A Graph Processing System with Actors

Abstract

Graph-based applications become hotter and hotter due to the rising of the social-networks and other problems come up such as paths of disease outbreaks, or chemical compounds, or biological structures. A strong desire to process large graph motivates researchers to study on distribute memory machines. Unfortunately, developing distributed graph algorithm still requires some cost, especially to non-expert. While manufacturing technology improves, physical limits of semiconductor-based microelectronics have become a major design concern. A combination of increased available space and the demand for increased thread level parallelism led to the development of multi-core CPUs. Today, a single multi-core server already has a very powerful computing capabilities, which means we can exploit its capabilities to do more job.

In this paper, we present GPA, a graph processing system with actors on a single machine. With actors, we avoid the consumption caused by the frequent switching among threads. Additionally, we utilize the memory mapping for better performance. We show, through experiments and theoretical analysis, that processing large-scale graph on a modern PC with actors performs well.

Introduction

Graph algorithms are becoming increasingly important for solving many problems in scientiﬁc computing, data mining and other domains such as social networks, web graphs, chemical compounds, and biological structures. The scale of real graphs is so large that may consist of billions of vertices, trillions of edges. For Example, the Yahoo Web graph have 1.4 billion vertices and 6.6 billion edges. However, Graph processing is difficult because of the inherent complicated data structure of the graph and the extremely large size of the graph. Therefore, designing a scalable, fault-tolerant, robust system for processing the large-scale graphs is one of the most urgent problems facing systems researchers.

Motivated by the demands, there are already some solutions, which are able to process large scale graphs with distributed system such as Pregle, PowerGraph and GPS, proposed by other researchers. Nowadays, though distributed computional resources are more accessible than ever before, processing these graphs still remains many challenges. In distributed system, the first main problem is workload balancing which caused by partitioning the large scale graph into small partition to fit the cluster nodes. The second main issue is message passing. Messages exit among the different computional nodes or inside of the cluster node, the cluster nodes take cost to communicate with each other and the communication between nodes causes latency which matters in a BSP based graph processing system. Therefore, many researchers spend a lot of energy to study the distributed system based on Pregel to solve the problems mentioned above and some gain reasonable performance, such as Mizan which aims to the workload balancing, GPS focus on the messaging latency. But from a developer’s perspective, developing, debugging and optimizing distributed algorithm on distributed system is quite difficult because the user needs to be skilled at managing and tuning a distributed system in a cluster, which is a nontrivial job for the ordinary user. Besides, these distributed systems need many machines in a cluster which brings both money and energy cost.

Recently, some graph processing engines on single PC have been proposed to address the problems of the distributed graph systems. For example, Ligra is a lightweight graph processing framework this is specific for shared-memory multicore machine. Specially, Graphchi, a disk-based graph processing engine on a single modern PC, significantly outperforms all representative distributed graph engines. TurboGraph inspired by Graphchi focuses on parallelism and overlapping of CPU processing and I/O processing with a novel concept pin-and-slide.

However, we observe that none of the existed single PC engine underutilize the resource of the multicore machine or the operating system. Parallelism could improve the computing efficiency. Given the success of parallel computing in scientific computing, data analysis and other areas, parallel processing appears to be necessary to overcome the resource limitations of single processors in graph computations. While the inherent complicated characteristics of the graphs make them hard to match the current parallelism computational problem-solving approaches. Vertex-centric is a very brilliant idea for graph processing. In the mode, every single vertex is a little compute unit, which simplifies the process, and every vertex communicate with each other by message. Unfortunately, a modern PC could not support a large number of concurrent, because the concurrent implemented by thread has its limitation. Motivated by this issue, we present a totally different graph processing engine on a single modern PC. We implements the engine with actor/coroutine, actor/coroutine is an ultra-lightweight thread that could greatly improve concurrency. The higher the concurrency, the more suitable the vertex-centric is. We are bold attempt to map a vertex to an ultra-lightweight thread and transfer the communication between vertices into the communication of ultra-lightweight thread. Besides, we notice that the IO processing is the main efficiency killer, we exploit the ability of memory mapping provided by the operating system.

The rest of the paper is as follows. Section 2 reviews related work. In next section, we adopt the vertex-centric model to fit the ultra-lightweight thread and show the difference. Section 4 presents the detailed implementation of the engine. Section 5 describes the experiments results. And in section 5, and section 6 summarizes and concludes the paper.

Related Work

Many scalable graph systems have recently been proposed. We survey some of the most relevant works, which may be broadly classified into single-machine approaches and distributed approaches depending on the size of the system.

**Distributed Systems:** Pregel is a synchronous vertex-centric programming model proposed by Google for graph system. In Pregel, each vertex is executed in parallel and the user rewrite the compute function invoked by the vertex. Pregel introduced the first bulk synchronous parallel (BSP) distributed message-passing system. In BSP model, all vertex kernels run simultaneously in a sequence of super-steps. Within a super-step, each vertex receives messages from in-neighbor of the last super-step and sends messages to out-neighbor of the next super-step. And a barrier is imposed between two super-steps to guarantee that all vertices finish processing messages. However, Pregel is not open-source and many other systems drown from Pregel such as GPS, PowerGraph, GraphLab, Mizan etc. Other systems Pegasus and gbase are based on MapReduce and support matrix-vector multiplication using compressed matrices.

**Single-machine Systems:** X-Stream is a system for processing both in-memory and out-of-core graphs on a single shared-memory machine. X-Stream take advantage of using an edge-centric model and streaming completely unordered edge lists rather than performing random access. Ligra is a lightweight graph processing framework this is specific for shared-memory multicore machine. Graphchi is a disked-based single-machine system following the asynchronous vertex-centric programming model. Graphchi proposed parallel sliding windows (PSW) to handling disk-based large-scale graphs and it updates values to the edges in Graphchi. PSW partitions the vertices into P execution intervals, and each execution interval contains a shard file that stores all in-edges sorted by their source vertices. PSW processes one shard at a time. During the processing, first of all, loading an execution interval into memory from the disk, then update the vertices and edges. At last, writing the updated content to disk.

Overview

In this section, we now briefly introduce our computation model and an overview of the framework.

Actor of Kilim

While manufacturing technology improves, reducing the size of individual gates, physical limits of semiconductor-based microelectronics have become a major design concern. A combination of increased available space (due to refined manufacturing processes) and the demand for increased thread level parallelism (TLP) led to the development of multi-core CPUs. Since the multi-core has already become the main architecture of the modern PC, in theory, the modern PC should have a powerful computional ability. However, the concurrent computing is still not mature which causes underutilization of the multi-core machines. There are main two ways to implement concurrent computing, multi-process and multi-thread. However, in a single machine, the number of process or thread has its limitation which means poor concurrency and concurrent with thread invoke locks and synchronous. In fact, there is another concept called actor. The actor model is a different way of modeling concurrent processes. Rather than threads interacting via shared memory with locks, the actor model leverages "actors" that pass asynchronous messages using mailboxes. A mailbox, in this case, is just like one in real life — messages can be stored and retrieved for processing by other actors. Rather than sharing variables in memory, the mailbox effectively separates distinct processes from each other. Actors act as separate and distinct entities that don't share memory for communication. In fact, actors can only communicate via mailboxes. There are no locks and synchronized blocks in the actor model, so the issues that arise from them — like deadlocks and the nefarious lost-update problem — aren't a problem. What's more, actors are intended to work concurrently and not in some sequenced manner. As such, actors are much safer (locks and synchronization aren't necessary) and the actor model itself handles coordination issues. In essence, the actor model makes concurrent programming easier.

The basic programming model of languages, like Java, is thread based and while multithreaded applications aren’t terribly hard to write, there are challenges to writing them correctly. What’s difficult about concurrent programming is thinking in terms of concurrency with threads. Alternate concurrency models have arisen along these lines — one that is particularly interesting, and gaining mindshare in the Java community, is the actor model. The actor model facilitates concurrent programming by allowing a safer mechanism for message-passing between processes (or actors). Implementations of this model vary between languages and frameworks. Luckily, there are a number of choices for leveraging this model on the Java platform. Kilim is a library written in Java that embodies the actor model. In Kilim, "actors" are represented by Kilim's Task type. Tasks are lightweight threads and they communicate with other Tasks via Kilim's Mailbox type. Mailboxes can accept "messages" of any type. Tasks can send String messages or even custom message types — it's entirely up to you. Everything is tied together in Kilim via method signatures; if you need to do something concurrently, you specify the behavior in a method by augmenting its signature to throw Pausable. Thus, creating concurrent classes in Kilim is as easy as implementing Runnable or extending Thread in Java. Lastly, Kilim's magic is enabled by a post process, called a weaver, which alters the bytecode of classes. Methods containing the Pausable throws clause are processed at runtime by a scheduler, which is part of the Kilim library. The scheduler manipulates a limited number of Kernel threads. It is able to leverage this pool for a higher number of lightweight threads, which can context-switch and start up quite fast. Each thread's stack is automatically managed. The actor model (and thus Kilim) makes it easier and safer to write asynchronous-acting objects that depend on similar objects.

Memory Mapping

The primary benefit of memory mapping a file is increasing I/O performance, especially when used on large files. Memory-mapping is a mechanism that maps a portion of a file, or an entire file, on disk to a range of addresses within an application's address space. The application can then access files on disk in the same way it accesses dynamic memory. Accessing memory mapped files is faster than using direct read and write operations for two reasons. Firstly, a system call is orders of magnitude slower than a simple change to a program's local memory. Secondly, in most operating systems the memory region mapped actually is the kernel's page cache (file cache), meaning that no copies need to be created in user space.

The principal benefits of memory mapping are efficiency, faster file access, the ability to share memory between applications, and more efficient coding. Another benefits of memory mapping is we can access the data like an array. Mapping a file into memory allows access to data in the file as if that data had been read into an array in the application's address space.

For 64-bit operating system,

Model of Computation with Actor/Coroutine/Kilim

The vertex-centric programming model introduced by Pregel is based on the Bulk Synchronous Parallel (BSP) computation model. As mentioned above, BSP consists of a sequence of super-steps and a barrier is imposed between two super-steps. Within super-step, all vertices kernels run simultaneously. Receiving messages from the last super-step, invoking the user defined computing method, updating the value of the vertices and sending messages to the next super-step. In fact, every vertex kernel cannot execute simultaneously because of the size of memory and the number of cores of the CPU. Here we present our simplified computation model with actors that drawn from Vertex-centric model: vertices are the data carriers and the actors are the computational unit. All the message communication between vertices implements by actors. What we could do is improve the degree of the concurrency with actors and appoint the vertices to the actors.