Fast Parallel Graph-Search with Splittable and Catenable Frontiers

Umut A. Acar, Arthur Charguéraud, Mike Rainey

Fast Parallel Graph-Search with Splittable and Catenable Frontiers

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

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

Mike Rainey
Inria
mike.rainey@inria.fr

Abstract

With the increasing processing power of multicore computers, parallel graph search (or graph traversal) using shared memory machines has become increasingly feasible and important. There has been a lot of progress on parallel breadth first search, but this algorithm can be suboptimal in certain applications, such as in reachability, where the level-ordered visit schedule of breadth first search is unnecessary and even sometimes undesirable. The fundamental problems in developing fast parallel graph-search algorithms can be characterized as: (1) the ability to create parallelism as needed, (2) the ability to load balance effectively, and (3) keep overheads low for not just some but all graphs.

In this paper, we present a new *frontier* data structure for representing the work in graph algorithms that solve the aforementioned three challenges, enabling highly efficient and scalable parallel executions. As the primary application of our frontier data structure, we present a parallel (pseudo) depth-first-search based algorithm, prove work efficiency bound for the algorithm, and evaluate its effectiveness empirically. As a secondary application, we consider parallel breadth first search. Our empirical evaluation shows that our parallel DFS algorithm significantly improves over prior work and almost uniformly outperforms parallel BFS.

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 [18, 20, 27], physical sciences [2], and parallel garbage collection [6, 12, 13, 24]. The diversity of applications makes it crucial to employ different traversal algorithms. For example, while the *Breadth-First Search* (*BFS*) algorithm and its parallel variant *PBFS* appear to be preferable is some applications, in others, such as in parallel garbage collection, depth-first-based traversal algorithms are usually preferred [17].

While there has been much research on parallel breadth-first search [4, 5, 8, 19, 23, 28], other algorithms such as *Parallel Depth-First Search* (*PDFS*) or *Pseudo Parallel Depth First Search* remain much less explored. PDFS is a traversal strategy that approximates depth-first search by allowing individual processors to explore the graph in depth-first order, while permitting different processors to work independently on different portions of the graph. This class of graph-traversal algorithms is important because: (1) depth-first traversal can require less memory in some graphs, (2) depth-first traversal order can exploit the inherent parallelism of a graph better because it does not have to synchronize at each level of the graph as the breadth-first algorithm does, and (3) depth-first traversal order may be desirable in certain situations; for example in parallel garbage collectors depth-first tends to align better with allocation order, leading to improved locality.

Of the much prior work concerning parallel BFS algorithms, Leiserson and Schardl's algorithm [19] stands out because it is *work efficient*—i.e., its total work is competitive with the comparable serial BFS algorithm both in asymptotic complexity and in practical terms (i.e., constant factors). Work efficiency is an important concern, especially in cases where parallelism is severely limited by the structure of the graph, because overheads relating to parallelism are likely to be amplified by certain patterns that are present in the graph, such as long chains of dependencies. To our knowledge, however, there are no theoretical or practical claims about work-efficient PDFS algorithms. Cong et al [9] presented heuristics for improving work-efficiency of PDFS by assigning threads to work on batches of edges. They do not show, however, that the heuristics employed by their algorithm guarantee work-efficiency.

In this paper, we consider several key problems in the design and implementation of work-efficient and highly parallel PDFS and more generally graph-traversal algorithms. Specifically, we make two contributions: (1) a *frontier data structure* for representing the work to be performed by a graph-traversal algorithm while supporting the operations needed for effective parallel executions and (2) a new parallel thread-creation policy that delivers a high degree of parallelism while ensuring that the overheads remain bounded.

One key challenge that our frontier data structure overcomes is that of achieving low overheads on common operations. This feature enables our frontier data structure to remain competitive with highly optimized data structures, such as stacks and queues, that are used in sequential data structures. Furthermore, our frontier data structure supports the iteration, edge-balanced split, and merge operations that are needed for parallel execution. As we describe, the ability to support a rich set of operations at the level of edges enables the client graph-search algorithm to perform three operations crucial to efficient parallelization: (1) effective load balancing (by enabling each processor share half of the work they have), (2) effective granularity control based on an accurate representation of the immediately available work (number of edges), and (3) implement lazy splitting to further reduce the overheads of parallelization.

Based on our frontier data structure, we design and implement a parallel depth-first search algorithm. Like previously proposed approaches to PDFS [9, 21, 22], our algorithm uses individual processors to traverse locally through the graph in depth-first order. Also, like others PDFSs, ours allows multiple processors to deviate

from that order by letting them work on disjoint pieces of the graph in parallel. We prove a work-efficiency bound for our algorithm and show that it performs well in practice by considering both real-world and synthetic graphs designed to test worst-case scenarios.

In principle, our frontier data structure can be used to implement any parallel frontier-based traversal algorithm. To assess the effectiveness of our frontier structure more broadly, we also consider its application to PBFS and compare the resulting algorithm to the work-efficient algorithm of Leiserson and Shardl [19] as well as the direction optimized algorithm of Blelloch and Shun [23].

2. Overview

We present a brief overview of the challenges of and prior work on parallel graph-search algorithms and our approach to overcoming these challenges by using our frontier data structure.

Graph search algorithms, such as breadth-first search (BFS) and depth-first search (DFS), maintain a *frontier* that contains the set of unvisited vertices that are "on the frontier", i.e., connected by an edge to an already visited vertex. Such algorithms proceed by visiting the vertices in the frontier in a specific order and adding new vertices to the frontier in the process of visiting a vertex. Graph-search algorithms typically differ in the order that they choose to visit the vertices in the frontier (and thus the graph). In sequential graph-search algorithms, the frontier data structure is primarily used to insert and delete vertices and can usually be implemented by a simple sequence data structure such as a queue or a stack.

Sharing work. In parallel graph search, however, the frontier data structure needs to support a *split* operation that divides the work into two pieces, one of which can be shared with another processor. For the parallel graph-search algorithm to be *work efficient* that is competitive with the sequential algorithm, it is critical for this data structure to be highly efficient, preferably as efficient as the sequential frontier data structures. This challenge, that is, designing a work efficient frontier data structure that can support split operations, turns out to be highly nontrivial.

Leiserson and Shardl [19] present a *bag* data structure that represents the frontier in PBFS. The bag data structure is processed in a recursive fashion by PBFS. At each intermediate level of recursion, the bag representing the current round of the BFS traversal is divided evenly in two parts and processed in parallel. When the size of the bag falls below a threshold, the bag is processed serially. Because their bag data structure divides parallel work based on the number of vertices, their granularity-control scheme is necessarily imprecise in estimating the amount of immediately accessible work, which is, in fact, proportional to the number of edges.

Cong et al [9] present a *batching* technique for representing the frontier in PDFS. Their batches are small buffers of vertices that enable a processor to share chunks of work. Processors share batches with each other via work-stealing queues. Their batching technique relies on a heuristic that estimates the number of vertices to be placed in a batch. The heuristic is based on the number of vertices in the portion of the frontier operated on by the processor. As with Leiserson and Shardl's bag data structure, Cong et al's frontier data structure is a vertex-based representation and therefore does not guarantee edge-balanced work sharing. In addition, the heuristic cannot guarantee balanced sharing of work based on the number of vertices either.

Our frontier data structure takes into account the number of outgoing edges of each vertex in the frontier, and thus supports *edge-balanced* split operations. To this end, the data structure uses a hierarchical representation that enables both fast insertion and deletion at the level of vertices as well as split operations at the level of vertices. Figure 1 illustrates an example split operation. To enable splitting edges evenly, we use "carry" structures to represent edges split off a single vertex. In the example, each part contains a single non-empty carry consisting of a half of the edges of vertex b. The hierarchical representation is key to the efficiency of our data structure because it allows operating at the level of vertices, which is the common operation. In addition to the ability to split, our frontier data structure supports merging of frontiers. The data structure can thus be used in both depth-first-order and breadth-first-order based traversal algorithms.

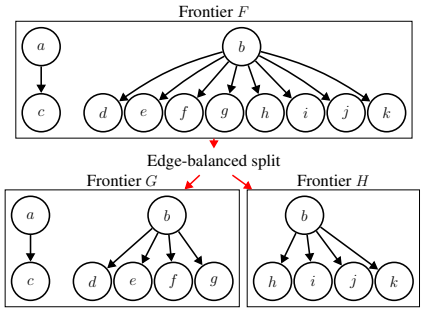

Figure 1. The edge-balanced split operation on the frontier. The frontier F consists of the vertices a, b and implicitly their out-edges. Performing a split operation divides the frontier into two frontiers with equal number of outgoing edges (within a margin of 1), by dividing the edges of the vertices as necessary.

Granularity Control. An important problem in engineering any parallel algorithm is determining the minimum granularity of work that can be performed by a parallel, user-level thread. The common technique to find such a target is to predict the work that will be performed by a piece of computation and assign that computation to a new parallel thread only if the computation is going to perform sufficient work to amortize the cost of creating the thread. For example, we can parallellize a loop by reverting to a serial loop when the number of iterations is smaller than a threshold, sometimes known as the "grain size". An important limitation of this technique stems from the fundamental difficulty in predicting work of a computation: clearly, this technique can withhold creating parallel threads when the amount of work is predicted to be too small, even though the work may turn out to be large. This issue is faced by PDFS and PBFS. Another, perhaps less important, limitation of prediction-based granularity control is that it does not immediately handle nested loops. For example, in PBFS, by treating nested loops separately, prediction-based granularity control can make unbounded amounts of error, thereby over or under sequentializing by potentially large amounts.

In this paper, we introduce a technique for amortizing the cost of thread creation by charging it to the work that was done in the past, rather than the work that is predicted be done by the created thread. This technique enables us to create parallel threads even for work that is predicted to be tiny (but may turn out to be large), allowing us to achieve high parallelism and work efficiency at the same time. We also show that since our frontier data structure allows us to split off work based on the number of edges, it effectively allows us to flatten nested loops in the parallel BFS algorithms into a single loop, offering better control over granularity.

Eager versus lazy splitting. Essentially any parallel algorithm has to make a decision about how to generate parallel work. One approach is to generate work eagerly by splitting large units of work into separate threads even when there is no demand for parallel work. Eager splitting has the advantage of being simple and predictable but it can lead to increased overheads due to creation of parallel threads, which are expensive. An alternative approach is to generate parallel work lazily only when there is a demand for work. In this technique, called lazy binary splitting [25, 26], processors rely on heuristics to determine the global demand for parallelism

by extrapolation on the status of their local work queue and create parallel threads only when the heuristic suggests so. The disadvantage of the approach is that it can be more difficult to apply as it requires techniques for sharing the available work as evenly as possible on demand. Prior work shows this technique can be profitably applied to parallel loops, but can be challenging if the loops may be nested [26], which is the case in algorithms such as BFS.

Since our frontier data structure support edge-balanced partitioning of the work, as we show, it can be used to generate parallel work on demand, as required by lazy splitting. We therefore use lazy splitting in our parallel breadth-first and depth-first-traversal algorithms.

3. Edge-Weighted Frontiers

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

The interface. Our goal is to design a frontier data structure that supports the following operations:

- `push_edges_of`, which pushes all the out-edges of the given vertex into the frontier;
- `iter_pop_nb`, which iterates over `nb` edges (or fewer, depending on the availability), and removes each edge considered from the frontier;
- `split`, which carves out half of the edges into an independent frontier data structure;
- `merge`, which transfers all the vertices from a bag into another.

The `merge` operation is only used by PBFS (not by PDFS), and for PBFS we may assume that merging operations only apply to frontiers that have been constructed by applications of the `push_edges_of` and `merge` functions, but that have not yet been subject to any `iter_pop_nb` or `split` operations.

Our goal is to implement the above operations efficiently, in such a way that `push_edges_of` and iteration operations are nearly as fast as an optimized sequential frontier data structure, and that `split` and `merge` run in logarithmic time. Our solution is based on a recently proposed weighted-sequence data structure, which we describe next, before presenting the implementation of the frontier data structure.

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 B, controls the size of the chunks. For a given B, the concatenation and split operations have a cost bounded by $O(B * \log_{B/2} n)$. This cost is in practice close to $O(\log_2 n)$ operations on binary trees, because $\log_{B/2} n$ is much smaller than $\log_2 n$, and because the constant factor associated with the multiplicative B is tiny (chunk manipulation relies on highly-optimized *memcpy* operations).

```
class range
  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

  int iter_pop_nb(int nb, body_type body)
      where body_type = void body(int src, int dst)
    if nb == 0 then return 0
    nb = min(nb, hi-low)
    int stop = low + nb_real
    for k = low to stop-1
      body(vertex, neighbors[vertex][k])
    low = stop
    return nb
```

Figure 2. Implementation of the range of edges data structure.

In general, the worst-case asymptotic space usage of chunked sequences is $(2 + \frac{O(1)}{B}) * n$. 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 $(1 + \frac{O(1)}{B}) * n$, which, for practical values of B, is close to optimal. In such situations, concatenation and split only cost $O(B * \log_B n)$.

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.

The implementation of our frontier data structure relies on the range data structure, described by the pseudo-code from Figure 2, and relies on the weighted sequence data structure, whose interface was described in the previous section (corresponding pseudo-code may be found in Figure 12 from the appendix). The implementation of the vertex-sequence appears in Figure 3 and is described next. Note that, in the pseudo-code, we treat the adjacency list, called `neighbors`, as a global variable, even though in the real code the definitions are actually parameterized by the adjacency list structure.

The operation `frontier` constructs an empty frontier. The operation `nb_edges` returns the number of edges in the frontier, computed as the total weight of the vertex-sequence plus the sum of the width of the two ranges. The operation `push_edges_of` pushes the vertex given to the vertex-sequence if it has outgoing edges associated with it.

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; it leaves the other half in place. The operation is implemented as follows. 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

```
class frontier
  weighted_seq<int> vs
  range r1
  range r2

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

  int degree(int vertex)
     return neighbors[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)
    if degree(vertex) > 0
      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)

  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 deg = degree(vertex)
      if deg <= nb
        full_range(vertex).iter(body)
        nb -= deg
      else
        r1 = full_range(vertex)
        nb -= r1.iter_pop_nb(nb, body)
        return nb
    nb -= r2.iter_pop_nb(nb, body)
    return nb
```

Figure 3. Implementation of the frontier data structure.

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.

For the `merge` operation, recall that we may assume that the two frontiers passed as arguments have been constructed through calls to `push_edges_of` and `merge`, but have not been subject to `iter_pop_nb` or `split`. Thanks to this assumption, we know that the two frontiers involved in a merge both have their two ranges empty. We may therefore implement `merge` simply by concatenating the vertex-sequences associated with the two frontiers.

The function `iter_pop_nb` iterates over at most `nb` edges, popping them from the frontier as it processes them. It returns the number of edges effectively processed. The edges considered are first picked from the first range, then from the edges associated with the vertices stored in the vertex sequence, and finally from the second range. Note that if a vertex has a large arity, it is possible that only a fraction of its edges are processed; in such case, the remaining edges are placed into the first range, which must be empty in this case. The challenge in implementing this function is that efficiency is critical in the loop over the edges—we are careful to limit the number of operations performed compared with the corresponding loop in the sequential DFS algorithm.

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

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. Graph representation and scheduling interface

Before focusing on our PDFS and PBFS algorithms, we introduce the graph representation and scheduling interface that our algorithms use. We assume the graph to be represented by an adjacency list, and we rely on an array of booleans, which we call `visited`, in order to mark the vertices that have been visited.

```
bag<int> neighbors[nb_vertices]
bool visited[nb_vertices] = { false, false, ... }
```

For load balancing, our algorithms rely on the scheduling interface shown below and explained next.

```
void acquire(frontier& fr)
void has_incoming_query()
void reject_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. The calling processor passes to `acquire` the address of its (empty) frontier structure so that target processors may directly transfer data into the frontier without performing unecessary copy operations. In the load balancing scheme that we consider, the `acquire` function targets a single processor at a time and the call blocks until an answer is received.

The implementation of `acquire` takes care to reject any incoming query on the processor out of work.

Busy processors need to poll for serving queries. More precisely, they are responsible for periodically calling the function `has_incoming_query`, and to reply to potential queries by calling either `reject_query` or `reply`. The `reply` function is presented using a callback argument, which allows the processor to obtain the address of the empty frontier data structure where the processor is going to migrate edges.

In the pseudo-code provided for describing our algorithms, we explicitly insert polling instructions (`has_incoming_query`). However, when executing a split or merge operation on the frontier, we need to ensure that incoming queries are rejected quickly enough. This can be implemented either by inserting polling instructions inside the split and merge functions of the frontier data structure, or, more simply, by executing an atomic compare-and-swap operation on the query cell for temporarily preventing queries to successfully reach the processor.

5. Parallel Depth-First Search

Using our frontier data structure, it is relatively straighforward to implement a version of PDFS where each processor maintains a frontier of edges. In our PDFS, we generate parallelism by splitting available work into two halves when the frontier is larger than a predetermined grain size. To ensure that each vertex is visited no more than once, the algorithm can use atomic read-modify-write operations such as compare-and-swap to lock a vertex before visiting. In this section, we identify an important limitations of this version of PDFS, describe an amortization technique overcoming it, and present an algorithm based on this technique.

Granularity challenge. A central question in PDFS is determining the grain size. In many cases, the difficulty with determining grain size relates to the identification of the smallest constant that amortizes the costs of creating a thread. In the case of PDFS, we show that there is no such constant. To see why, consider the example graph shown in Figure 4. In this graph, two processors can profitably traverse the two long chains in parallel, improving run time by as much as a factor of two over a one-processor execution. Taking advantage of such parallelism can be important in parallel executions. Doing so, however, requires splitting a frontier consisting of just two edges into two by creating parallel threads.

The trouble is that systematically splitting such small frontiers can lead to significant overheads. To see this suppose that we split frontiers of any size (greater than one edge), and consider the example shown in Figure 5. Assume processor 1 begins execution with the source s share the edges leading to b with processor 2, and continues working on vertex a. Since a has no out-edges, processor 1 runs out of work. Luckily, processor 2, which has just received the vertex b, has a frontier that contains two edges (leading to c and d), so it shares the edge leading to d with processor 1. But, then processor 2 runs out of work and we are back to the same condition as in the beginning, but this time with d as the new source. In summary, systematically splitting small frontiers creates one thread for each vertex visited, an unacceptably high overhead. In fact, we have verified empirically that in this and similar examples, parallel runs (that create such small threads) are slower than sequential.

Thus in summary, splitting small frontiers is essential to take advantage of the parallelism in some graphs but in other graphs doing so leads to high overheads. We note that while these examples are small, it is not difficult to generalize them to more interesting examples that exhibit essentially the same problem.

The root of the problem is that the approach to controlling granularity by setting a grain size is just a heuristic: it works when the grain size gives a good measure of the total work to be performed. In PDFS, the size of the frontier is a relatively poor measure of the total work to be performed by a thread: in one case it can lead to a lot of work, in another it can lead to no additional work. The example presented above illustrates exactly this issue.

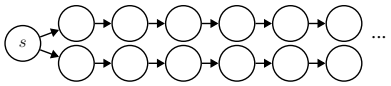

Figure 4. Example for aggressive work sharing.

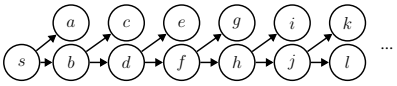

Figure 5. Example against aggressive work sharing.

Parallel DFS algorithm. To solve the granularity problem, we propose a technique based on amortization: each processor shares work only if either (1) the amount of instantaneous work in its frontier exceeds some threshold, or (2) the processor has already performed some predetermined amount of work locally since the last time it shared work. As we describe next, this technique, when implemented with our frontier data structure, enables us to design a fast and robust parallel DFS algorithm.

Figure 6 shows the pseudo-code for the algorithm being executed by each of the processors taking part in a run of our PDFS. Each processor maintains the portion of the frontier that it is working on in a frontier data structure and keeps track of the number of edges it processed since the last load balancing operation in a variable called `nb`.

Until the traversal is complete, as tested by calling the function `traversal_completed`, each processor is busy performing one of three actions: (1) it is working, or (2) it is requesting work from another processor by calling `acquire`, or (3) it is responding to a work request.

To work on its frontier, the processor visits the edges in its frontier and adds the outgoing edges of each visited vertex to its frontier. To test whether a vertex is visited, the processor first executes a conventional read. If the vertex appears to be previsouly-unvisited, it performs an atomic compare-and-swap (CAS) operation to mark the vertex visited.

In order to perform load balancing actions, each processor checks for work requests after visiting `polling_cutoff` edges. If the processor finds an incoming query from an idle processor, it shares half of the edges in its frontier, if either of the following conditions hold: (1) the frontier contains more than `cutoff` edges, or (2) the processor has locally processed more than `cutoff` edges since the last work transfer and there is at least one edge to send. The first condition corresponds to the classical granularity-based approaches to amortizing cost of thread creation by charging to the amount of work that *will be* performed by the DFS algorithm on that frontier. As we described above, however, the first condition alone does not successfully expose the parallelism available in the graph. The second condition solves this problem by amortizing the cost of thread creation to the work that *has already been performed* locally by checking that it has processed at least `cutoff` many edges. This bi-directional (future and past) amortization technique thus allows us to create threads for work that may actually be tiny, and still amortize the cost of thread creation.

The PDFS traversal terminates when the frontiers associated with each of the processors all become empty. Various techniques may be used to detect termination, e.g. by having one processor being responsible, when it has no work left, to check whether all the other processors are idle. We refer to Appendix B for details.

While the technique presented here uses lazy splitting to perform load balancing, it is also possible to create threads eagerly by using essentially the same amortization argument.

```
void parallel_dfs_thread()
  frontier fr = frontier()
  int nb = 0
  while not traversal_completed()
    if fr.empty()
      nb = 0
      acquire(fr)
    else
      if has_incoming_query()
        int sz = cur.nb_edges()
        if sz > cutoff || (nb > cutoff && sz > 1)
          reply(fun (frontier& other_fr) →
                fr.split(other_fr))
          nb = 0
        else
          reject_query()
      nb +=
       fr.iter_pop_nb(polling_cutoff, fun(v,target) →
         if   (not visited[target])
           && cas(&visited[target], false, true)
             fr.push(target))
```

Figure 6. PDFS code executed by each of processor.

We note that the constants `polling_cutoff` and `cutoff` are quite different: the first should be just large enough to amortize the (typically small) cost of polling for queries whereas the latter should be just large enough to amortize the (relatively large) cost of splitting and communicating work.

6. Parallel Breadth-First Search

We describe how to parallelize the sequential BFS algorithm by parallelizing its nested loops independently and discuss the shortcomings of this approach. We then describe how to parallelize BFS using our frontier data structure.

Serial BFS. The core of the sequential BFS algorithm can be expressed simply as a nested loop as shown below.

```
foreach vertex in current frontier
  foreach target in neighbors[vertex]
    if cas(&visited[target], false, true)
      next.push(target)
```

Each execution of this nested-loop statement corresponds to a single phase of BFS, in which vertices at distance d from the source are visited and vertices and distance $d+1$ are added to the next frontier. The first loop traverses over the vertices in the current frontier. The second loop ranges over the outgoing edges of a vertex and visits the target of the edge if it has no already been visited, pushing the target to the next frontier.

Parallelizing nested loops. We can parallelize the serial BFS algorithm by parallelizing each of the loops independently, as for example what is done by Leiserson and Schardl [19]. Using the bag data structure, their algorithm employs a parallel divide-and-conquer strategy to parallelize outer loop. More precisely, the algorithm splits the bag (frontier) recursively by creating parallel threads (via a binary `fork` construct) until the splitting process reaches a bag of less than L vertices (from some constant L). Bags containing L or fewer vertices are processed sequentially. In addition to supporting logarithmic-time split, the bag structure supports logarithmic-time concatenation. When executed, each thread produces another bag of vertices that correspond to a portion of the next frontier. These portions are merged on the way back up the recursion tree, constructing the next frontier along the way. To parallelize the inner loop over the outgoing edges of a vertex, Leiserson and Schardl's PBFS uses a similar divide-and-conquer strategy by splitting ranges of outgoing edges for parallel processing, until these ranges consists of less than K edges, where K is a parameter of the algorithm. Below this threshold the edges are processed sequentially.

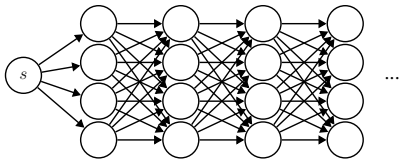

Figure 7. A large graph serialized by parallel BFS.

For this parallelization strategy to be efficient in practice, the constants K and L must be chosen with care: the constants should be just large enough to amortize the cost of thread creation, because a much larger setting can lead to reduced parallelism. As we illustrate, independent parallelization of the loops can lead to serialization of large chunks of work, complicating efforts for controlling granularity. More precisely, this scheme can lead to sequentializing of many as $L * K$ edges. To see this issue, suppose that we chose $L = K = 4$. Consider now the graph shown in Figure 7. In this graph, note that the frontier always contains 4 or fewer vertices and each vertex has 4 or fewer edges. As such, parallelizing nested loops leads to serial processing of the whole graph. Ideally, we would have liked the PBFS to generate 4 parallel threads, each of which consisting of 4 edges of work. We note that in reality the value for L and K are larger (usually in the hundreds); the example generalizes trivially for such values.

Parallelizing BFS with the frontier data structure. Our frontier data structure enables controlling granularity more precisely by expressing the two nested loops as a single loop over the outgoing edges of the frontier. In other words, we can view the frontier as a set of edges rather than set of vertices and create parallel threads by splitting the frontier in half (based on the number of edges), until the frontier has K or fewer edges. (We don't need the second parameter, L.) This approach avoids the aforementioned problem with independently parallelizing the two loops.

Using the parameter K, we can control granularity reasonably effectively. However, it is difficult to do so in a way that performs well for all graphs. The difficulty is that K is too small, then the algorithm typically suffers from large overheads because it creates too many threads; if it is too large, then the algorithm is not able to exploit the parallelism available in graphs where frontiers are small.

To avoid falling in one of these two pitfalls, we can use the lazy splitting technique, which allows us to only fork tasks when needed, that is, when an out-of-work processor queries a busy processor to obtain work [25]. Lazy splitting makes a relatively-small value for K feasible, thanks to the relatively small overhead of polling when compared to eagerly creating threads. For example, in our practical evaluation we have determined that we can set K as small as in low one hundreds and achieve low overheads; without lazy splitting, acceptable values of K are approximately 4-5 times larger.

Figure 8 shows the pseudo-code for our algorithm. For simplicity, we omit the details for storing the distance of the vertices from the source. The `bfs` function describes the main loop of the BFS traversal. It uses a function called `swap` to permute the current and the next frontier structure at the end of each phase; and it relies on the auxiliary function `step` to build the next frontier from the current one. (The two frontier structures are passed by reference to the function `step`.)

The function `step` starts by working as a single thread. Periodically, the `step` function calls `has_incoming_query` to test for incoming queries from other processors. If no query is detected, the `step` function processes a fixed number of edges (as controlled by `polling_cutoff`), before polling again for incoming queries. If, however, the thread has received a query, then it considers the number of edges remaining in the current frontier. If this number

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

void step(frontier& cur, frontier& next)
 while not cur.empty()
  if has_incoming_query()
    if cur.nb_edges() <= cutoff
      reject_query()
    else
      frontier cur2 = frontier()
      frontier next2 = frontier()
      cur.split(cur2)
      fork2((fun _ → step(cur,next)),
            (fun _ → step(cur2,next2)))
      next.merge(next2)
      return
  cur.iter_pop_nb(polling_cutoff, fun (v,target) →
    if   (not visited[target])
      && cas(&visited[target], false, true)
         next.push_edges_of(target))
```

Figure 8. PBFS code based on fork-join and lazy splitting.

is less that the minimum amount of work that is allowed to be split (fewer than `cutoff` edges), then the thread rejects the query. Else, the thread calls `fork2` with two subtasks, corresponding to the processing of the left and the right half of the remaining edges from the frontier held by the current thread, respectively. The right subtask is immediately sent away in response to the query. When both subtasks complete, their output frontier are merged, using the `merge` function on frontiers.

7. Analysis

We prove theorems common to our PDFS and our PBFS algorithms. Our theorems establish that our algorithms are essentially work efficient, and that a process never holds back work that would be worth sharing for more than a short period of time. For simplicity of presentation, we consider an execution model where one unit of time corresponds to the maximal time taken for processing a single edge, and we neglect the cost of polling for incoming queries (this cost is relatively small in practice anyway).

Definition 7.1 *Consider a graph. We let:*

- n *denote the number of vertices;*
- m *denote the number of edges;*
- K *be a shorthand for* `cutoff`*;*
- D *be a shorthand for* `polling_cutoff`*;*
- B *denotes (as before) the size of a chunk in vertex-sequences;*
- C_{fork} *be a bound on the cost of executing a fork-join operation.*

Lemma 7.1 (Maximal number of frontier split operations) *PBFS performs at most $\frac{2m}{K}$ split operations. PDFS performs at most $\frac{3m}{K}$ split operations.*

Theorem 7.1 (Quasi work efficiency) *The total work performed by the processors does not exceed:*

$$O\left((n+m)\cdot\left(1+\frac{C_{\mathsf{fork}}+B\log_B(n)}{K}\right)\right).$$

This result is slightly weaker than work efficiency, because our bound is more than a constant factor larger than the $\Theta(n+m)$ bound achieved by the sequential algorithms. However, for all practical purposes, our algorithms remain work efficient, because the bound is quite loose (it makes worst-case assumptions that correspond to an unlikely sequence of events), and because the base of the logarithmic (non-constant) term, B, a parameter, can be set to a large value. Indeed, B is typically set to 32 (so as to minimize $B\log_B(n)$, while being a power of two that allows for chunks to align on cache lines), and K is typically set to at least 256 (for the grain size to be large enough to amortize thread creation overheads). Under these settings, for any n up to 1000 billion (one tera), the non-constant factor $\frac{B\log_B(n)}{K}$ is less than 1.

Theorem 7.2 (Quasi maximal parallelization) *If a processor receives a query at a moment when its frontier stores more than $K+D$ edges, then the processor responds to the query by sharing its work, within a delay $O(D+B\log_B(n))$.*

Moreover, in PDFS, if a processor receives a query and has processed more than K edges since it last shared (or received) work, then it responds to the query by sharing its work.

The proofs of the results can be found in the appendix. The proofs are straightforward, except for the bound on the number of splits in PDFS. This bound requires the use of a potential function to show that every split is amortized over at least $O(K)$ edges.

8. Experiments

Experimental setup. We implemented our algorithms in C++, using our lightweight multithreading library, called PASL, that we have developed for programming parallel algorithms on multicore platforms. At the start time of the program, PASL creates one POSIX thread (i.e., *pthread*) for each core available. We added support to PASL for the scheduling interface described in Section 4.

We compiled all programs with GCC version 4.9.1, 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 2.4.

Our benchmark machine has four Intel E7-4870 chips and 1Tb of RAM. Each chip has ten cores and shares a 30Mb L3 cache. Each core runs at 2.4Ghz and has 256Kb of L2 cache and 32Kb of L1 cache. Additionally, each core hosts two SMT threads, giving a total of eighty hardware threads. However, to avoid complications with hyperthreading, we did not use more than forty threads.

Our machine has a non-uniform memory architecture (NUMA). Its main memory is distributed across four banks: one per chip. For sequential programs, we allocate all pages to the bank that is closest to the chip that is running the program, as this policy gives the best running time. For parallel programs, we allocate pages across memory banks in a round-robin fashion, thereby balancing memory traffic across chips, as this policy appears to give the best running times in general for parallel runs.

Implementation of the algorithms. We compared our algorithms against three parallel programs:

- Cong et al's algorithm, which we implemented in PASL. This implementation required significant care on our part to achieve high performance, as their paper gives few details and the code was not made publicly available. In particular, we implemented batches of vertices are represented by fast, fixed-capacity stacks, each storing 32 vertex ids (other capacities lead to worse performance), and implemented load balancing by the state-of-the-art concurrent-deque algorithm proposed by Chase and Lev [7].
- Leiserson and Schardl's PBFS algorithm, which we implemented in PASL, following the specifications of the original publications, with the exception of Leiserson and Schardl's bag data structure, which we reused directly from their original code. We did check that the same code implemented in Cilk Plus [16] (instead of PASL) delivers comparable performance.

- Ligra's direction-optimizing parallel BFS algorithm, which was first proposed by Beamer [4]. In this case, we did not port Ligra to PASL, but used instead a binary that we compiled from the publicly-available Ligra sources, using the Cilk Plus scheduler provided with GCC. We use Ligra for an additional point of comparison, even though this algorithm specialized for social network graphs is not work efficient in general.

Regarding the sequential baselines, we implemented two versions of DFS (using fixed-size stack and resizeable array) and three versions of BFS (using fixed-size FIFO queue, resizeable array, and pair of non-resizeable arrays). 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%). For all the graphs that we considered, when executed by a single core, Leiserson and Schardl's algorithm and our PBFS were either both faster than the baseline, or both slower than the baseline, usually in comparable proportion. Details can be found in Appendix C.

For PBFS, we carefully investigated the selection of the cutoff values (thresholds K and L introduced in Section 6). For Leiserson and Schardl's algorithm, there is no single combination of cutoff values that can deliver optimal performance for all graphs. We selected $K = L = 512$, as these choices delivered the best performance on average over all graphs that we considered. We found that larger cutoffs only improved performance by a few percent but severely degraded speedups on graphs with limited parallelism; while smaller cutoffs led to noticeable overheads on all graphs. (See Appendix D for an example report of the effect of varying the cutoff values.) For our algorithms, which all rely on lazy binary splitting, we were able to use a smaller cutoff value, that is, $K = 128$, thanks to the fact that we do not have to pay for thread creation overheads at high load.

Graphs used in the benchmarks. The graphs we consider are laid out in memory in the adjacency-list format suggested by Cormen et al [10]. We considered the following large publicly available graphs that come from data that was sampled from the real world. The twitter, friendster, and livejournal graphs describe social networks [1, 18]. (Following Shun and Blelloch [23], 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 also 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 par-chains-x graphs are different instantiations of the pattern shown in Figure 4, 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-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-d-y graphs are instances of the structure shown in Figure 7. 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-d-2-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 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 traversal 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.

Benchmark results. Figure 9 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 PDFS results, on the PBFS results, and then on the comparison between PDFS and PBFS.

Baseline and speedups. First, observe that the execution times 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 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 most 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, sequential algorithms benefit from the fact that all the data is allocated on the chip of the processor carrying out the work, as opposed to being distributed. Third, graph traversals are memory bound, and the memory systems of multicore machines are the limiting factor. Studies by Leiserson and Schardl [19] and Shun and Blelloch [23] report similar speedups on similar machines and, moreover, offer evidence suggesting that the sublinear speedups are limited by hardware.

Another explaination for sublinear speedups is the lack of work to parallelize. For graphs with fewer than 5 million vertices, which are typically processed in fewer than 1.5 seconds, speedups appear to be capped at 18x. Essentially, there is not enough work to feed 40 cores. Other graphs, such as the square grid, exhibit relatively poor speedups in PBFS (about 4x), due to the fact that the traversal involves many frontiers that each store a fairly small number of edges (from 2 to 14000).

PDFS results. Looking now at PDFS results, we first observe that, on large real-world graphs, such as friendster, and on grid graphs, Cong et al's algorithm is competitive with ours, suggesting that our implementation of Cong's algorithm is fairly optimized. On other graphs, however, our PDFS significantly outperforms 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 bal-

Input graph				DFS				BFS					
graph	verti. (m)	edges (m)	max dist	seq DFS	our PDFS	Cong. PDFS	**Cong. vs ours**	seq BFS	our PBFS	LS PBFS	**LS vs ours**	Ligra vs ours	our PBFS vs PDFS
friendster	125	1806	28	55.8s	26.2x	24.1x	**9%**	67.3s	21.0x	17.9x	**17%**	-0%	50%
twitter	62	2405	14	57.8s	24.8x	22.2x	**12%**	68.6s	21.6x	17.8x	**21%**	-52%	36%
livejournal	4.8	69	14	1.1s	17.4x	2.5x	**592%**	1.1s	13.4x	11.7x	**14%**	57%	26%
wikipedia-2007	3.6	45	459	0.7s	15.0x	2.4x	**532%**	0.7s	12.0x	10.6x	**13%**	70%	31%
cage15	5.2	99	49	1.2s	16.7x	2.6x	**534%**	1.2s	9.8x	9.5x	**3%**	238%	61%
random-arity-3	33	100	27	10.7s	22.9x	15.2x	**51%**	11.0s	22.0x	15.7x	**40%**	102%	8%
random-arity-8	12	100	12	5.0s	21.7x	9.1x	**139%**	6.2s	22.0x	19.5x	**13%**	-12%	22%
random-arity-100	1.0	100	4	0.9s	19.0x	2.7x	**605%**	0.9s	16.4x	14.3x	**15%**	21%	18%
square-grid	50	100	14k	13.3s	22.0x	18.6x	**18%**	21.1s	4.4x	4.0x	**11%**	384%	691%
cube-grid	33	99	960	10.4s	23.1x	22.1x	**5%**	14.9s	15.8x	14.9x	**6%**	292%	111%
chain	50	50	50m	17.4s	0.8x	0.4x	**72%**	19.3s	0.8x	0.8x	**0%**	74%	11%
par-chains-8	50	50	6.3m	17.7s	6.3x	1.9x	**233%**	17.6s	1.1x	1.1x	**5%**	138%	460%
par-chains-20	50	50	2.5m	17.9s	15.2x	3.1x	**393%**	17.5s	1.2x	1.1x	**4%**	24%	1187%
par-chains-100	50	50	500k	17.9s	24.3x	7.1x	**242%**	18.2s	1.0x	1.2x	**-13%**	-9%	2354%
para-chains-524k	50	50	96	17.4s	27.8x	17.7x	**57%**	25.7s	25.2x	28.0x	**-10%**	38%	63%
phases-50-d-5	40	197	50	15.4s	27.8x	23.5x	**18%**	22.1s	28.8x	30.2x	**-5%**	74%	39%
phases-10-d-2-one	33	93	10	13.2s	32.5x	5.5x	**489%**	13.8s	29.7x	23.6x	**26%**	337%	14%
trees-524k	200	200	381	15.4s	17.4x	0.5x	**3165%**	15.6s	12.0x	12.3x	**-3%**	234%	47%
complete-bin-tree	134	134	26	32.9s	17.5x	27.5x	**-37%**	44.3s	30.0x	30.8x	**-2%**	169%	-22%
trees-10k-10k	100	100	2	7.0s	17.9x	10.1x	**76%**	7.1s	12.8x	11.1x	**14%**	65%	41%
trees-512-512	100	100	762	7.2s	16.8x	7.4x	**128%**	7.1s	9.6x	0.8x	**1039%**	843%	73%
trees-512-1024	100	100	380	7.2s	17.4x	7.6x	**128%**	7.1s	10.8x	1.2x	**820%**	1090%	59%

Figure 9. Benchmark results. Parallel runs use 40 cores. Sequential runs benefit from data fully allocated locally. Number of vertices and edges are expressed in millions. Sequential baselines are expressed in seconds; smaller is better. PDFS and PBFS speedups are relative to the DFS and BFS sequential code, respectively; higher is better. The percentage figures describe variations in execution time; higher values indicate larger overheads. For example, in the column labeled "Cong vs ours", the value 534% in row "cage15" indicates that Cong's algorithm is 534% slower than ours on that input graph.

ancing 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.

PBFS results. 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 five cases where it is not, our algorithm is no more than 13% slower. The graphs for which our PBFS is slower are the parallel chains graphs. The relatively slow performance here is owing to the higher cost of pushing and popping in our frontier structure when the size of the frontier is small and parallelism is lacking.

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-d-2, 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-512-512 graph near the bottom of the table, the LS algorithm sequentializes all of the computation. Similarly, with tree-512-1024, in each frontier, the vertices are processed sequentially, and, for each vertex, exactly two tasks are created to process the outgoing edges, significanty limiting the speedup (1.1x). In contrast, our algorithm is able to take advantage of the limited amount of available parallelism, achieving speedups exceeding 10x.

Comparing Ligra's direction-optimizing BFS to ours, we see that in only five cases is Ligra faster. The first three graphs where Ligra is faster are the graphs with low diameter: friendster, twitter, and random-arity-8. This result is to be expected because Ligra's algorithm has been designed specifically for such graphs. Looking deeper, we investigated running times of runs with a single core and reported the results in Appendix C. There results show that the better results of Ligra's algorithm are not explained by a better scalability of this algorithm but instead by the fact that Ligra performs a smaller amount of raw work. For all other graphs, our algorithm is performing the same as or better than Ligra, and in nine cases our algorithm runs more than twice faster.

PDFS vs PBFS. The last column of the table in Figure 9 shows the speedup of our PDFS algorithm over our PBFS algorithm. For all graphs but one, PDFS runs faster. For the complete binary tree graph, 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 at all because the frontiers are too small, PDFS exhibits excellent speedups: 6x for 8 parallel chains, and 15x for 20 parallel chains. More generally, we can expect PDFS to significantly outperform PBFS in general on large diameter graphs.

9. Related Work

We have already discussed in detail the most closely related algorithms, namely the PDFS algorithm of Cong et al and the work-efficient PBFS of Leiserson and Schardl. In what follows, we discuss a number of other related work.

Concurrent steal-half work queues. Hendler and Shavit propose a concurrent data structure that supports constant-time push and pop along with logarithmic-time split [15]. Our work shows that steal half using private work queues is also a viable approach. Moreover, by relying on private rather than concurrent access, we

are free to use a queue structure, such as the one described in prior work [3], that offers low constant factors and asymptotically efficient operations, both in time and space. Moreover, the concurrent steal-half algorithm does not ensure that splits are amortized over sufficient work, and, as such, concurrent steal half faces the granularity-control challenges that were described in Section 2.

Hybrid algorithms. Recent work has shown benefits of using combinations of different traversal strategies. The KLA graph-processing system features a traversal algorithm that switches adaptively between PBFS (level synchronous) and PDFS (asynchronous) traversals to accelerate certain graph algrorithms, such as PageRank and k-core decomposition [14]. Beamer et al [4] and subsequently Shun and Blelloch [23] propose using direction-optimizing BFS for applications, such as graph search, PageRank, connected components, radii estimation, etc. Although faster under certain assumptions, such as small world, direction-optimizing algorithms are sometimes asymptotically slower because they are not work efficient. The reason direction optimizing is not work efficient is that the approach relies on a heuristic to switch adaptively between a work-efficient BFS and non work-efficient one. On each round, the non work-efficient BFS must iterate over all the not-yet-visited vertices to find the ones accessible from the already-marked vertices. By passing worst-case graphs, one can defeat the switching heuristic and thereby trigger running times that are quadratic in the number of vertices.

Parallel garbage collection. In Chapter 14 of their book, Jones et al survey a number of studies of parallel garbage collection [17]. The survey identifies three mark-sweep collectors that use PDFS during the mark phase. To tame overheads, the algorithms proposed by Endo et al [12] and Siebert [24] rely on batching schemes that bear resemblance to the batching scheme proposed by Cong et al. The algorithm proposed by Flood et al [13] uses concurrent per-worker deques. Each of these algorithms relies on sharing work at the level of vertices rather than at the level of edges. In particular, Flood's algorithm relies on sharing vertices one at a time, whereas the others share half of what is locally available at a time. However, unlike our PDFS, the ones that share half do not ensure that splits are amortized over enough work. As such, these algorithms face the granularity-control challenges that were described in Section 2.

10. Conclusion

We have presented a data structure for efficiently representing frontiers in parallel graph search algorithms, and applied this structure to derive a new parallel DFS algorithm and a parallel BFS algorithm similar to that of Leiserson and Schardl's but differs from it in the choice of the frontier data structure used and in the way it controls granularity. In addition to the frontier data structure, we present a simple but effective technique for amortizing the cost of creating parallel threads, which is a critical ingredient of our parallel DFS algorithm. We show that our algorithms are efficient in theory and also in practice by performing a careful experimental evaluation, which shows that our parallel DFS algorithm improves significantly on prior work and our parallel BFS algorithm delivers some (less substantial) improvements. Our evaluation also shows that parallel DFS outperforms parallel BFS, sometimes significantly, in all but one of the graphs considered.

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.

[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, Perry Cheng, and Phillip B Gibbons. Room synchronizations. In *SPAA '01*, pages 122–133. ACM, 2001.

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

[8] Jatin Chhugani, Nadathur Satish, Changkyu Kim, Jason Sewall, and Pradeep Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *IPDPS '12*, pages 378–389. IEEE, 2012.

[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] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *SC '97*, pages 48–48. IEEE, 1997.

[13] Christine H Flood, David Detlefs, Nir Shavit, and Xiolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *JVM '01*, 2001.

[14] 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.

[15] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *PODC '02*, pages 280–289, 2002.

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

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

[18] 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.

[19] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm. SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM.

[20] 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.

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

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

[23] 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.

[24] Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. In *ACM Sigplan Notices*, volume 45.

[25] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10*, pages 179–190, 2010.

[26] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *TOPLAS*, 36(3):10:1–10:51, September 2014.

[27] 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.

[28] Yinglong Xia and Viktor K Prasanna. Topologically adaptive parallel breadth-first search on multicore processors. In *IASTED '09*, volume 668, page 91, 2009.

A. Efficiency proofs

Lemma A.1 (Maximal number of frontier split in PBFS) *PBFS performs at most $\frac{2m}{K}$ frontier split operations.*

Proof In PBFS, we only split frontiers storing more than K edges. Thus, the pieces processed serially have size at least $\frac{K}{2}$. Since there are m edges in total, at most $\frac{m}{K/2}$ pieces may be created. It follows that the maximal number of split operations is $\frac{2m}{K}$. □

Lemma A.2 (Maximal number of frontier split in PDFS) *PDFS performs at most $\frac{3m}{K}$ frontier split operations.*

Proof To bound the number of split operations, we associate a *split-potential* of $\frac{3}{K}$ units to each of the m edges of the graph. As we prove next, using only this potential initially stored in the graph, we are able to consume one unit of potential for every split operation. From this result, we can derive that the total number of split operations does not exceed the total initial potential $\frac{3m}{K}$.

Given a processor having a frontier storing f edges and a local variable `nb`, we define its split-potential $\Phi(f, \texttt{nb})$ as the value $\frac{1}{K}(f + \texttt{nb} + 2 \cdot (f - \frac{K}{2})^+)$ where $(x)^+$ denotes $\max(0, x)$. When a processor adds an edge to its frontier, f increases by one, so the increase in potential is at most $\frac{3}{K}$, which corresponds to the potential brought by the edge. When a processor consumes an edge, `nb` increases by one and f decreases by one, so the potential does not increase. When a processor performs a split and shares work with another processor, we have to show that the operation frees at least one unit of split-potential.

The change in potential of the two processors involved in a split is: $(\Phi(f, \texttt{nb}) + \Phi(0, 0)) - \big(\Phi(\lceil \frac{f}{2} \rceil, 0) - \Phi(\lfloor \frac{f}{2} \rfloor, 0)\big)$, which is equal to $\frac{\texttt{nb}}{K} + \frac{2}{K}\left((f - \frac{K}{2})^+ - (\lceil \frac{f}{2} \rceil - \frac{K}{2})^+ - (\lfloor \frac{f}{2} \rfloor - \frac{K}{2})^+\right)$. The split may be triggered by one of two conditions. The first condition is $f > K$. If it holds, we have $\lfloor \frac{f}{2} \rfloor \geq \frac{K}{2}$ and thus all the parentheses above are nonnegative. The change in potential is thus at least $\frac{2}{K} \cdot (f - \lceil \frac{f}{2} \rceil - \lfloor \frac{f}{2} \rfloor + \frac{K}{2}) = 1$. The second condition that may trigger a split is $\texttt{nb} > K$. Observe that $(f - \frac{K}{2})^+ - (\lceil \frac{f}{2} \rceil - \frac{K}{2})^+ - (\lfloor \frac{f}{2} \rfloor - \frac{K}{2})^+ \geq 0$, Indeed, if $\lfloor \frac{f}{2} \rfloor \leq \frac{K}{2}$ then the last term is zero and the first term exceeds the second term; and otherwise all terms are positive and their sum is equal to $\frac{K}{2}$. It follows that the change in potential xis at least $\frac{\texttt{nb}}{K}$, which is greater than 1 when the second condition (i.e., $\texttt{nb} > K$) is satisfied. Thus, every split operation frees at least one unit of potential. □

Theorem A.1 (Quasi work efficiency) *The total work performed by the processors does not exceed:*

$$O\left((n+m) \cdot \left(1 + \frac{C_{\mathsf{fork}} + B \log_B(n)}{K}\right)\right).$$

Proof Processors work either (1) by manipulating vertices and edges, or (2) by splitting frontiers, forking, and joining and merging in PBFS. For (1), the operations on vertices and edges are amortized constant time, like their counterpart in the sequential algorithms; the total cost is thus $O(n + m)$. For (2), the cost of each fork-join is at most C_{fork}, and the cost of each split-merge is $O(B \log_B(n))$. The number of frontier split operations performed is $O(\frac{m}{K})$, as established by the two above lemmas. The total cost of (2) is thus $O(m \cdot \frac{C_{\mathsf{fork}} + B \log_B(n)}{K})$. Summing on the cost of (1) and (2) concludes the proof. □

Theorem A.2 (Quasi maximal parallelization) *If a processor receives a query at a moment when its frontier stores more than $K + D$ edges, then the processor responds to the query by sharing its work, within a delay $O(D + B \log_B(n))$.*

Moreover, in PDFS, if a processor receives a query and has processed more than K edges since it last shared (or received) work, then it responds to the query by sharing its work.

Proof The PDFS-specific result follows from the fact that each processor uses a variable `nb` to keep track of the number of edges processed since it last performed a split, and that it accepts to share work whenever `nb` exceeds K.

For the first part of the theorem, assume that a processor receives a query at a moment when its frontier stores more than $K + D$ edges. The processor can be in one of several states; for each state, we show that the processor responds by sharing its work within the required delay.

- Assume the processor is running `iter_nb_edges`. This function consumes at most D edges. Subsequently, when the processor polls for the query, it has more than K edges left. It will therefore split its frontier. Since each of the edges—at most D of them—is processed in $O(1)$, and since the split operation takes time $O(B \log_B(n))$, the response to the query is delivered within a delay $O(D + B \log_B(n))$.
- Assume the processor is running `has_incoming_query`. Then it splits its frontier, and responds within delay $O(B \log_B(n))$.
- Assume the processor is running `acquire`. This would mean that the processor owns no edges, contradicting the assumption that the processor owns $K + D$ edges.
- Assume the processor is running a split operation. Splitting can only happen has a result of another query. While splitting, the query that initiated the split has not yet been responded to. So, this query is still outstanding, and therefore the processor cannot receive another query. (Recall from Section 4 that processors may receive at most one query at a time.) A processor running a split operation thus cannot receives a query, contradicting the assumption that it does.
- Assume the processor is running a merge operation (applicable to PBFS only). In the lazy splitting scheme that we use, a processor only starts to run merge operations when it has consumed or given away all of the edges that it initially owned. Therefore, if a processor is running a merge operation, then it must be the case that it has no work left, contradicting the assumption that the processor owns more than $K + D$ edges. □

B. Termination detection

The problem of termination detection is for the most part orthogonal to our discussion. We rely on a simple scheme, described next, to detect when the frontiers of all the processors become empty.

To track the number of nonempty frontiers, we use 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.

C. Sequential Overheads of Parallel Algorithms

Figure 10 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.

D. 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 11 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.

E. Weighted-sequence data structure interface

Figure 12 shows the interface for the weighted-sequence data structure.

graph	LS PBFS	our PBFS	Cong. PDFS	our PDFS
friendster	105%	67%	49%	32%
twitter	117%	79%	62%	43%
livejournal	97%	72%	48%	53%
wikipedia-2007	96%	62%	47%	46%
cage15	120%	69%	99%	45%
random-arity-3	86%	57%	37%	55%
random-arity-8	80%	61%	73%	56%
random-arity-100	60%	45%	36%	32%
square-grid	7%	7%	81%	55%
cube-grid	13%	-1%	43%	51%
chain	33%	33%	144%	31%
par-chains-8	-4%	-8%	69%	28%
par-chains-20	-7%	-11%	50%	30%
par-chains-100	-11%	1%	36%	30%
para-chains-524k	11%	29%	33%	30%
phases-50-d-5	7%	18%	28%	34%
phases-10-d-2-one	24%	36%	17%	5%
trees-524k	33%	67%	126%	23%
complete-bin-tree	3%	8%	27%	14%
trees-10k-10k	61%	76%	134%	31%
trees-512-512	26%	71%	116%	65%
trees-512-1024	30%	74%	116%	66%

Figure 10. 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 11. 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 12. Interface for the weighted-sequence data structure