F_2 = \text{PROCESSFRONTIER}(F_2, d)$
11 return $\text{UNION}(F_1, F_2)$
```

Here we assume n_1 and m_1 are the nodes/edges in F_1 and n_2 and m_2 are the nodes/edges in F_2. We spend time $O(\log n')$ dividing set F into two subsets F_1 and F_2. Then, there are two recursive calls, producing new sets F_1 and F_2. Notice that these sets together contain at most m' elements, because we add only one item to the new frontier for each edge outgoing from F. Thus, there is $O(m' \log m')$ work merging the sets.

Notice that we do not know exactly how the nodes are split between the two sets when we perform the recursive call. What we do know is that $n' = n_1 + n_2$ and $m' = m_1 + m_2$. Moreover, we know that n' is split into two approximately equal sets. This latter fact ensures that the recursion tree is only of depth $O(\log n')$ before we get to the base case:

$$W(1, m') = O(1)$$

In this case, when there is only one node left in the set F, the cost is just $O(1)$, i.e., we return the list of neighbors. (We assume that we can return the entire list of neighbors without further processing. If more work is required to return a list of neighbors, then a slight modification is needed.)

Assume we begin with a graph containing n nodes and m edges and that $m \geq n$ (i.e., the graph is connected). To solve this recursion, observe that at each level of the recursion tree, the number of edges add up to m, i.e., the total cost per level of the recursion tree is at most $O(m \log m)$. Since there are $O(\log n)$ levels, the total cost is $O(m \log n \log m) = O(m \log^2 n)$. (The additional terms (i.e., the $O(\log n')$ and $O(1)$) all sum to $O(m)$.)

Claim 1 *The total work done during ProcessFrontier is $O(m \log^2 n)$.*

(Again, this analysis is somewhat sloppy and can be improved.)

For calculating the span, assume that when the set F is divided into two parts where the larger has no more than $9n'/10$ nodes. (This follows from the implementation of the divide primitive on the set.) We also assume that in the worst-case, all the neighbors m' are in one of the sets, and that the span for all the set manipulations is $O(\log^2 n)$ (i.e., a somewhat loose analysis). The base case, when $n = 1$, is:

$$S(1, m') = O(\log m')$$

In general, the span is:

$$S(n', m') = S(9n'/10, m') + O(\log n') + O(\log m') + O(\log^2 m')$$

Solving this, assuming that initially $m \geq n$ (and at most n^2), we see that the span is:

$$S(n, m) = O(\log^3 n)$$

Note that this analysis is somewhat loose, as it assumes that the the span of the union operation is $O(\log^2 n)$ (rather than the tighter, more difficult analysis).

Claim 2 *The total span of ProcessFrontier is* $O(\log^3 n)$.

We can now complete the analysis for the parallel BFS algorithm. Assume the initial graph has n nodes and m edges. We begin by calculating the work. In each iteration, if there are n' nodes in the frontier and m' edges adjacent to the frontier, we have already found that there will be $O(m' \log^2 n)$ work processing the frontier, and there is an additional $O(m' \log n')$ to take the set difference. Thus the total cost for each iteration is $O(m' \log^2 n)$.

Now, notice that over all the iterations, each node is counted at most once in a frontier (after which it is done and not included in any future frontiers). And notice that each of the edges is counted at most twice as adjacent to a frontier: once for each endpoint. Thus all the n' and m' values sum to n and $2m$ respectively, yielding to a total work of $W = O(m \log^2 n)$.

Claim 3 *The total work of BFS is* $W = O(m \log^2 n)$.

We also know that for each iteration of the main loop, the span is $O(\log^3 n) = O(\log^3 m)$. Let D be the diameter of the graph. After D iterations, the algorithm will complete and every node will be visited. (This is a property of BFS.) Thus, we can conclude:

Claim 4 *The span of BFS is* $S = O(D \log^3 n)$.

If we run the BFS algorithm on a system with p processors (and a good scheduler), the total running time will be:

$$O((m/p) \log^2 n + D \log^3 n)$$

4.3 Discussion

To compare, notice that sequentially BFS take $O(n+m)$ time. This improves it by a factor of $O(p/\log^2 n)$ (assuming D is small compared to m/p). Is that better? Obviously, it depends on p. Also, this is where the loose analysis hurts us: clearly, the algorithm would look more useful without the log-factors.

Notice that doing better than $\Omega(D)$ time is basically impossible: you need to explore an entire path from end to end. Similarly, if you try to parallelized DFS, you get running time $\Omega(n)$: it does not parallelize well. So in a parallel setting, if you have a choice between BFS and DFS (e.g., to count the number of connected components), you will want to use BFS!

5 Map-Reduce

To this point, we have been talking about "fork-join" style parallelism: in this case, one writes parallel code by explicitly executing several procedures in parallel (e.g., forking off several parallel processes) and then waiting for them to return (i.e., to re-join the main stream of computation). For multicore programming, this is a powerful and useful paradigm.

For large-scale cluster computing, however, a different paradigm has come to dominate: Map-Reduce. One key advantage of Map-Reduce is that it focuses on data, how it is organized, and how it evolves. (By contrast, the fork-join model is more focused on computation.)

The Map-Reduce framework is designed to run on a well-connected cluster of servers; this is often referred to as Warehouse Scale Computing. It was originally invented by Google for indexing the web, and subsequently used heavily by Google, Yahoo, Facebook, etc. It has become one of the predominent models for handling big data. It is designed to scale to 1000's of machines and 10,000's of disks. It uses mostly commodity hardware, commodity network infrastructure, and has built-in fault-tolerance (due to the many machines). A typical setup might include:

- Each node (e.g., for Yahoo terasort) has 8 2GHz cores, 8GB RAM, and 4 disks (e.g., about 4TB).
- Nodes are connected on racks, with 40 nodes per rack connected by a 1Gbps switch.
- Racks are connected by 8Gbps switch.
- You might have between 25 and 100 racks, leading to a total of, say, 2000 nodes and 16000 cores.

You can see how this leads to a whole different scale of parallelism than we might hope to achieve on a single high-performance multi-core server.

The Map-Reduce framework has many implementations, the most common of which is Hadoop. Today we are focusing on Map-Reduce as an algorithmic model, and will not be paying too much attention to implementation details.

5.1 The Map Reduce Framework

The Map-Reduce framework suggests that a parallel program should be written in terms of only two primitives: Map and Reduce. A large part of the simplicity and power of this paradigm comes from limiting what you can do! A single phase of Map-Reduce consists of:

1. Map: execute the map function on all the input data (one key/value pair at a time), producing intermediate data.
2. Intermediate step: automatically re-organize the intermediate data and route it properly to the next server in the computation. (This is often referred to as a shuffle step.)
3. Reduce: execute the reduce function on the re-organized intermediate data, producing output data.

Your job as an algorithm designer is to write the map and reduce functions, and specify how many phases to execute.

The input data to the algorithm is assumed to be stored on shared storage. (For example, it may be stored on the Google File System (GFS).) It is either available as a large flat list, e.g., a big array of objects. (In practice, of course, the data is divided into smaller chunks divided among different servers.) For the purpose of designing an algorithm, we can imagine that the array slot $IN[i]$ holds the ith piece of input data. (Note that input data may be a single value or a large file; the only restriction is that it is small enough for a single core to operate on.)

Throughout the Map-Reduce execution, data is treated as a $(key, value)$ pair. Initially, we will treat each element in the input as an individual key/value pair, i.e.: $(i, IN[i])$. As the algorithm progresses, it will modify key/value pairs and produce new key/value pairs.

When the Map-Reduce execution terminates, all the key/value pairs are written out to disk again in sorted order. That is, the Map-Reduce system sorts all the values by key and writes the values in that order. (Notice that since Map-Reduce typically has built-in sorting, there is not much need to consider designing parallel Map-Reduce sorting algorithm.)

The Map Function. The Map function is designed to perform parallel computation on key/value pairs individually. It takes as input one key/value pair and outputs one (or more) key/value pairs.

For example, imagine you want to square every value in the input, then you can might:

```
function map(key, value)
     // Do some computation.
     // Produce new key' and value'
     value' = value * value
     emit(key, value')
```

Or, alternatively, you might want to choose new keys for your values. For example, maybe each value represents a node in a graph, and you want to produce a new key/value pair for each outgoing edge:

```
function map(key, value)
     // Assume each key is a node identifer.
     // Assume each value is a node in a graph.
     Node n = value
     For each nbr in n.nbrs do:
        emit(nbr, <n.nodeId, n.nbr.nodeId>)
```

This type of mapping routine essentially processes all the nodes to produce all the edges in the graph.

The general principle to keep in mind is that Map allows you to operate in parallel on all the key/value pairs. However, the procedure can only operate on exactly one key/value pair at a time.

The Reduce Function. The goal of the Reduce function is to combine key/value pairs. Specifically, it takes all the values that have the same key and processes them with a single Reduce function. When it is complete, it outputs one or more key/value pairs. The template for the Reduce function is:

```
function reduce(key, value1, value2, value3, ...)
```

The MapReduce system shuffles all the key/value pairs in the system, finding all the values that share the same key. For example, if prior to the Reduce step, there are the following key/value pairs in the system:

```
(2, x)
(1, y)
(2, z)
(3, w)
(2, v)
(1, u)
(3, t)
(1, s)
```

Then the shuffle step will organize these into three different sets: $\langle 1, \{y, u, x\}\rangle, \langle 2, \{x, z, v\}\rangle, \langle 3, \langle\{\rangle\, w, t\}\rangle$. Each of these three sets is processed by a single Reduce.

The overall structure of a phase. A single phase of Map-Reduce begins by executing a Map function, which processes all the key/value pairs in parallel producing a set of intermediate key/value pairs. Then it performs the shuffle, reorganizing the key/value pairs and group them by key. Finally, it executes the Reduce function, combining values with the same key. This then produces a final set of key/value pairs.

If we are executing multiple phases of Map-Reduce, then these output key/value pairs can be re-used as the input to the next phase. Otherwise, they are output to disk in sorted order. Typically, you might measure the performance of a Map-Reduce algorithm in the number of phases it requires.

Limitations. There is, of course, a trivial Map-Reduce algorithm for any problem: simply map all the input values to the same key, e.g., key 1; then write a Reduce function that solves your problem sequentially. In this way, any problem can be solved in exactly one phase of Map-Reduce. However, that solution exposes no parallelism at all! And that one phase will be very slow, since the entire computation is being performed by a single server.

There is a second somewhat more subtle problem with the sequential solution: all the data is being routed to a single server. In that case, the network bandwidth may become the bottleneck. More generally, thoughout a Map-Reduce computation, the key-value pairs are being constantly shipped from input disk to mappers to reducers to the output. This data movement has a cost, and may slow down your program.

From a pragmatic perspective, then, you want to limit: (1) the size of the data in a key/value pair, (2) the amount of data processed by a single reducer, and (3) the amount of data produced by a single reducer.

If a key/value pair is not too big, then it can be processed by a single map. If the number of key/value pairs is not too large and none of them are too big, then the shuffle step can sort the data efficiently. If the amount of data processed by a single reducer is not too big, then the data can be efficiently routed to the reducer and the reducer can complete its computation efficiently. Finally, if the amount of data produced by a reducer is not too big, then the data can be properly routed from the reducer to the output.

From an algorithm design perspective, we aim to limit: (1) the number of key/value pairs processed by a single reducer, (2) the space needed by a single reducer, and (3) the number of key/value pairs produced by a reducer by n^ϵ, for some fixed ϵ. For example, it may be reasonable to suggest that no single reducer should process more than $\sqrt{n}$ key/value pairs. At the same time, we typically limit each key/value pair to be size $O(\log n)$ and require the total number of key/value pairs to be polynomial, e.g., $O(n^2)$. Also, all the Map and Reduce functions should be efficient polynomial time algorithms. The belief is that an algorithm that satisfies these limitation can be efficiently implemented.

Given these limitations, it is likely reasonably to measure the performance of a Map-Reduce algorithm by the number of phases. We can conclude that the length of a phase is probably relatively short (compared to the whole comptutation), since each mapper or reducer is executing an efficient algorithm on a relatively small amount of data. And we can assume that the shuffling and sorting is reasonably efficiently implemented.

5.2 Example: Word Count

The original paper on MapReduce contained two examples. The first of these was counting words. Imagine you have a massive amount of text, e.g., all the web pages in the world. And you want to create an index indicating how frequently each word appears. That is, you want a sorted list of words where each has attached a count of the number of times it appears in your text corpus. (This relates closely to one of Google's initial motivations in developing MapReduce, i.e., indexing the web.)

From the perspective of MapReduce, we can imagine that the input consists of a flat file where $IN[i]$ is the ith word in the corpus. Each word may appear many times, of course, in this array. The eventual output should consist of key/value pairs $(word, count)$ where each word appears in exactly one pair and the count represents the number of times each word appears.

Map. The map function, here, is relatively simple: it takes each word and annotates it with a count:

```
function map(key, word)
     emit(word, 1)
```

After the map phase is complete, we have a set of key/value pairs where each word has its initial count: one.

Reduce. The reduce function needs to combine all the key/value pairs related to the same word. Recall, a reducer receives all the key/value pairs with the same key. In this case, the key is a word and so the reducer receives exactly the right pairs:

```
function reduce(word, count[])
     sum = 0
     for (i = 1 to count.size)
         sum = sum + count[i]
     emit(word, count)
```

The reduce function aggregates all the counts, producing a single new final count.

Notice one nice feature of this reducer is that it is associative: all the counts do not need to be summed at the same time by the same reducer. This allows the scheduler to compensate for the situation where there are too many key/value pairs with the same key. For example, consider the number of times the word 'the' appears on the web; it is highly likely that this one word will create a massive number of key/value pairs that may overwhelm any one reducer. It is better to distribute the load among many reducers. (Recall how in the fork-join model we built a tree to perform such additions in parallel.) Many implementations of Map-Reduce support this, and hence it is preferrable that reduce functions be associative.

5.3 Example: Join

A second example of Map-Reduce is performing a join between two sets. Imagine you have two sets: $A = (x_1, y_1), (x_2, y_2), \ldots$ and $B = v_1, v_2, \ldots$. The goal is to output all the y_i values that are "selected" by the B set, e.g., the set:

$$\{y_i \;:\; \exists j,\; x_i = v_j\}$$

Whenever there is some $x_i = v_j$, we should output that y_i. This type of join operation is immensely common in database applications, and is typical of many data processing applications. There are many ways it can be solved sequentially, e.g., by a double-loop, with a hash function, etc.

It is also a problem that is well-designed for MapReduce. In this case, there are two types of input values:

```
function mapA(key, x, y)
     emit(x, y)

function mapB(key, v)
     emit(v, BVALUE)
```

Here we are writing these as two different map functions. Of course in practice this might be written a single map function that differentiates which input it is receiving.

In essence, both of these map functions use the parameter being compared as the key. For the A set, the x_i values are the key, with the y_i values being the data. For the B set, we just have the v_i values as the key and use a special symbol to indicate that this value derives from the B set.

We can then design a reduce function:

```
function reduce(key, values[])
     if (BVALUE in values) then
        for (i = 1 to values.size)
            if values[i] != BVALUE then emit(key, values[i])
```

In this case, the reducer simply checks if there is any key/value pair that has the special symbol BVALUE. If not, then this key does not appear in the B input array and we should not output any of these values. Otherwise, if BVALUE does appear, then we want to emit all the values received (except for the special symbol BVALUE, of course).

This yields a one phase MapReduce algorithm for a relatively complicated parallel algorithm. (Essentially all the work has been transferred to the parallel sorting that happens in the shuffle phase.)

Notice that this reduce function is not associative: it can only execute once it has received all the specified data. And if there are too many identical values with the same key in the input array A, then it is potentially going to be slow.

5.4 Bellman-Ford Single Source Shortest Paths

Now we turn to using MapReduce to perform computations on a graph. We will look at the problem of finding a single-source shortest path. In the sequential setting, the typical efficient solution is Dijkstra's Algorithm. However, Dijkstra's Algorithm is not easy to parallelize since it depends on sequentially relaxing edges in a precise order. Instead we look at Bellman-Ford, an algorithm that yields much more natural parallelization.

Assume we have a weighted graph $G = (V, E)$ with weights w and a specified source s. Our goal is to compute the distacne of each node from s (along a shortest path). We will maintain for each node u an estimate of the proper distance $u.est$. We will assume that the graph is stored as an adjacency list and we can access the neighbors of u via the array $u.nbrs$. Bellman-Ford proceeds as follows:

Algorithm 5: BellmanFord$(G = (V, E), w, s)$

```
1 $s.est = 0$
2 **for each** node $u \in V$
3     $u.est = \infty$
4 **repeat** $n$ times
5     **for each** node $u \in V$
6         **for each** neighbor $v \in u.nbrs$
7             **if** $v.est > u.est + w(u, v)$ **then**
8                 $v.est = u.est + w(u, v)$
```

Typically, this algorithm has sequential cost $O(nm)$, as you need n iterations each of which enumerates all m edges.

However, there is a sense in which this algorithm is easy to parallelize: all the edge relaxations within an iterations can be performed in parallel. It does not matter in what order edges are relaxed (within an iteration), and it does not matter if they happen simultaneously. (This is what makes it very different from Dijkstra's.)

We are going to design a MapReduce algorithm where each phase of MapReduce executes a single iteration of Bellman-Ford. Assume, initially, we have a set of key/value pairs with one per node as follows:

- key = node identifier nid
- value = $\langle nid, est, nbrIds, nbrWeights\rangle$

We are assuming that each node has a unique id nid. Each node has an array of $nbrIds$, the ids of its neighbors, as well as $nbrWeights$, i.e., the weights of the edges to the respective neighbors. Initially, $est = 0$ for the source s and $est = \infty$ for every other node.

As an exercise, think about how to design a simple MapReduce phase that would create these initialized key/value pairs, given the graph in some other format.

We can now design the map and reduce functions to implement a single phase of BellmanFord. The Map function is designed to relax all the outgoing edges, sending the estimate of a node u to all of its neighbors: It is important

Algorithm 6: Map$(nodeId, u)$

```
1 emit $(nodeId, u)$ // Re-output the same key/value pair for next time
2 **for** $i = 1$ **to** $u.nbrIds.size$
3     emit $(u.nbrIds[i], u.est + u.nbrWeights[i]$
```

that this only requires accessing information about node u: its estimate, its neighbors, and the weights of the adjacent edges. It simply enumerates all the outgoing edges of u and sends to each of its neighbors a copy of its estimate combined with the weight of the edge in question.

It is now the job of the reduce function to combine the different estimate updates received. There are two different

Algorithm 7: Reduce($nodeId, values$)

```
1 // First, recover the node
2 $node = null$
3 for $i = 1$ to $values.size$
4     if $values[i]$ is a node then $node = values[i]$
5 // Next, find the minimum value
6 for $i = 1$ to $values.size$
7     if $values[i]$ is not a node then
8         if $node.est > values[i]$ then $node.est = values[i]$
9 // Output the updated node emit($node.id, node$)
```

types of key/value pairs that the reducer recieves: it receives nodes of the form $\langle nodeId, node\rangle$ and it receives estimate updates of the form $\langle nodeId, distance\rangle$. A practical implmeentation will have to differentiate these carefully. Here, we assumpe that the reduce function can readily distinguish these.

First, the reduce function for a given $nodeId$ finds the node data associated with that id. It enumerates through all the values until it finds the right one.

Second, it enumerates through all the estimate updates, finding the minimum value. If any value is smaller than the estimate of the current node, then it adopts that new estimate. In this we, all the incoming edge are relaxed.

Each iteration of these Map and Reduce functions serves to perform one parallel phase of BellmanFord. After n phases, the algorithm is complete and each node's estimate is equal to the minimum distance to the source s.

You might recall that Bellman-Ford can actually terminate early. If there is a complete iteration with no change in estimate, then you can stop. I'll leave it as an exercise to develop a MapReduce phase to detect termination. (The precise implementation of the control flow, i.e., how the MapReduce algorithm signals to the controller that it is ready to stop, depends on the implementation.)

There is one other issue that might be concerning: what if some nodes have a very large number of outgoing or incoming edges? For example, nodes may certainly have $\Theta(n)$ adjacent edges, and this may be very slow. Both the Map and Reduce functions here can be readily parallelized, but it is not necessarily easy to express this next level of parallelization within the MapReduce framework.

For example, we can certainly partition the job of mapping a given node u into several further pieces. In a preprocessing phase, each node with more than u neighbors could be mapped to several different key/value pairs, each of which was responsible for at most $\sqrt{n}$ outgoing edges. Similarly, the incoming edges for a node could be partitioned, and several copies of the node could be generated. Then each partition of incoming edges could be handled separately, and eventually all the node copies could be merged.

Even so, this additional level of parallelism rapidly gets clumsy. As a result, this style of MapReduce algorithm is most easily applied to graphs where the maximum degree is not too large.

Of course, if the maximum degree is sufficiently small, then also MapReduce provides little benefit. The MapReduce algorithm takes n phases, in the worst case. By contrast, Dijkstra's algorithm (run sequentially) takes time $O(m \log n)$. If $m = \Theta(n)$, then this is only a $\log$ factor slower. In all likilihood, the extra overhead of the MaxpReduce framework would make it slower than the sequential version.

6 PageRank

One of the key original application for MapReduce was calculating the PageRank. From Google's perspective, the web is simply a graph, where each page is a node and each link is an edge. The PageRank assigns a value to each node. It turns out that calculating PageRank is much like running BellmanFord.

6.1 Background

Let $G = (V, E)$ be a directed graph. The idea behind PageRank is as follows. Imagine you start at an arbitrary node u. Then, you either:

- With probability $1/2$, do nothing.
- With probability $1/2$, choose a random outgoing edge (u, v) uniformly at random. Go to node v.

Repeat these steps for a very long time. We can ask the question: after repeating t times, what is the probability that you are standing on node w? That probability is the pagerank of w.

There are a variety of interesting reasons for why this might be a good way to measure the importance of node w, there are many interesting connections between these probabilities and the spectral properties of the graph, and there is much work showing that this probability will converge as t gets large (and how fast it converges). We will ignore all of those issues for today.

For now, let's focus on one way to calculate this value. Let $P_t(u)$ be the pagerank of a node u after t iterations, i.e., the probability that the random walk stops at node u after t iterations. For the initial case, we will arbitrarily set $P_1(u) = 1/n$, i.e., assume that we start at a uniformly random node. (The initial distribution does not really matter, but this is perhaps likely to converge a little faster.)

Now, we want to calculate $P_{t+1}(u)$. Assume we have already calculated $P_t(v)$ for all $v \in V$. There are two ways that we can arrive at node u after step $t + 1$:

- We were already at node u at the end of step t, and we decided to stay with probability $1/2$. This occurs with probability $(1/2)P_t(u)$.
- We were at a neighbor v of u at the end of step t and we chose to go to node u. For neighbor v, this occurs with probability $P_t(v) \cdot (1/2) \cdot (1/|v.nbrs|)$. We then sum this probability over all the neighbors of u.

Putting these two cases together, we get a single expression for the PageRank of u at time $t + 1$:

$$P_{t+1}(u) = \left(\frac{1}{2}\right) P_t(u) + \left(\frac{1}{2}\right) \sum_{v \in u.nbrs} \frac{P_t(v)}{|v.nbrs|}$$

(If you read about PageRank online, you will find several variants of this, but this is the basic equation at the heart of all PageRank implementations.)

To calculate the PageRank, then, we begin with an initial P_1. We then recursively calculate this equation repeatedly until it converges, i.e., until from one iteration to the next the values change relatively little. The number of iterations needed depends significantly on the topology of the graph. For some graphs (e.g., expanders), they will converge in $O(\log n)$ iterations. For other graphs, it will take significantly longer, e.g., $\Omega(n)$ or $\Omega(n^2)$ iterations.

6.2 MapReduce for PageRank

Our goal, now, is to calculate the PageRank of a graph using MapReduce.

Notice that PageRank is very much like Bellman-Ford. We begin with a set of estimates, in this case, the estimated PageRank. In Bellman-Ford, we initialize each estimate to ∞. Here, we initialize each estimate to $1/n$. We then proceed in each phase to to calculate one iteration of PageRank.

Algorithm 8: Map($nodeId, u$)

```
emit $(nodeId, u)$ // Re-output the same key/value pair for next time
for $i = 1$ to $u.nbrIds.size$
    emit $(u.nbrIds[i], (u.est/u.nbrIds.size)$
```

As in Bellman-Ford, the map phase sends out its estimate to each of its neighbors. Notice that it divides its current PageRank by its number of neighbors, in order to make it easier for the destination to calculate its PageRank: The reduce phase is then responsible for aggregating those estimate to produce a new estimate:

Algorithm 9: Reduce($nodeId, values$)

```
// First, recover the node
$node = null$
for $i = 1$ to $values.size$
    if $values[i]$ is a node then $node = values[i]$
// Next, calculate the new pagerank
$sum = 0$
for $i = 1$ to $values.size$
    if $values[i]$ is not a node then
        $sum = sum + values[i]$
    $node.est = (1/2)node.est + (1/2)sum$
// Output the updated node emit($node.id, node$)
```

After sufficiently many iterations, the PageRank will converge.

6.3 Discussion

There has been much discussion and debate about the benefits and trade-offs associated with the MapReduce framework.

First, some argue about whether it is particularly novel: it is based on very traditional ideas in functional computing and the idea of using these to implement a parallel algorithm predated Google's invention of MapReduce. I think this argument is not immensely important: the Google development of MapReduce clearly spurred a massive change in how people write parallel algorithms, and was certainly immensely important.

Second, there is some argument as to whether this is a useful way to write algorithms. As we have seen, for applications like BellmanFord, it is not necessarily all that much simpler or more efficient than other parallel frameworks, e.g., the fork-join framework. (As an exercise, trying writing Bellman-Ford for the fork-join version.) And it feels difficult to control the level of parallelism. For example, when you want to switch from processing nodes in parallel to processing adjacency lists in parallel, it begins to feel messy. Trying to do less obviously data-parallel tasks can be tricky. And, of course, many of the costs are hidden in the scheduler and the amount of computation needed in a given phase.

On the other hand, MapReduce provides a remarkably simple way to implement some very sophisticated parallel algorithms. It identifies a very common algorithmic pattern for parallel algorithms and builds an entire programming language around it. And I suspect that part of the reason for its success is that it hides (and isolates) a key primitive in a lot of parallel problems: sorting. A huge number of efficient parallel algorithms depend on sorting (e.g., to partition data efficiently, or to assign processors, etc.). MapReduce hides the sorting within a shuffle phase. And by implementing a highly optimized parallel sorting routine, they simplify many other problems.

Finally, the success of the MapReduce paradigm itself speaks to its importance. It is, today, the de facto standard for how to write large-scale data analysis. Programmers enjoy working with it, and it yields good solutions to real problems.

From a theory perspective, there has been increasing work trying to understand the limits of MapReduce. What sorts of problems can we solve in $O(1)$ phases? What sorts of problems are impossible to solve efficiently (e.g., without overwhelming an individual mapper or reducer)? There has been a lot of exciting work, and there remain many interesting open questions.