Data Structures and Algorithms for Robust and Fast Parallel Graph Search

Umut A. Acar
Carnegie Mellon University & Inria
umut@cs.cmu.edu

Arthur Charguéraud
Inria & LRI, Université Paris Sud, CNRS
charguer@inria.fr

Mike Rainey
Inria
rainey@inria.fr

Abstract

Graph traversal based on algorithms such as depth-first search and breadth-first-search is a critical part of many applications. With the advent of multicore computers and the ability to furnish them with large shared random-access-memory, it has become possible to process large-scale graphs in parallel. For such parallel algorithms to be effective in general, they should be highly parallel, exposing as much parallelism as possible and remain work-efficient with respect to optimal serial graph-traversal algorithms. Since the topology of graphs can vary dramatically, simultaneously achieving these two properties is challenging.

In this paper, we present two highly parallel and work-efficient algorithms for performing graph traversals on directed (and undirected) graphs. Our first algorithm is a Parallel Breadth-First Search (PBFS) that improves over the state of the art by exposing more parallelism without detrimentally effecting work efficiency. Our second algorithm is a Parallel Depth-First Search (PDFS) algorithm that improves over the state of the art by guaranteeing work efficiency while remaining highly parallel. Both of our algorithms take advantage of our novel *frontier data structure* that supports very efficiently several key operations on the set of outgoing edges of visited vertices (the frontier), including push, iteration, split in half, and merge. Also based on this data structure, we present techniques for controlling granularity for improved practical efficiency.

We implement our algorithms and evaluate them carefully by considering both synthetic and real-world graphs and by comparing with the state of the art. The experiments show that, for the graphs considered, our algorithms remain robust, outperforming the state of the art except in a few cases, and dramatically outperforming them in certain cases.

1. Introduction

With the increasing use of parallel and multicore computers, high-performance parallel graph-search (or graph-traversal) algorithms have become increasingly important to a variety of areas, such as social networks [19, 21, 29], physical sciences [2], and even parallel garbage collection [17]. To be effective in such a broad range of application domains, parallel graph algorithms need to be *work-efficient*, that is, comparing favorably to optimized serial algorithms, not just in theory, but also in practice. Parallel graph algorithms that are not work efficient often yield poor speedups, especially with small to moderate numbers of cores, are wasteful in terms of resources such as hardware and energy, and might impose high overheads when given inputs with high diameter, such as long paths. Parallel graph algorithms must also be *highly parallel*, exposing as much parallelism as feasibly available so that speedups can be scaled to larger numbers of processors or cores. Last but not least, matching the diversity of the areas and the many shapes of topologies graph may possess, parallel graph algorithms should be *robust*. Robust algorithms are work efficient and highly parallel, for not just a specific class of graphs, but ideally for all graphs.

Despite having received a great deal of attention over many years [6, 18, 20, 23–25], designing and implementing work-efficient, highly parallel, and robust graph search algorithms remains a major challenge. In this paper, we present algorithmic and scheduling techniques to overcome this challenge. We then use these techniques to design and implement parallel depth-first-search (PDFS) and parallel breadth-first-search (PBFS) algorithms, and evaluate their effectiveness.

BFS and DFS algorithms both can be viewed as operating on a *frontier* data structure that contains the set of vertices to be visited next. In the serial versions of these algorithms, the frontier data structure can be implemented using off-the-shelf data structures such as stacks and queues. The parallel versions, however, require frontier data structures that can also be used to generate parallelism.

We present an *edge-weighted frontier* data structure for parallel graph search. Our frontier data structure allows assigning to each vertex in the frontier a weight and supports a weighted-split operation that partitions the frontier in a way to balance the total weight between the two halves. By using the out-degrees of the vertices as weight, we are able to generate work-balanced parallel tasks and precisely control their granularity, both of which are key to efficiency. We also present an asymptotically efficient implementation of our weighted-frontier data structure that remains competitive with the highly-optimized container data structures used in sequential graph-search algorithms.

To be work-efficient, essentially any parallel algorithm must be implemented with care to avoid creating an excessive amount of tiny parallel threads. This problem, also known as the granularity-control problem, can sometimes be solved by serializing loops that contain fewer iterations than some *threshold* value that is chosen to amortize the cost of creating parallel tasks. Since, however, graphs can be highly irregular, this approach does not work robustly in parallel graph algorithms because the cost of scheduling (load-balancing) parallel tasks can far outweigh their creation cost depending on the topology of the graph, making it impossible to find a threshold that works well for all graphs.

To overcome the granularity challenge, we present a technique for creating parallel tasks on demand based on the load of the system. The basic idea behind this approach is lazy splitting [14, 22, 27], which enables creating parallel tasks only when there is a demand for them. Lazy splitting has been applied in the context graph-search algorithms before [27]. What makes our approach different is that, by combining lazy splitting with our weighted frontier data structure, our algorithm is able to share, on demand, exactly half of the total instantaneous work, potentially expressed across nested parallel loops, by using out-degrees of vertices as weights. As our experiments show, these techniques lead to significant increases in performance compared to state-of-the-art algorithms.

Graph-search algorithms differ in the order they visit the vertices and edges of a given graph. Depending on the application, we may prefer one of the algorithm over another. For example, certain algorithms such as Bulk-Synchronous Parallel algorithms [28] can be naturally expressed on top of PBFS, which computes the distance of every vertex to a source vertex, processing the graph layer by layer. Other algorithms such as those used by garbage collectors are more naturally expressed as a PDFS [17], which enumerates the set of vertices reachable from a source vertex. We note that PDFS is not a faithful parallelization of the sequential DFS algorithm, because PDFS is not able to guarantee the same visit order as sequential DFS [24, 25]. Nevertheless, PDFS can be more work efficient and can expose more useful parallelism [12]. Moreover, PDFS can achieve better data locality in certain circumstances, making is crucial in applications such as parallel garbage collection [17].

Recent work on PBFS and PDFS which target multicore platforms has produced several major advances. Beamer et al. propose a *direction-optimizing* PBFS that accelerates certain graph traversals by exploiting characteristics of low-diameter graphs [4]. There is a growing literature on hardware-specific techniques, such as prefetching, compression, and lightweight synchronization, to accelerate PBFS [5] and PDFS [8]. Such graph- and hardware-specific techniques are largely orthogonal to ours in the sense that, in many cases, the specific optimizations can be applied in combination with traditional PBFS and PDFS techniques, with the exception of load balancing techniques that abandon dynamic for static load balancing. Of the recent work, however, we are aware of only two studies which propose algorithmic, hardware-independent solutions that are applicable to all graphs.

The first study, which is presented by Leiserson and Schardl, approaches the problem of making a scalable, work efficient implementation of PBFS [20]. Our work improves on their results by contributing a PBFS that exposes, in a robust fashion, all exploitable parallelism in the input graph, without sacrificing work efficiency. The second study, which is presented by Cong et al., uses an adaptive batching technique to determine the size of the parallel tasks to create based on the load of the system [9]. Relative to their work, we contribute a PDFS algorithm that is 1) work efficient in a robust fashion, 2) that eliminates the need for batching schemes, and 3) that adds the ability to exploit edge-level parallelism.

We present a C++ implementation and evaluation of our techniques. For our evaluation, we consider a range of real-world and synthetic graphs testing difficult cases, and compare our algorithms to the state of the art PBFS and PDFS algorithms. Our contributions include the frontier data structure, an adaptation of lazy splitting for parallel graph-search algorithms based on this structure, and the design, implementation, and evaluation of PBFS and PDFS.

2. Challenges and Ideas

We illustrate the challenges in developing robust, highly parallel, and strongly work-efficient parallel graph-search algorithms and the ideas behind our work by considering examples. The rest of the paper makes these ideas precise by describing them in more detail and illustrating their practical utility. For simplicity, we consider small example graphs to illustrate the main points; it is not difficult to generalize these example graphs to larger examples.

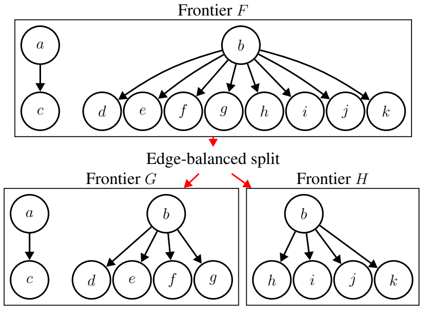

Figure 1. The edge-balanced split operation on the frontier.

Challenge I: representing frontiers. Graph search algorithms such as breadth-first search (BFS) and depth-first search (DFS) maintain a set of vertices called *frontier* that contains the set of unvisited vertices that are connected by an edge to an already visited vertex. They then visit the vertices in the frontier a specific order. The order itself is the primary difference between the breadth-first and depth-first traversals. Apart from the work performed when visiting a vertex, which can vary depending on the application, graph-search algorithms spend most of their time operating on the frontier. An efficient data structure for representing frontier is therefore key to effective parallel graph traversal.

In sequential graph-search algorithms, it suffices for the frontier to support two simple operations for inserting and deleting vertices into and from the frontier. In parallel graph search, however, these operations are not sufficiently powerful for creating parallelism. To create parallelism, an additional operation that splits the frontier into two halves suffices. Such a split operation can simply partition the frontier based on the number of vertices, but ideally it should also take into account the number of outgoing edges of each vertex in the frontier, and perform *edge-balanced* split operations. An edge-balanced split would partition the frontier into two parts such that each part has approximately half of the edges. This is important because, when processing a frontier, graph search algorithms perform work proportional to the number of out-edges of the vertices. In addition to the ability to split, merging or combining frontiers may also be necessary. Developing and implementing an efficient frontier data structure that can support these operations while remaining highly competitive with a sequential frontier data structure is a key challenge.

We address this challenge by presenting a frontier data structure that remains competitive with its simpler sequential counterpart while also supporting edge-balanced split, and merge operations. To this end, we view the frontier as a set of outgoing edges (rather than vertices) and use a sequence data structure that supports insert and delete operations at the two ends of the sequence as well as split operations efficiently. Our frontier data structure can be used to implement both DFS and BFS (possibly also their variants).

To guarantee strong work efficiency, we need to take care not to represent the edges explicitly, because this leads to disproportionate overheads relative to serial frontier data structure. We therefore rely on a hierarchical representation that represents the edges in the frontier as a set of vertices they originate from and use as a weight the out-degree of each vertex. In addition, we use two "carry" structures each consisting of a sequence of edges, which are implicitly represented as a pair of pointers. Figure 1 illustrates an example. The frontier F consists of the vertices a, b and implicitly their out-edges. Performing a split operation divides the frontier into two frontiers with equal number of outgoing edges (within a margin of 1), by dividing the edges of the vertices as necessary. In this example, each part contains a single non-empty carry consisting of a half of the edges of vertex b.

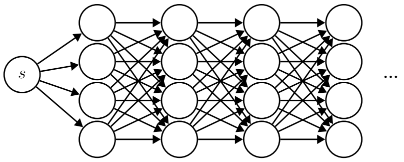

Figure 2. A large graph serialized by parallel BFS.

Challenge II: granularity control for irregular graphs. When programming essentially any parallel algorithm, it is critical to control the *granularity* of parallel threads so that we don't create too many parallel threads each with small amount of work. Controlling granularity is relatively simple in principle: all we have to do is to make sure that we avoid creating *small* threads. The challenge, however, is to determine the threshold amount of work that defines precisely what "small" means. If the threshold is too small, the overheads of managing parallel threads can be too high. If it is too high, then we may overly reduce parallelism and thus harm robustness.

In highly regular parallel computations, e.g., dense matrix computations, it can be relatively straightforward to determine the threshold because there is plenty of parallelism available at large granularity. Specifically, it often suffices to pick a threshold that is large enough to amortize the cost of thread creation, which can in turn be determined based on the architecture parameters. In irregular parallel computations such as graph algorithms, however, the structure of the computation can vary dramatically depending on the graph, making it a challenge to select the right threshold. This is because thread-scheduling costs, such as migration and synchronization of non-local threads, can far outweigh those of thread creation. Furthermore, increasing the threshold can amortize such costs but can harm robustness: there exists graphs for which increasing the thresholds would lead to significantly reduced parallelism. Thus it appears that we are doomed do choose between significant overheads or suboptimal parallelism.

Fortunately, the lazy splitting technique [27] can allow us to break this impasse. The basic idea behind lazy splitting is to create parallelism only when there is a demand for it. Specifically, during a parallel execution, each processor estimates the amount of parallelism available (exactly how this estimation is performed depends on the scheduler) and creates parallel threads only if there are idling processors. Prior work shows this technique can be profitably applied to parallel loops, but can be challenging if the loops may be nested [27], which is the case in algorithms such as BFS. Since, however, our frontier data structure essentially allows us to flatten the nested loops in BFS into a single, flat loop, we can (and do) apply lazy splitting in PDFS and PBFS.

Challenge III: breadth-first search. The sequential BFS algorithm stores in the frontier the set of vertices at a given distance from the source. Initially, the frontier contains the source vertex. At each iteration, the algorithm builds the next frontier by processing the outgoing edges of vertices in the current frontier. One way to parallelize BFS is to process the vertices in the current frontier in parallel, while also processing the out-edges of each vertex in parallel. For this algorithm to be work-efficient, it is important to avoid parallelizing loops (over vertices and edges) that contain fewer iterations than a "threshold" whose exact value is determined based on hardware so as to parallelize as much as possible, while avoiding to create too many small parallel threads.

Unfortunately, the parallelization strategy of considering the vertices and edges separately as described above is not robust because it can lead to suboptimal parallelism and speedups by serializing large computations. For example, our experiments show that an algorithm that uses this strategy can suffer as much as 10x decrease in speedups in certain graphs (Section 7). The problem is that if we sequentialize each loop when it contains smaller than the threshold value of K iterations, we can end up parallelizing K^2 iterations. (Choosing different threshold values might improve the precision slightly but would not solve the problem.) As an example, let's assume a threshold of 4 and consider the graph shown in Figure 2. In this graph, note that the frontier always contains 4 or fewer vertices and each vertex has 4 or fewer edges. Thus, at any and all levels, the loop over the vertices in the frontier and the edges will be processed serially, leading to serial processing of the whole graph, regardless of its size.

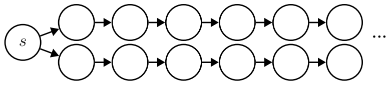

Figure 3. Example requiring aggressive work sharing.

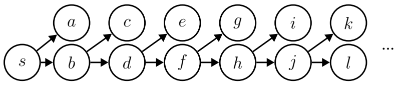

Figure 4. Problematic example for aggressive work sharing.

To overcome this challenge, we follow a different approach to parallelizing BFS. Instead of parallelizing over the vertices and edges separately, we use our edge-weighted frontier data structure to view the frontier as a set of outgoing edges (rather than vertices) and perform a parallel loop computation over the edges by using a single threshold. This approach allows effectively exploiting the parallelism available in the graph, significantly improve speedups in some cases (Section 7). For example, with a threshold of 4, in the graph shown in Figure 2, our algorithm creates 4 parallel threads for each frontier, each of which consists of 4 edges of work.

Challenge IV: parallel depth-first search. In PDFS [24, 25], each processor stores the vertices in the frontier in a stack data structure. The algorithm starts by placing the source on the stack of a processor. At each step, a processor removes the vertex on the top of the stack and visits it by pushing all its out-neighbors onto the stack. For load balancing, the scheduler transfers a subset on the vertices in the stack of a processor to the stack of another processor (typically, half of the vertices but the specifics vary depending on the implementation).

Intuitively, we might think that if the frontier contains a small number of vertices, it would not be worthwhile to share work with other processors. This is not the case, however; in fact it can be important to share even a single vertex. To see why, let's consider the graph shown in Figure 3. In parallel DFS, two processors can in principle traverse the two chains in parallel, after sharing a single vertex. To enable such a parallel execution, we must enable sharing a single vertex, even when the frontier is very small. Unfortunately, such aggressive sharing can lead to suboptimal performance. For example, consider executing on two processors the algorithm in the graph shown in Figure 4 with the aggressive sharing policy. In such an execution, it is possible for the vertices b, d, f, h, j, l to be shared with the other processor. Since such sharing actions involve communication between processors, they can significantly increase the run time by effectively shuttling the main spine of the graph between the two processors. In fact, in this and in similar examples, parallel run time can far exceed that of the sequential.

We thus have a dilemma. On the one hand, we need to share possibly small amounts of work (even as little as a single vertex). On the other hand, we need to take care that such aggressive sharing does not lead to prohibitive overheads. When the shape of the graph is known in advance, it might be possible to devise heuristics for controlling sharing. In this paper, we are interested in general-purpose techniques that work well for all inputs. We therefore solve this dilemma by using amortization technique: each processor shares work only if either (1) the work load shared

```
class frontier { // interface
  frontier()
  int nb_edges()
  void push_edges_of(int vertex)
  void split(frontier& other)
  // only used by parallel BFS
  void merge(frontier& other)
  // only used by eager parallel BFS
  void iter(body_type body)
  // only used by parallel DFS and lazy BFS
  int iter_pop_nb(int nb, body_type body)
}
where type body_type = void body(int src, int dst)
```

Figure 5. Interface for the frontier data structure.

exceeds some threshold, or (2) the processor has already performed some predetermined amount of work locally since the last time it shared work.

3. Edge-Weighted Frontiers

In this section, we present our frontier data structure, which can support merge operations and edge-weighted splits efficiently, both in theory and in practice.

3.1 The Interface

Figure 5 shows the interface for our frontier data structure. The operation `frontier` constructs an empty frontier; the operation `nb_edges` returns the number of edges in the frontier; the operation `push_edges_of` pushes all the out-edges of the given vertex into the frontier; the operation `split` carves out half of the edges into an independent frontier data structure; the operation `merge` transfers all the edges of a frontier into another one. We assume that the operation `merge` is never called after a `split` operation. Our parallel graph algorithms don't need to interleave these operations.

In addition, the data structure supports two forms of iteration. The operation `iter` iterates over all the edges in the frontier; the operation `iter_pop_nb` iterates over `nb` edges (or fewer, depending on the availability), and removes each edge considered from the frontier. Note that `iter_pop_nb` returns the number of edges that were actually processed. Remark: the data structure maintains a stack order on the edges. Such ordering is not necessary in BFS and is probably also not necessary in DFS because (non-deterministic) load-balancing can alter traversal order.

We next describe how to implement this data structure efficiently, such that `push_edges_of` and iteration operations are nearly as fast as an optimized sequential frontier data structure, and such that `split` and `merge` run in logarithmic time. To this end, we first summarize a recently proposed weighted-sequence data structure and then describe our implementation of the frontier data structure based on the weighted sequence data structure.

3.2 Splittable and Catenable Weighted Sequences

A splittable and catenable weighted sequence data structure supports push and pop operations at the two ends of the sequence, while also allowing us to put a weight on each item, splitting sequences at a specified weight, and concatenating sequences. In addition, the data structure allows iterating over all elements.

Recent work [3] gives a asymptotically efficient and practically fast, catenable and splittable weighted sequence data structure by using a chunking and a bootstrapping technique that allows representing the sequence data structure as a shallow tree. The data structure, called *bootstrapped chunked sequence*, stores a sequence of weighted items. Perhaps the most interesting operation for our purposes is the operation `split_at`, which takes a weight `w` and a sequence S and divides S into three parts: S_1, $\{x\}$, and S_2, in such a way that the total weight of S_1 is less than `w` and that the weight of $S_1 \cup \{x\}$ is greater than or equal to `w`.

Bootstrapped chunked sequences ensure practical efficiency by storing items in fixed-capacity chunks (represented as arrays). A *chunk size* parameter, called K, controls the size of the chunks; typical values for K on modern machines include 256 or 512. For a given K, the concatenation and split operations have a cost bounded by $O(K * \log_{K/2} n)$. This cost is in practice close to $O(\log_2 n)$ operations on binary trees, because $\log_{K/2} n$ is much smaller than $\log_2 n$, and because the constant factor associated with the multiplicative K is very small (chunks manipulation relies on highly-optimized *memcpy* operations).

In general, the worst-case asymptotic space usage of chunked sequences is $(2 + \frac{O(1)}{K}) * n$. However, when concatenation is not used, or when the order of the items in the sequence is not relevant (i.e., for a bag semantics), the bound can be improved in such a way as to guarantee asymptotic space usage of $(1 + \frac{O(1)}{K}) * n$, which, for practical values of K, is very close to optimal. In such situations, concatenation and split only cost $O(K * \log_K n)$.

3.3 The Implementation

We implement splittable, catenable edge-weighted frontiers on top of bootstrapped chunked sequence. The basic idea is to represent a frontier as a triple consisting of *vertex-sequence* and two *ranges* of edges. A vertex sequence is represented as a bootstrapped chunked sequence of vertices, where each vertex has a weight that matches its out-degree. A range of edges corresponds to a contiguous subset (subsequence) of the outgoing edges of a given vertex. A range is represented as a vertex and a pair of indices marking the start and the stop of the range.

To implement the frontier operation `push_edges_of`, we push the vertex given to the vertex-sequence. The operation `merge` is assumed to only be called on frontiers for which the two ranges are empty —e.g., on frontiers constructed using only `push_edges_of` and `merge`. We implement `merge` by concatenating the vertex-sequences of the frontiers.

The operation `split` transfers half—the smaller half in case the cardinality is not even— of the edges to another frontier data structure, which is assumed to be initially empty. If the first range contains at least half of the edges, we simply split this range and transfer a subrange to the other frontier. Else, if the second range contains at least half of the edges, we tranfer the appropriate subrange from it. Otherwise, we need to split the sequence of vertices. First, we transfer all of the second range to the other frontier. Then, we split the sequence of vertices in three parts: vertices that remains in the bag, vertices that go into the other bag, and one vertex which contains the median edge. We consider the full range of edges associated with this vertex and split this range at the appropriate position, storing the left subrange into the second range of the current frontier and storing the right subrange to the first range of the other frontier.

The operation `iter` iterates over the edges in the first range; then, for each vertex stored in the vertex sequence, it iterates over the edges of this vertex; finally, it iterates over the edges in the second range. The function `iter_pop_nb` follows a similar structure, but returns after processing `nb` edges, and pops the edges from the frontier as it processes them. The challenge in implementing this function is that efficiency is critical in the loop over the edges—we are careful to not perform any nontrivial additional operations compared with the corresponding loop in the sequential DFS algorithm.

3.4 Efficiency in Theory and in Practice

Based on the known bounds of the weighted-sequence data structure, and based on the fact that operations on ranges can be performed in constant time, it is straightforward to prove the following theorem, which bounds the asymptotic cost of the operations on the frontiers.

Theorem 3.1 (Efficiency of the frontier data structure) *Consider a chunk size parameter K for the underlying weighted sequence data structure. Assume that `merge` is allowed to reorder edges outgoing from distinct vertices. Recall that `merge` is assumed to never be called after a `split`.*

- *`nb_edges` is $O(1)$.*
- *`push_edges_of` is $O(1)$.*
- *`merge` and `split` are $O(K * \log_K n)$.*
- *`iter` and `iter_pop_nb` costs $O(1)$ per edge enumerated, in addition of the cost of actually processing the items.*
- *The asymptotic space usage is $(1 + \frac{O(1)}{K}) * n$, close to optimal.*[1]

In addition to good asymptotic bounds, the frontier data structure accepts a practically fast implementation by using the existing fast implementation for the weighted sequences [3] and carefully minimizing the interaction between the two ranges. In particular, we were careful in implementing the function `iter_pop_nb` to not introduce conditionals in the critical loops. Overall, the constant factors of the function `push_edges_of` and with the iterators are not too far from those of the push and iteration operations on plain arrays. These small constant factors are the key to achieving strong work efficiency.

4. Preliminaries

4.1 Scheduling interface

For plain fork-join programs, we do not need to make particular assumptions about the scheduler. We only need to assume the existence of a function called `fork2`, which takes two continuations as arguments, and only returns once the two continuations have executed —possibly in parallel.

```
void fork2(thread_type t1, thread_type t2)
   where thread_type = void f(void)
```

For parallel programs involving lazy splitting, however, we need a richer interface to the scheduler. This interface appears below, and is explained next.

```
void acquire(frontier& fr)
void has_incoming_query()
void reply(split_type split)
   where split_type = void split(frontier& other_fr)
```

A processor that runs out of work calls the function `acquire` in order to make queries to busy processors. This function may communicate the address of the frontier of the idle processor, so that the busy processor may directly transfer data into this frontier, without performing unecessary copy operations. Note that, while calling the `acquire` function, idle processors are blocking incoming queries from other idle processors.

Busy processors need to poll for serving queries. More precisely, they are responsible for periodically calling the function `has_incoming_query`, in order to check whether they received a query from an idle processor. The processor may then call the function `reply` to share some work (possibly none). The function `reply` is presented using a callback argument, which allows the busy processor to obtain the address of the idle processor's frontier data structure, so as to be able to migrate items into it.

Note that it is possible for the busy processor to refuse to share its work. In any case, at the end of the call to the function `reply`, the idle processor that made the query receives a notification that its query was processed.

Remark: we assume in this paper that an idle processor queries at most one other processor at any given time, and that busy processors can be delivered at most one query at a time. More complex communication schemes could be used, but it is unclear whether they would improve the performance on current multicores.

4.2 Graph representation and marking of vertices

For both BFS and DFS traversals, we assume the graph to be represented by an adjacency list, whose signature is as follows.

```
bag<int> neighbours[nb_vertices]
```

We also rely on an array of booleans, which we call `visited`, in order to mark the nodes that have been visited. Note that, for BFS, we may actually want to store the distance at which the vertices are visited, but we drop this information here as it is orthogonal to the problem of parallelizing the algorithm.

```
bool visited[nb_vertices] = { false, false, ... }
```

When considering an edge, the traversal algorithms check whether the target vertex has already been visited or not. If not, the vertex is marked as visited. For sequential algorithms, we use a function `attempt_first_visit` to perform this action. The function, shown below, returns a boolean indicating whether the node is being visited for the first time.

```
// non atomic version
bool attempt_first_visit(int target)
  if visited[target]
     return false
  else
     visited[target] <- true
     return true
```

For parallel algorithms, however, we need to implement the function differently in order to prevent data races, which may occur when two processors discover a given vertex at the same time. Indeed, we do not want both processors to add a same vertex to their frontiers. To prevent such races, we rely on the atomic compare-and-swap (CAS) operation. This operation applies to a memory cell and, atomically, reads the content, compares it with a given value, and updates it with a that given value it the comparison succeeds. The compare-and-swap operation returns a boolean indicating whether it succeeded. The atomic version of `attempt_first_visit` is therefore implemented as follows.

```
// atomic version
bool attempt_first_visit(int target)
   return cas(&visited[target], false, true)
```

Remark: when traversing a graph in a practical application, we may want to perform some processing on each vertex as the moment it is discovered for the first time; Such processing may me performed when the function `attempt_first_visit` returns true.

Observe that, during a PBFS or PDFS traversal, processors are only interacting with each other in one of two ways: when they atomically read or write in the array marking visited vertices and when they transfer pieces of frontier from a processor to another in order to perform load balancing.

5. Parallel breadth first traversal

5.1 From sequential to parallel BFS

PBFS follows the same structure as the traditional two-stack version of sequential BFS, which we recall next. The algorithm makes use of two bags: one to represent the previous frontier (holding vertices at distance d) and one to represent the next frontier (holding

[1] If we disallow `merge` to reorder the edges, then the bound for space usage is $(2 + \frac{O(1)}{K}) * n$, and the cost of `merge` and `split` is $O(K * \log_{K/2} n)$. However, since only our parallel BFS algorithm uses `merge`, and since for this algorithm the order of processing of the edges is unspecified anyway, we allow reordering to take advantage of the better bounds shown above.

vertices at distance $d + 1$). Until the current frontier is empty, the algorithm iterates over the edges associated with the vertices of the current frontier in order to potentially discover new vertices and push them into the next frontier. This processing is performed using an auxiliary function called `step`.

```
void bfs()
  bag<int> cur = bag()
  bag<int> next = bag()
  cur.push(source_vertex)
  while not cur.empty()
    step(cur, next)
    cur.swap(next)
```

For sequential BFS, the function `step` is implemented as the function `step_seq`, which simply uses two nested `for` loops to process all the outgoing edges, and makes calls to the function `attempt_first_visit` in order to visit the target of the edges. Note that the bags are here passed by reference (symbol `&`).

```
void step_seq(bag<int>& cur, bag<int>& next)
  foreach vertex in cur
    foreach target in neighbours[vertex]
       if attempt_first_visit(target)
          next.push(target)
```

To derive a parallel version of BFS, let us begin with the parallelizion of the loop over the vertices of the current frontier. To that end, we implement the function `step` using a recursive function called `step_par`. This function, whose code is shown below, applies the divide-and-conquer approach to recursively split the current frontier in halves until it becomes small enough to be processed sequentially. The end of recursion is controlled by a parameter called `vertices_cutoff`. Below this cutoff, the vertices are processed sequentially. Above the cutoff, two subtasks are generated, one for each half of the frontier. These two tasks are forked as recursive function calls, which may potentially run in parallel.

The division of the frontier in two halves is implemented using a method called `split`, which we assume provided by the bag data structure. Those two tasks contribute to the discovery of vertices, stored in frontiers called `next` and `next2`, which, after the join, are merged using a `merge` operation on bags.

```
void step_par(bag<int>& cur, bag<int>& next)
  if cur.size() <= vertices_cutoff
    step_seq(cur, next)
  else
    bag<int> cur2 = empty
    bag<int> next2 = empty
    cur.carve_half(cur2)
    fork2((fun _ → step_par(cur,next)),
          (fun _ → step_par(cur2,next2)))
    next.merge(next2)
```

It is key to observe that the setting of `vertices_cutoff` is critical to the efficiency of the algorithm. On the one hand, if the value is too small, then the overhead of the operations `carve_half`, `fork2`, and `merge` may be significant relative to the cost of `process_vertex_seq` (which may process as little as zero or one edge), and the overall program will suffer from poor constant factors. On the other hand, if the value of `vertices_cutoff` is too large, then the algorithm would process in sequence large numbers of vertices, potentially reducing the parallelism exposed.

The above algorithms exposes parallelism at the level of vertices, but not at the level of outgoing edges of each vertex. In order to expose as much parallelism as possible for graphs in which the arity of the vertices is not bounded by a small constant factor, we need to also parallelize the processing of the edges. As we have explained in the challenges section, we can parallelize the loop on the edges using a divide-and-conquer recursive function, just like we did for the vertices. However, as we argued, being able to exploit parallelism both at the vertex and at the edge level independently in general does not allow exploiting all the parallelism available. We next present our solution.

5.2 Parallelization at the edge level, using the frontier

The key idea is to split the frontier according to the number of edges, and not just according to the number of vertices. This splitting is made possible by the introduction of our frontier data structure. (Recall the interface given in Figure 5.)

Using the frontier structure, we implement PBFS as shown below. The main loop is essentially the same as before, except that it manipulates two frontiers (`prev` and `next`) as opposed to manipulating a bag of vertex identifiers. The sequential processing of a piece of frontier is also the same as before, up to this change to the frontier data structure. The divide-and-conquer function `step_frontier` is used to parallel process the frontier. The end of the recursion is controlled by a `cutoff` parameter, which is expressed in terms of a number of edges.

```
void bfs()
  frontier cur = frontier()
  frontier next = frontier()
  cur.push_edges_of(source_vertex)
  while not cur.empty()
    step_frontier(cur, next)
    cur.swap(next)

void step_frontier_seq(frontier& cur, frontier& next)
    cur.iter(fun (int vertex, int target) →
      if attempt_first_visit_atomic(target)
        next.push_edges_of(target)

void step_frontier(frontier& cur, frontier& next)
  if cur.nb_edges() <= cutoff
    step_frontier_seq(cur, next)
  else
    frontier cur2 = frontier()
    frontier next2 = frontier()
    cur.split(cur2)
    fork2((fun _ → step_frontier(cur,next)),
          (fun _ → step_frontier(cur2,next2)))
    next.merge(next2)
```

Overall the code is basically as simple as that of the previous code that only exploits parallelism at the vertex level, except that the new code is able to exploit parallelism that is available either at the vertex level, or at the edge level, or only across the two levels.

5.3 Lazy splitting based on number of edges

As argued in challenges section, there is no value for the cutoff that will work well for all graphs: either the cutoff is too small, and the algorithm suffers from large overheads; or the cutoff is too large, and the algorithm is not able to exploit all the parallelism available. Lazy splitting enables us to use a small value of the cutoff, enabling parallelization, without inducing large overheads when the graph exposes a lot of parallelism. Our lazy-splitting PBFS only forks tasks when needed, that is, when an idle processor queries a busy processor to obtain work. The function `has_incoming_query` is used to detect queries; a follow-up call to `fork2` leads to the creation of two threads, the second of which is immediately sent by the scheduler in response to the query.

We implement the lazy-splitting scheme by replacing the function `step_frontier` with a new function, `step_frontier_lazy`, whose code appears below. In addition to the `cutoff` parameter, which decides the minimum amount of work that is allowed to be split, the code now also involves a `polling_cutoff` parameter which controls how frequently polling is performed. In practice, processing a few dozen edges is sufficient for amortizing the cost of polling on a memory cell and possibly rejecting a query.

```
void step_frontier_lazy(frontier& cur,frontier& next)
```

```
while not cur.empty()
 if has_incoming_query()
   if cur.nb_edges() <= cutoff
     reply(fun _ → return) // reject the query
   else
     frontier cur2 = frontier()
     frontier next2 = frontier()
     cur.split(cur2)
     fork2((fun _ → step_frontier_lazy(cur,next)),
           (fun _ → step_frontier_lazy(cur2,next2)))
     next.merge(next2)
     return
 cur.iter_pop_nb(polling_cutoff, fun (v,target) →
   if attempt_first_visit(target)
     next.push_edges_of(target))
```

Observe that if a (busy) processor owns more than `cutoff+polling_cutoff` edges, and if it receives a query while processing the first `polling_cutoff` edges, then the processor will share its work. Therefore, it is the case that when (sufficient) parallelism is present in the graph, this parallelism will be exposed by the algorithm after a fairly short period of time.

As an optimization, we can modify the main loop of the BFS algorithm in order to reduce overheads in the case where the frontier at a given distance contains no more than `cutoff` edges. In this case, there is no need to poll for queries, because at most one processor has work to do. Note that, in this case, the processor can use non-atomic operation to mark the vertices. (This later optimization actually also applies to the previous algorithms.)

6. Parallel depth first traversal

6.1 From sequential to parallel DFS

Consider the following sequential algorithm for visiting all the vertices reachable from a source vertex (in an unspecified order). Maintain a *frontier*, storing the set of vertices whose edges need to be processed, and repeat the following action until the frontier becomes empty: pop a vertex from the frontier and visit all its neighbours, pushing previously-unvisited vertices into the frontier.

```
void dfs()
  bag<int> fr = bag()
  fr.push(source_vertex)
  while not fr.empty()
    int vertex = fr.pop()
    foreach target in neighbours[vertex]
      if attempt_first_visit(target)
        fr.push(target)
```

This traversal can be parallelized, by having each processor work on its own frontier, and having work being transfered to processors which have emptied their frontier completely. Eventually, the DFS traversal needs to terminate when the frontiers associated with each of the processors all become empty. Various techniques can be used to detect termination, for example by having one processor being responsible, when it has no work left, to check whether all the other processors are idle.

6.2 Work efficient parallel DFS

As argued for in the challenges section, trying to share the work too eagerly can be counter-productive. At the same time, withholding the work for too long may prevent opportunities to exploit parallelism. Our PDFS is designed in such a way that the communication overheads are always amortized over a sufficiently-large amount of work. This work may either be the work being transfered, or it may be work that was performed locally immediately prior to the transfer.

We show below the code being executed by each of the processors taking part in a run of our PDFS. Until the traversal is complete, as tested by calling the function `traversal_completed`, each processor is either out of work, in which case it calls `acquire` to try and acquire work, or it has an nonempty frontier, in which case it is able to process edges. As a processor visits the endpoint of an edge, it may discover new vertices. For each vertex, the processor pushes the out-going edges into the same frontier data structure from which it is consuming its edges.

A busy processor checks, in between every `polling_cutoff` edges, whether it received an incoming query from an idle processor. If so, it shares half of the edges in its frontier, at the following condition: either the frontier contains more than `cutoff` items, or the processor has locally processed more than `cutoff` edges since the last work transfer. Only when both of those contitions are met is an edge sent. In the code below, the variable `nb` is used to keep track of the number of edges processed since the last transfer.

```
void parallel_dfs_thread()
  frontier fr = frontier()
  int nb = 0
  while not traversal_completed()
    if fr.empty()
      acquire(fr)
      nb = 0
    else
      if has_incoming_query()
        if cur.nb_edges() > cutoff || nb > cutoff
          reply(fun (frontier& other_fr) →
                  fr.split(other_fr))
          nb = 0
        else
          reply(fun _ → return) // reject the query
      nb +=
       fr.iter_pop_nb(polling_cutoff, fun(v,target) →
         if attempt_first_visit(target)
           fr.push(target))
```

Like our PBFS, our PDFS is highly parallel: if a processor has work and receives queries, then it will soon afterwards share the work that it has. Here, however, work efficiency is trickier to achieve: the migration cost cannot always be amortized on the work being sent, sometimes the cost needs to be amortized on work performed locally in between transfers.

7. Experiments

Remark: due to lack of space, we were not able to include details about the implementation of termination detection, and benchmarking results for single-processor runs of parallel programs. We have made the technical appendix that contains this information available to the program chair.

7.1 Implementation of the scheduler

We implemented our algorithms in C++, using a lightweight multithreading library that we have developed for programming parallel algorithms on multicore platforms. At the start time of the program, our implementation creates one POSIX thread (i.e., *pthread*) for each core available. Work items that are created by our graph algorithms are scheduled on and balanced among the pthreads using our implementation of the scheduling primitives described in Section 4.1. We chose against using an off-the-shelf system, such as Cilk Plus [15] or TBB [16], because we could find no obvious way to add support for our scheduling primitives. In applicable cases, such as the Leiserson and Schardl's BFS algorithm, we compared our library implementation against our reference Cilk Plus implementation to confirm that both implementations deliver comparable performance on fork-join programs.

For fork-join algorithms, we follow the standard deque discipline of work stealing: processors push and pop threads from the bottom of their deque, and share the threads from the top of their deque. For our lazy splitting algorithms, load balancing relies on the operations `acquire`, `has_incoming_query` and `reply`, which we introduced in Section 4.1. Internally, these operations use a lightweight protocol based on compare-and-swap (CAS) operations to register queries. Polling on queries involves only one non-atomic read operation, and rejecting one query involves only a non-atomic write operation. Following the work-stealing scheme, idle processors make queries to non-idle targets selected at random.

7.2 Implementation of the algorithms

We ported to our library all the algorithms that we compare against, namely Leiserson and Schardl's PBFS and Cong et al's PDFS. Our porting effort involved rewriting the algorithms to the specifications of the original publications, with the exception of Leiserson and Schardl's bag data structure, which we reused directly. We also considered a port of the non-deterministic version of the PBFS algorithm from PBBS [6]. We do not include the corresponding results because this algorithm performed either similar to or worse than Leiserson and Schardl's code. Note that Cong et al's algorithm required significant care on our part to achieve high performance, as their paper gives fewer details. We focused on three crucial features: batches of vertices are represented by fast fixed-capacity stacks storing 32 vertex ids each (other capacities lead to worse performance); load balancing is implemented by our port of the state-of-the-art concurrent deque structure proposed by Chase and Lev [7]; finally, termination detection is mostly the same as the technique used by our PDFS, except for one subtle complication relating to work stealing with concurrent deques.[2]

Regarding the sequential baselines, we implemented three versions of BFS (using fixed-size FIFO queue, resizeable array, and pair of non-resizeable arrays) and a two versions of DFS (using fixed-size stack and resizeable array). We kept the version that was performing best overall: pair of non-resizable arrays for BFS and fixed-size stack for DFS. However, there was no single algorithm that was delivering the best results on all graphs. In particular, there are graphs for which the single-processor execution of the PBFS algorithms runs faster than the baseline (by up to 35%). In all the graphs that we considered, Leiserson and Schardl's algorithm and ours PBFS were either both faster than the baseline, or both slower than the baseline, usually in comparable proportion.

We compared several possible choices for the cutoff values of each of the algorithms, and selected the cutoffs that were performing best overall. For Leiserson and Schardl's algorithm, we used cutoff 512 (both for vertices and for the edges). Larger cutoffs only improve performance by a few percent, but severely degrade speedups on graphs with limited parallelism. Smaller cutoffs lead to noticeable overheads on all graphs. For our algorithms, which rely on lazy binary splitting, we are able to use a smaller cutoff value, namely 128, as we do not have to pay for thread creation overheads at high load.

7.3 Graphs used in the benchmarks

We considered the following large publicly available graphs that come from data that was sampled from the real world. The Twitter, Friendster, and Livejournal describe social networks [1, 19]. Remark: following Shun and Blelloch [26], we symmetrize and remove duplicates from the Twitter graph. The Wikipedia (as of 6 February 2007) and cage15 graphs are taken from the University of Florida sparse-matrix collection [11].

We considered a set of synthetic graphs that we selected to range from moderately to highly parallelizeable. The square- and cube-grid graphs are directed grids in two- and three-dimensional space in which each vertex has 2 and 3 edges, respectively. The random-arity-x graphs are uniform random graphs, with average arity x on every vertex. The complete tree is a perfect binary tree.

We chose several worst-case graphs to test the robustness of our graph algorithms. The chain graph is a single, long path and the parallel-chain-x graphs are different instantiations of the pattern shown in Figure 3, where x denotes the number of independent paths. For PBFS algorithms, parallel chain graphs stress the ability to exploit limited parallelism. For PDFS, they stress the ability to handle large amounts of sequential dependencies.

The trees-arity-x-y graphs are built upon trees of depth two in which the first and second level have out degree x and y, respectively. These trees are chained in the following sense: one random leaf of one tree becomes the root of the next tree. These graphs test the ability of the algorithms to exploit parallelism in the lists of neighbors of the vertices. The phases-x-arity-y graphs are instances of the structure shown in Figure 2. These graphs generalize the idea of the grids, thus allowing us to have an even smaller number of frontiers (e.g., 50, or 10), and control the arity of the vertices (e.g., 5, or 2). In the graph, phases-10-arity-2-but-one, each of the 10 frontiers contains 3.3 million vertices and each vertex has arity 2, except one particular vertex, which is linked to all the vertices in the next frontier (and thus has arity 3.3 million). The goal of these graphs is to stress the need for splitting the frontier according to the number of edges and not just the number of vertices.

The graphs we consider are laid out in memory in the adjacency-list format suggested by Cormen et al [10]. We arranged that our format use a contiguous layout so that the graph contents can be read quickly from disk using a single disk-read operation. In the execution time that we report, we do not include the time taken for loading the file from disk into the memory. However, we do include the time taken for the (possibly-parallel) initialization of the `visited` array. Note that this initialization time is typically relatively small in front of the traversal time.

The different algorithms that we consider may traverse the vertices in different order, depending in particular on the scheduling decisions. For graphs with a regular shape, such as a grid graph, if the adjacent vertices are laid out contiguously in memory, then the order of visit can have a tremendous impact on the execution time (easily more than a factor 10), due to cache effects. In order to compare the algorithms in a fair way, we need to avoid such massive cache effects. To that end, we shuffled the vertices of all the graphs that we generated, so that they get assigned random labels. This shuffling limits the divergence between the algorithms in terms of the number of cache misses.

7.4 Experimental setup

We compiled all programs with GCC version 4.9.2, using optimizations `-O2 -march=native`. For the measurements, we considered an Ubuntu Linux machine with kernel v3.2.0-58-generic. For scalable heap allocation, we used `tcmalloc` from gperftools version 2.2.1. Our test machine hosts 4 Intel E7-4870 chips running at 2.4GHz and has 1Tb of RAM. Each chip has 10 cores and shares a 30Mb L3 cache. Each of the 40 cores has 256Kb of L2 cache and 32Kb of L1 cache. Each core hosts 2 SMT threads, giving a total of 80 hardware threads but, to avoid complications with hyperthreading, we did not use more than 40 threads.

7.5 Benchmark results

Figure 6 reports benchmark results. For parallel runs, the values are averaged over 30 runs. In a few cases, the noise was as high as 10%, but it was mostly below 5%. Sequential runs showed negligible variance. We first comment on the baseline and the maximum speedups achieved, then focus on the PBFS results, on the PDFS results, and end on the comparison between PBFS and PDFS.

First, observe that the execution time of the sequential programs are not proportional to the number of edges involved in the graph. In particular, graphs involving fewer vertices are usually processed faster. The reason is that when the `visited` array is smaller, accesses into it are more likely to result in cache hits than cache

[2] For details, see the end of Chapter 17.6 of Herlihy and Shavit's book [13].

graph	vertices	edges	seq	LS	ours	runtime	seq	Cong.	our	runtime	our PDFS
			BFS	PBFS	PBFS	diff	DFS	PDFS	PDFS	diff.	vs PBFS
friendster	125m	1806m	74.6s	19.9x	23.4x	-15%	58.4s	25.2x	27.2x	-7%	1.5x
twitter	62m	2405m	76.4s	19.9x	24.1x	-17%	64.5s	25.0x	27.9x	-10%	1.4x
livejournal	4.8m	69m	1.5s	16.4x	18.7x	-12%	1.4s	3.2x	21.7x	-85%	1.2x
wikipedia-2007	3.6m	45m	1.4s	19.7x	22.2x	-11%	0.9s	2.9x	18.2x	-84%	1.3x
cage15	5.2m	99m	1.5s	12.1x	12.4x	-3%	1.6s	3.3x	20.4x	-84%	1.6x
random-arity-3	33m	100m	13.7s	19.6x	27.3x	-28%	13.4s	19.0x	28.3x	-33%	1.1x
random-arity-8	12m	100m	7.5s	23.9x	26.9x	-11%	6.6s	12.0x	26.3x	-55%	1.1x
random-arity-100	1.0m	100m	1.1s	17.3x	19.3x	-10%	1.1s	3.3x	22.5x	-85%	1.2x
squared-grid	50m	100m	27.5s	5.1x	5.8x	-12%	18.2s	25.9x	29.5x	-12%	7.7x
cube	33m	99m	20.3s	20.1x	21.4x	-6%	12.7s	27.1x	28.1x	-4%	2.1x
chain	50m	50m	24.5s	1.0x	1.0x	-0%	22.3s	0.6x	1.0x	-44%	1.1x
parallel-chains-8	50m	50m	23.4s	1.4x	1.4x	-3%	22.6s	2.3x	8.0x	-71%	5.7x
parallel-chains-20	50m	50m	23.3s	1.5x	1.5x	-3%	22.7s	3.9x	19.2x	-80%	13.1x
parallel-chains-100	50m	50m	23.7s	1.4x	1.5x	-12%	22.9s	8.9x	31.0x	-71%	21.0x
parallel-chains-524k	50m	50m	34.1s	36.4x	33.4x	+9%	22.3s	22.7x	35.6x	-36%	1.6x
phases-50-arity-5	40m	197m	29.2s	39.8x	37.9x	+5%	18.9s	28.8x	34.1x	-15%	1.4x
phases-10-arity-2-but-one	33m	93m	16.8s	28.1x	36.3x	-23%	16.5s	6.7x	39.2x	-83%	1.1x
trees-arity-524k	200m	200m	17.4s	13.7x	13.4x	+3%	17.5s	0.7x	18.0x	-96%	1.3x
complete-binary-tree	134m	134m	54.7s	37.8x	37.0x	+2%	41.5s	35.4x	22.2x	+59%	0.8x
trees-arity-10k-10k	100m	100m	7.6s	12.1x	13.8x	-12%	7.9s	11.8x	18.6x	-37%	1.3x
trees-arity-512-512	100m	100m	7.8s	0.9x	10.5x	-91%	8.2s	8.4x	19.1x	-56%	1.7x
trees-arity-512-1024	100m	100m	7.7s	1.3x	11.7x	-89%	8.1s	8.5x	18.9x	-55%	1.5x

Figure 6. Benchmark results. Number of vertices and edges are expressed in millions. Sequential run times are expressed in seconds; smaller is better. PBFS and DFS speedups are relative to the BFS and DFS sequential code, respectively; higher is better. The percentage figures are describing variations in execution time; large negative values are better.

misses. Second, observe that the sequential BFS and sequential DFS baseline are quite close. The few differences (in particular on grid-style graphs) can be explained by different access patterns to the `visited` array, which may affect the number of cache misses.

Now, looking at the speedup results, we observe that, for the vast majority of the graphs, speedups do not exceed 30x. Several factors contribute to these sublinear speedups. First, the parallel algorithms typically perform a little bit more work than the baseline algorithm. For example, PBFS algorithms (both Leiserson and Schardl's and ours), when run with a single processor, can be 20% to 40% slower than the sequential baseline. Second, graph traversals are memory bound, and the memory systems of multicore machines are the limiting factor. Studies by Leiserson and Schardl [20] and Shun and Blelloch [26] report similar speedups on similar machines and, moreover, the Leiserson and Schardl study offers evidence suggesting that the sublinear speedups are limited by hardware.

For graphs with fewer than 5 million vertices, which are typically processed in fewer than 1.5 seconds, speedups appear to be capped at 20x. Essentially, there is not enough work to feed 40 cores. Other graphs, such as the square grid, exhibit relatively poor speedups in PBFS (about 5x), due to the fact that the traversal involves many frontiers that each store a fairly small number of edges (from 2 to 14000).

Comparing the speedups of Leiserson and Schardl's (LS) PBFS with ours (looking at the column which shows the relative change in execution time), we observe that our algorithm is usually faster, and, in the rare cases where it is not, our algorithm is no more than 10% slower. There are also a few graphs where our algorithms perform significantly better than LS. On the graph random-arity-100, LS creates large sequential tasks. For the graph phases-10-arity-2-but-one, we have a frontier that contains many vertices with small out-degree, except for one vertex. On this graph, splitting the frontier according to the number of edges as opposed to the number of vertices leads to significantly better load balance. With the tree-arity-512-512 graph near the bottom of the table, the LS algorithm sequentializes all of the computation. Similarly, with tree-arity-512-1024, in each frontier, the vertices are processed sequentially, and, for each vertex, exactly two tasks are created to process the out-going edges, significanty limiting the speedup (1.3x). In contrast, our algorithm is able to take advantage of the limited amount of available parallelism, achieving speedups exceeding 10x.

Looking now at DFS results, we first observe that, on large real-world graphs such as friendster, Cong et al's algorithm is fairly competitive. On other graphs, however, we are able to significantly outperform Cong et al's algorithm. Our better speedups can be explained (1) by the fact that we are able to exploit parallelism at the edge level, where Cong et al do not, and (2) by the fact that our load balancing operations transfer half of the frontier, and not just a small constant number of vertices. Moreover, there are a few extreme differences, such as with parallel chains, where Cong et al's batching strategy induces significant overheads, In contrast, our algorithm implements techniques for controlling the overheads.

The last column of the table shows the speedup of our PDFS algorithm over our PBFS algorithm. For all graphs but the complete binary tree, PDFS runs faster. For the complete binary tree, synchronizing all the processors at each of the $\log n$ phases actually helps PBFS achieving a close-to-optimal load balancing in this specific situation. At the other end of the spectrum, on the parallel chain graphs, where PBFS shows no speedup because the frontiers are too small, PDFS exhibits near optimal speedups: 8x for 8 parallel chains, and 19.3x for 20 parallel chains. More generally, if we leave out the particular case of the single-chain graph which inherently contains no parallelism, we can expect PDFS to significantly outperform PBFS on all large diameter graphs.

8. Related Work

PBFS Leiserson and Schardl described essentially the algorithm presented in section 5.2, with independent parallelization of the outer loop over the vertices and the inner loop over the outgoing edges [20]. A central aspect of Leiserson and Schardl's contribution is a bag data structure that supports the efficient operations needed for expressing parallel BFS as a divide and conquer algorithm: `push` (to add a vertex to the bag), `split` (to extract approximately half of the vertices from the bag into another bag; at least one third, at most two thirds), `merge` (to transfer all the vertices from a bag into another), and sequential iteration of the items. This bag data structure is implemented using binomial trees (lists of trees whose size grow exponentially), in which each node stores a pointer on a fixed-capacity array of vertex ids. Our frontier data structure, which builds on top of a weighted sequence data structure, improves over Leiserson and Schardl's bag in two ways: first, it supports splitting based on the number of edges as opposed to the number of vertices; and second, it supports exact splitting as opposed to approximate splitting, improving the balancing of the work load and thereby reducing the communication overheads.

The strength of Leiserson and Schardl's PBFS is that it is work efficient: the number of read and write operations that it performs is just a tiny fraction greater than the number of operations performed by the sequential BFS algorithm. Our PBFS improves on theirs by adding the ability to exploit parallelism that spans across the two nested loops (the loop on the vertices and the loop on the edges).

PDFS Cong et al [9] propose an algorithm for implementing PDFS on multicore, using concurrent deques for implementing dynamic load balancing. Cong et al argue that, since pushing vertices one by one into the deques induces too large overhead, vertices should be batched, and batch pointers should be pushed into the concurrent deques. They observe that the size of the batches should be small at the beginning of the traversal, to allow for fast balancing at low load, and that batches may then be of some size S (e.g., 128 vertices) otherwise. More precisely, a processor produces batches of size $\min(S, 2^{|Q|})$, where $|Q|$ denotes the size of the deque of the process. Thanks to this adaptative batching strategy, Cong et al's algorithm is able to expose all the parallelism available. (Recall, e.g., the case of a graph with two parallel chains.) Our PDFS takes a different approach to taming overheads. Instead of using batches, our PDFS relies on an efficient frontier data structure that can expose all instantaneous parallelism on demand. Instead of using concurrent deques, our PDFS relies on a message-passing-like scheme to dynamically balance work among cores. This scheme gives us more flexibility on the choice of the frontier data structure.

9. Conclusion

We have presented two new algorithms for PBFS and PDFS that are based on our novel frontier data structure. We argued that our PBFS and PDFS are efficient in theory and practice, comparing favorably to two state-of-the-art algorithms: Leiserson and Schardl's PBFS and to Cong et al's PDFS. Our evaluation demonstrates several key improvements achieved by our techniques. Our PBFS is more robust, achieving several-fold speedups on particular graphs, where prior work does not yield any speedup. Although it uses slightly more complex data structures, our algorithms are usually not slower than prior work by more than a few percent, and never more than 10% slower on all our benchmark graphs. Our PDFS is typically faster than PBFS, usually by 10% to 30%, as it does not need to synchronize at every layer. There are even particular cases where PDFS achieves over 20x where PBFS achieves no speedup. Our PDFS, on social network graphs, performs just slightly better than Cong et al. On other type of graphs, our algorithms performs a lot better, being often more than twice faster.

References

[1] Stanford large network dataset collection. http://snap.stanford.edu/.

[2] The 9^{th} dimacs implementation challenge, 2013. http://www.dis.uniroma1.it/challenge9/.

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Theory and practice of chunked sequences. In *ESA 2014*, volume 8737 of *LNCS*, pages 25–36. Springer Berlin Heidelberg, 2014.

[4] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *SC '12*, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[5] Rudolf Berrendorf and Mathias Makulla. Level-synchronous parallel breadth-first search algorithms for multicore and multiprocessor systems. In *FC '14*, pages 26–31, 2014.

[6] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP '12*, pages 181–192, New York, NY, USA, 2012. ACM.

[7] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA '05*, pages 21–28, 2005.

[8] Chen-Yong Cher, Antony L Hosking, and TN Vijaykumar. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In *ASPLOS '04*, volume 38, pages 199–210. ACM, 2004.

[9] Guojing Cong, Sreedhar B. Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay A. Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pages 536–545, 2008.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[11] T. A. Davis. University of florida sparse matrix collection, 2010. Available at http://www.cise.ufl.edu/research/sparse/matrices/.

[12] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. Kla: A new algorithmic paradigm for parallel graph computations. In *PACT '14*, pages 27–38, New York, NY, USA, 2014. ACM.

[13] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[14] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. In *PPoPP '09*, pages 55–64. ACM, 2009.

[15] Intel. Cilk Plus. http://www.cilkplus.org/.

[16] Intel. Intel threading building blocks. 2011. https://www.threadingbuildingblocks.org/.

[17] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

[18] Vipin Kumar and V. Nageshwara Rao. Parallel depth first search. part II. analysis. *IJPP*, 16(6):501–519, 1987.

[19] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *WWW '10*, pages 591–600. ACM, 2010.

[20] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA '10*, pages 303–314, 2010.

[21] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *SIGCOMM '07*, pages 29–42. ACM, 2007.

[22] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *LFP '90*, pages 185–197, 1990.

[23] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *CSUR*, 16(3):319–348, September 1984.

[24] V.Nageshwara Rao and Vipin Kumar. Parallel depth first search. part i. implementation. *IJPP*, 16(6):479–499, 1987.

[25] E. Reghbati (Arjomandi) and D. Corneil. Parallel computations in graph theory. *SIAM JoC*, 7(2):230–237, 1978.

[26] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPOPP '13*, pages 135–146, New York, NY, USA, 2013. ACM.

[27] Alexandros Tzannes. *Enhancing Productivity and Performance Portability of General-Purpose Parallel Programming*. PhD thesis, University of Maryland, 2012.

[28] Leslie G. Valiant. A bridging model for parallel computation. *CACM*, 33:103–111, August 1990.

[29] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna PN Puttaswamy, and Ben Y Zhao. User interactions in social networks and their implications. In *EUROSYS '09*, pages 205–218. Acm, 2009.

A. Termination detection

In our parallel BFS, termination detection is implicitly handled by the fork-join construct (`fork2`). When a thread is migrated to another processor, the thread synchronizes upon termination using the atomic counter associated with the join thread which is performing the merge operations. In our PDFS, however, we do not need to perform any merge operation. We can therefore rely on a more efficient scheme, which simply detects when the frontiers of all the processors become empty.

We implement this scheme by using an array of integers, with one cell per processor. Initially, the array is filled with zeros. Then, every time a processor sends a nonempty part of its frontier, it increments its cell; and every time a processor empties its frontier, it decrements its cell. Termination can be detected by observing that the sum of the cells equals zero. A leader processor, chosen arbitrarily, is responsible for checking this sum periodically when it stands out of work. Note that the correctness of this scheme relies on the assumption that store operations are not reorderd. Intel architectures (x86-TSO) guarantee this. For others, such as Power, a lightweight store-store memory fence is needed.

B. Sequential Overheads of Parallel Algorithms

Figure 7 reports on the run time of the single-processor parallel programs compared with that of the corresponding sequential baseline. Note that negative values indicate situations where the parallel algorithm outperforms the baseline, which is possible because there is no sequential program that dominates all the others on all input graphs.

C. Cutoff selection for parallel BFS

We investigated the choice of cutoff values on many graphs. Here, we only report data for one particular graph, Friendster, to illustrate our experimental protocol. As Figure 8 shows, optimal cutoff values for LS PBFS algorithm lie between 512 and 2014. As we seen in general, performance quickly drop both with smaller values (due to high overheads) and with larger values (due to lack of parallelism). Since we want to maximize parallelism, we choose the smallest cutoff value that achieves limited overheads, and therefore select 512. For our PBFS algorithm, which relies on lazy splitting, the choice of the cutoff has no impact on graph with sufficient parallelism, such as the one considered. Note that, of course, the choice of the cutoff for our PBFS algorithm has an impact on graphs with limited parallelism.

D. Further Optimizations

In all algorithms, including the sequential ones, we implemented the following optimization: we do not push into the frontier vertices that have no out-going edges. This optimization significantly improves the execution time for graphs with many leaves, in particular graphs which contain trees. The cost of reading the degree of each vertex when it is first discovered does add some overhead, however this overhead is very small. Indeed, the degree of a vertex needs to be read anyway for later processing the out-going edges of this vertex, and the value is in most cases still in the cache by the time it is read again.

E. Pseudo-code for the Frontier Data Structure

The implementation of the frontier data structure is described by the detailed pseudo-code from Figure 11. This code relies on the implementation of ranges, given in Figure 10, and the interface to the weighted sequence data structure, given in Figure 9. Note that, in the pseudo-code, we treat the adjacency list, called `neighbours`, as a global variable, even though in real code the frontier is actually parameterized by the adjacency list structure.

graph	LS	our	Cong.	our
	PBFS	PBFS	PDFS	PDFS
friendster	+85%	+50%	+43%	+29%
twitter	+95%	+60%	+46%	+28%
livejournal	+41%	+23%	+16%	+22%
wikipedia-2007	+4%	-12%	+20%	+22%
cage15	+73%	+33%	+61%	+20%
random-arity-3	+49%	+26%	+11%	+26%
random-arity-8	+47%	+32%	+29%	+31%
random-arity-100	+33%	+21%	+14%	+12%
squared-grid	-17%	-16%	+32%	+17%
cube	-17%	-27%	+16%	+25%
chain	+5%	+5%	+90%	+2%
parallel-chains-8	-27%	-31%	+33%	+1%
parallel-chains-20	-30%	-34%	+19%	+2%
parallel-chains-100	-24%	-35%	+7%	+2%
parallel-chains-524k	-16%	-4%	+4%	+1%
phases-50-arity-5	-19%	-11%	+3%	+10%
phases-10-arity-2-but-one	+2%	+11%	-7%	-10%
trees-arity-524k	+19%	+49%	+79%	+29%
complete-binary-tree	-17%	-12%	-0%	-10%
trees-arity-10k-10k	+50%	+62%	+88%	+41%
trees-arity-512-512	+16%	+56%	+71%	+38%
trees-arity-512-1024	+20%	+60%	+73%	+39%

Figure 7. Execution time of single-process runs of the parallel algorithms, expressed relatively to their sequential baseline. Smaller values are better.

friendster	40 cores
LS PBFS, vertex-cutoff=2048, edge-cutoff=2048	19.4x
LS PBFS, vertex-cutoff=1024, edge-cutoff=2048	19.7x
LS PBFS, vertex-cutoff=1024, edge-cutoff=1024	19.8x
LS PBFS, vertex-cutoff=512, edge-cutoff=1024	20.0x
LS PBFS, vertex-cutoff=512, edge-cutoff=512	19.9x
LS PBFS, vertex-cutoff=512, edge-cutoff=256	19.3x
LS PBFS, vertex-cutoff=256, edge-cutoff=256	19.4x
LS PBFS, vertex-cutoff=256, edge-cutoff=128	14.1x
our PBFS, cutoff=2048	23.4x
our PBFS, cutoff=1024	23.3x
our PBFS, cutoff=512	23.4x
our PBFS, cutoff=256	23.3x
our PBFS, cutoff=128	23.5x
our PBFS, cutoff=64	23.3x

Figure 8. Effect of the cutoff selection on the execution time on the Friendster graph of PBFS algorithms.

```
class weighted_seq<class A> { // interface
  weighted_seq(weight_type f)
    where weight_type = int f(A x)
  int weight()
  void push(A x)
  A pop()
  void concat(weighted_seq<A>& other)
  void split_at(int w, A& x, weighted_seq<A>& other)
  void iter(body_type body)
}
where body_type = void body(A x)
```

Figure 9. Interface for the weighted sequence data structure, which can be implemented using boostrapped chunked sequences.

```
class range { // implementation
  int vertex
  int low
  int hi

  range()
    vertex = 0; low = 0; hi = 0

  weight()
    return hi-low

  void split_at(int w, range& other)
    other.low = low + w
    other.hi = hi
    other.vertex = vertex
    hi = low + w

  void iter(body_type body)
    for k = low to hi-1
      body(vertex, neighbours[vertex][k])

  int iter_pop_nb(int nb, body_type body)
    if nb == 0 then return 0 // optimization
    nb = min(nb, hi-low)
    int stop = low+nb_real
    for k = low to stop-1
      body(vertex, neighbours[vertex][k])
    low = stop
    return nb
}
```

Figure 10. Implementation of the range of edges data structure.

```
class frontier { // implementation
  weighted_seq<int> vs
  range r1
  range r2

  frontier()
    vs = weighted_seq<int>(degree)
    r1 = range()
    r2 = range()

  void swap(frontier& other)
    // exchange vs, r1 and r2 with other

  int degree(int vertex)
     return neighbours[vertex].size()

  range full_range(int vertex)
     return range(vertex, 0, degree(vertex))

  int nb_edges()
    return vs.weight() + r1.weight() + r2.weight()

  bool empty()
    return nb_edges() == 0

  void push_edges_of(int vertex)
    vs.push(vertex)

  void split(frontier& other)
    int w = (nb_edges()+1) / 2
    if w <= r1.weight()
      r1.split_at(w, other.r1)
    else if w <= r2.weight()
      r2.split_at(w, other.r1)
    else
      w -= r1.weight()
      other.r2 = r2
      int v;
      vs.split_at(w, v, other.vs)
      r2 = full_range(v)
      r2.split_at(w - vs.nb_edges(), other.r1)

  void merge(frontier& other)
    vs.concat(other.vs)

  void iter(body_type body)
    r1.iter(body)
    vs.iter(fun vertex →
      full_range(vertex).iter(body))
    r2.iter(body)

  int iter_pop_nb(int nb, body_type body)
    nb -= r1.iter_pop_nb(nb, body)
    while nb > 0 && not vs.empty()
      int vertex = vs.pop()
      int d = degree(vertex)
      if d <= nb
        full_range(vertex).iter(body)
        nb -= d
      else
        r1 = full_range(vertex)
        nb -= r1.iter_pop_nb(nb, body)
        return nb
    nb -= r2.iter_pop_nb(nb, body)
    return nb;
}
```

Figure 11. Implementation of the frontier data structure.