A Work-Efficient Algorithm for Parallel Unordered Depth-First Search

Umut Acar
Carnegie Mellon University

Arthur Charguéraud
Inria & LRI Université Paris Sud, CNRS

Mike Rainey
Inria

Supercomputing 2015
High-performance graph traversal

• In a *graph traversal*, computation proceeds from one vertex to the next through the edges in the graph.

• Improved performance for graph traversal means improved performance for many other algorithms.

• The main challenge is coping with irregularity in graphs.

• In this work, we present a new algorithm
  • to perform fast traversal over large, in-memory directed graphs
  • using a (single, dedicated) multicore system
  • achieving:
    • analytical bounds showing work-efficiency and high-parallelism, and
    • an implementation that outperforms state-of-the-art codes (almost always)
Motivation

• Most of the recent attention in the research literature on graph traversal is paid to parallel BFS.

• Why parallel BFS but not parallel DFS?
  • Parallel DFS with strict ordering is known to be P-complete (i.e., hard to parallelize).

• However, loosely ordered, parallel DFS:
  • relaxes the strict DFS ordering slightly
  • achieves a high degree of parallelism
  • has many applications, e.g.,
    • reachability analysis & graph search
    • parallel garbage collection (Jones et al 2011), etc…
    • KLA graph-processing framework (Harshvardhan et al 2014)

• When feasible, Pseudo DFS is preferred because it is usually faster than the alternatives.
Pseudo DFS (PDFS)

- **Input:**
  - directed graph and ID of source vertex

- **Output:**
  - the set of vertices connected by a path to the source vertex
PDFS

visited

vertex ids

frontier

migrate

pop↑ push

pop↑ push

pop↑ push

pop↑ push
PDFS vs. PBFS

**Synchronization**

- PDFS is *asynchronous*:
  - Each core traverses independently from its frontier.

- PBFS is *level synchronous*:
  - Cores traverse the graph level by level, in lock step, synchronizing between every two levels.

**Data locality**

- DFS is preferred in parallel GC.
  - e.g., mark sweep

- Why?
  - DFS visits heap objects in the order in which objects were allocated.
The granularity-control challenge

• The key tradeoff is between:
  • the cost to pay for migrating some chunk of work, and
  • the benefit of parallelizing the migrated work

• Migrate too often, it’s too slow; too infrequently, it’s too slow.

• Granularity control is a particular challenge for PDFS because, when you migrate a piece of frontier, you have little information about how much work you’re giving away.
Example in favor of aggressively sharing work
Example against sharing work
Granularity control by batching vertices

• A batch is a small, fixed-capacity buffer that stores part of the frontier.

• In batching, each work-stealing queue stores pointers to batches of vertices.

• Idea: use batches to amortize the cost of migrating work.

• Previous state of the art for PDFS:
  • Batching PDFS (Cong et al 2008)
  • Parallel mark-sweep GC (Endo 1997 and Seibert 2010)

• No batching PDFS so far guarantees against worst-case behavior.
Our work

**Central question:**
Can we bring to PDFS the analytical and empirical rigor that has been applied to PBFS, but keep the benefits of a DFS-like traversal?

- We present a new PDFS algorithm.
- In a realistic cost model:
  - We show that our PDFS is *work efficient*:
    - Running time on a single core is the same as that of serial DFS, up to constant factors.
  - We show that our PDFS is highly parallel.
- In experiments on a machine with 40 cores, we show the following:
  - Our PDFS outperforms alternative algorithms across many of a varied set of input graphs.
  - Our PDFS can exploit data locality like sequential DFS.
Our solution to granularity control

- Migration of work is realized by message passing.
  - Each core regularly polls the status of a cell (in RAM).
  - When core $C_1$ requests work from $C_2$, $C_1$ writes its ID into the cell owned by $C_2$.
  - Each core owns a private frontier.

- Our granularity control technique: when receiving a query, a core shares its frontier only if one of the following two conditions is met:
  - The frontier is larger than some fixed constant, $K$.
  - The core has treated at least $K$ edges already.

- The setting for $K$ can be picked once based (solely) on the characteristics of the machine.
Why is our granularity-control technique effective?
Our PDFS algorithm

Tuning parameters:
- $K$: positive integer controlling the eagerness of work sharing
- $D$: positive integer controlling the frequency of polling

Each core does:
- if my frontier is empty
  - repeatedly query random cores until finding work
- else
  - handle an incoming request for work
  - process up to $D$ edges:
    - for each edge ending at vertex $v$
      - if this core wins the race to claim $v$, push outgoing neighbors of $v$ into the frontier
      - remove $v$ from the frontier

To handle a work request, a core does:
- if frontier contains at least $K$ edges or has at least two edges and has treated at least $K$ edges since previously sending work:
  - transfer half of the local frontier to the frontier of the hungry core
  - notify the hungry core
Analytical bounds

**Theorem 1**
The number of migrations is $3m/K$.

**Theorem 2**
The total amount of work performed is linear in the size of the input graph.

**Theorem 3**
Each work query is matched by a response in $O(D + \log n)$ time.

Shows that each work migration is amortized over at least $K/3$ edges.

Shows that all polling and communication costs are well amortized.

Shows that the algorithm can achieve almost every opportunity for parallelism.
Our frontier data structure

- It is based on our previous work on a chunked-tree data structure.
- It's a sequence data structure storing weighted items.
- It can
  - push/pop in constant time
  - split in half according to the weights of the items in logarithmic time.
- In the PDFS frontier, a weight represents the outdegree of a vertex.
- It enables:
  - rapidly migrating large chunks of frontier on the fly
  - efficiently parallelizing high-outdegree vertices
Experimental results

higher = better

- 40 Xeon cores
  @ 2.4Ghz
- 1 TB RAM

- 40 Xeon cores
  @ 2.4Ghz
- 1 TB RAM
Related work

- PDFS
  - Batching PDFS (Cong et al 2008)
  - Parallel mark-sweep GC (Endo 1997 and Seibert 2010)
- PBFS
  - Work-efficient Parallel BFS (Leiserson & Schardl 2010)
  - Direction-optimizing BFS (Beamer et al 2012)
  - Ligra (Shun & Blelloch 2013)
- Hybrid PDFS/PBFS
  - KLA graph-processing framework (Harshvardhan et al 2014)
Summary

• We presented a new PDFS algorithm.

• Our results lift PDFS to a level of rigor similar to that of work-efficient PBFS.

• In our paper:
  • We show that PDFS exploits data locality as effectively as serial DFS.

• Our results show that PDFS performs well both in theory and practice.

• The results suggest that our PDFS may be useful as a component of other algorithms and graph-processing systems.