POSTER: T-thinker: A Task-Centric Distributed Framework for Compute-Intensive Divide-and-Conquer Algorithms

Da Yan
University of Alabama at Birmingham
yanda@uab.edu

Guimu Guo
University of Alabama at Birmingham
guimuguo@uab.edu

Md Mashiur Rahman Chowdhury
University of Alabama at Birmingham
mashiur@uab.edu

M. Tamer Özsu
University of Waterloo
tamer.ozsu@uwaterloo.ca

John C.S. Lui
The Chinese University of Hong Kong
cslui@cse.cuhk.edu.hk

Weida Tan
University of Alabama at Birmingham
weidatan@uab.edu

Abstract

Many computationally expensive problems are solved by a divide-and-conquer algorithm: a problem over a big dataset can be recursively divided into independent tasks over smaller subsets of the dataset. We present a distributed general-purpose framework called T-thinker which effectively utilizes the CPU cores in a cluster by properly decomposing an expensive problem into smaller independent tasks for parallel computation. T-thinker well overlaps CPU processing with network communication, and its superior performance is verified over a re-engineered graph mining system G-thinker available at http://cs.uab.edu/yanda/gthinker/.

CCS Concepts • **Theory of computation → Parallel computing models**; **Distributed computing models**.

1 Problem Definition

Many computationally expensive problems can be solved by divide and conquer: the computation over a big dataset can be recursively divided into ***independent*** tasks over smaller subsets of the dataset, exposing great parallelism opportunities. To illustrate, we provide 3 examples described as follows.

Application 1: Mining Subgraphs. We consider the problem of mining those subgraphs in a big input graph $G = (V, E)$ that satisfy certain conditions, such as maximum clique finding, quasi-clique enumeration and triangle counting. The search space is the power set of V: for each vertex subset $S \subseteq V$, we check whether the subgraph of G induced by S satisfies the conditions. This giant search space can be organized into a set-enumeration tree [3] as shown in Figure 1. A graph G with four vertices $\{a, b, c, d\}$ is considered where $a < b < c < d$ (ordered by ID). Each node in the tree represents a vertex set S, and only vertices larger than the last (and also largest) vertex in S are used to extend S. For example, node $\{a, c\}$ can be extended with d but not b as $b < c$. Edges can be used for the early pruning of a tree branch: for example, to find cliques (i.e., complete subgraphs), one only needs to extend a vertex set S with those vertices in $(V - S)$ that are common neighbors of every vertex of S.

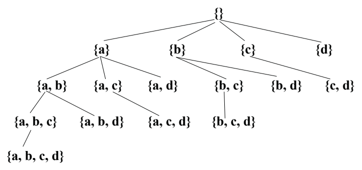

Figure 1. Set-Enumeration Tree

Application 2: Decision Tree. Figure 2 shows the divide-and-conquer training of a decision tree. The root node is associated with all input data instances, and for each attribute x_i and for each splitting value v of that attribute, we compute the decrement of a impurity function when splitting the node into two child nodes based on the condition $x_i < v$, and split based on the condition that maximizes that metric. Each child node is associated the part of data instances that satisfies the condition along its branch, and it can be recursively split into two nodes, making the tree training process a top-down divide-and-conquer algorithm.

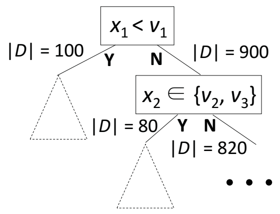

Figure 2. Training a Tree

Application 3: Frequent Pattern Mining. We consider the pattern-growth approach: we check whether a pattern is frequent, and if so, we grow the pattern for further examination. Figure 3 illustrates the PrefixSpan algorithm for mining frequent sequential patterns, where the sequence database D in Figure 3(a) is projected by prefix pattern A (i.e., Figure 3(b)), and then AB (i.e., (c)), and finally ABC (i.e., (d)). We see that the projected database $D|_P$ is shrinking in size.

PPoPP '19, February 16–20, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6225-2/19/02.
https://doi.org/10.1145/3293883.3295709

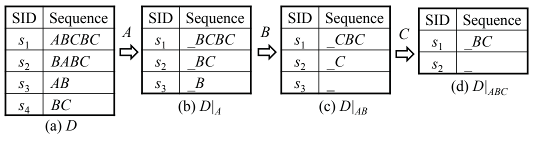

Figure 3. Frequent Sequential Pattern Mining

Without loss of generality, we illustrate the use of T-thinker by considering the application of mining subgraphs in the rest of this paper. Earlier work attempts to tackle this problem using MapReduce but is found to be 10 times slower than a single-threaded program [2] due to a communication-bound execution pattern that underutilizes CPU cores. Other attempts face a similar problem [4, 5] which motivates the development of task-centric graph mining systems like G-thinker [6] and G-Miner [1]. However, the latter two systems still suffer from design problems such as (1) expensive initial graph partitioning and task generation, (2) threads contend on a single data cache for one-at-a-time access, (3) a disk-based task queue that is expensive to insert new tasks.

2 The T-thinker Framework

T-thinker makes it convenient for users to write divide-and-conquer programs for execution with a high parallelism.

T-thinker considers two kinds of objects: (1) tasks and (2) data objects. An underlying distributed data store is created for tasks to request their needed data objects. Each machine also maintains a cache T_{cache} to maintain remote data objects for use by tasks, which allows different tasks on a machine to share data and reduce redundant data requests.

Data objects in T_{cache} are organized as buckets hashed by object IDs, so that two concurrent tasks can access data objects in T_{cache} together as long as they are not in the same bucket. In each bucket, we track whether a data object o is cached, or has been requested. This is because, if a task already requested o, then even if o's content has not been received yet, another task on the same machine should not send a redundant request for o. To keep T_{cache} bounded, data objects no longer needed by any task are tracked for eviction.

A task may be spawned from input data, or recursively generate more new (but smaller) tasks by divide and conquer. Upon their generation, tasks are independent of each other and there is no need of complicated scheduling (e.g., by a dependency DAG). T-thinker adopts a lightweight scheduling strategy. It generates tasks in a proper pace to keep memory consumption bounded (as each task maintains its associated data objects and states), i.e., more tasks are created for processing only if enough memory space is released by finished tasks. Tasks are maintained by an in-memory concurrent queue Q_{task} for fetching by computing threads, and since an active task may generate many new tasks, we spill tasks to local disk in batches if Q_{task} overflows. These tasks are later loaded back when memory permits, prioritized over spawning new tasks to keep the pool of active tasks minimal.

Table 1. System Comparison: Maximum Clique Finding

Dataset	$\|V\|$	$\|E\|$	Arabesque	G-Miner	T-thinker
Orkut	3.1 M	117.2 M	2007 s / 44.7 GB	691 s / 2.5 GB	95.9 s / 1.3 GB
BTC	164.7 M	361.4 M	x	> 24 hr / 7.3 GB	1831 s / 3 GB
Friendster	65.6 M	1806.1 M	x	10644 s / 7.4 GB	252 s / 3.4 GB

Note: (1) M = 1,000,000; (2) Cell format: running time / memory cost; (3) x = Out of memory.

If a task is waiting for data objects, it is suspended to release CPU core for use by other tasks, and it will be timely resumed when the requested data are all received. This allows communication cost to be hidden by computation (which is the performance bottleneck). Load balancing among machines is handled by allowing an about-to-be-idle machine to steal (i.e., prefetch) tasks from other machines for processing.

Programming Interface. T-thinker provides base classes such as *Task* and *Object*, and users define their contents using C++ template arguments and implement abstract functions in their subclasses to specify the task computing logic including (1) how to spawn tasks from input data, and (2) how to continue computing a task given previously requested data. A task may call a function *pull*(o) to request a data object, and may call a function *add_task*(t) to add a new task t. If a task involves too many objects to be collected at a machine, it may request aggregated data (e.g., pattern frequency) instead which is computed at each machine over its local dataset.

3 Performance Evaluation

To verify T-thinker's efficiency, we reengineered G-thinker [6] on top of it, and compared it with the state-of-the-arts such as G-Miner [1] and Arabesque [5]. The experiments were conducted on a cluster of 16 virtual machines (model D16S_V3) on Microsoft Azure, and the code is released at http://cs.uab.edu/yanda/gthinker/. As an illustration, Table 1 shows the performance of the systems for the application of maximum clique finding over 3 big graphs, where we can see that T-thinker is a clear winner thanks to its efficient design.

Acknowledgments

This work was funded in part by NSF OAC-1755464 and Microsoft Azure Research Award.

References

[1] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *EuroSys*. 32:1–32:12.

[2] Shumo Chu and James Cheng. 2012. Triangle listing in massive networks. *TKDD* 6, 4 (2012), 17:1–17:32.

[3] Guimei Liu and Limsoon Wong. 2008. Effective Pruning Techniques for Mining Quasi-Cliques. In *PKDD*. 33–49.

[4] Abdul Quamar, Amol Deshpande, and Jimmy J. Lin. 2016. NScale: neighborhood-centric large-scale graph analytics in the cloud. *VLDB J.* 25, 2 (2016), 125–150.

[5] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *SOSP*. 425–440.

[6] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C. S. Lui. 2017. G-thinker: Big Graph Mining Made Easier and Faster. *CoRR* abs/1709.03110 (2017). http://arxiv.org/abs/1709.03110