J. Parallel Distrib. Comput. 74 (2014) 2687–2698

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Streaming data analytics via message passing with application to graph algorithms

Steven J. Plimpton *, Tim Shead

Sandia National Laboratories, Albuquerque, NM, United States

HIGHLIGHTS

- A framework for processing streaming data in parallel in a distributed-memory manner is described.
- The framework performs message-passing to achieve parallelism, using either an MPI or sockets (ZMQ) backend.
- Performance of MPI versus sockets (ZMQ) is benchmarked for communicating large numbers of small messages.
- Three algorithms are presented for processing streaming graph edges to deduce structure and patterns in the graphs.

ARTICLE INFO

Article history:
Received 25 October 2012
Received in revised form
8 November 2013
Accepted 11 April 2014
Available online 6 May 2014

Keywords:
Streaming data
Graph algorithms
Message passing
MPI
Sockets
MapReduce

ABSTRACT

The need to process streaming data, which arrives continuously at high-volume in real-time, arises in a variety of contexts including data produced by experiments, collections of environmental or network sensors, and running simulations. Streaming data can also be formulated as queries or transactions which operate on a large dynamic data store, e.g. a distributed database.

We describe a lightweight, portable framework named PHISH which provides a communication model enabling a set of independent processes to compute on a stream of data in a distributed-memory parallel manner. Datums are routed between processes in patterns defined by the application. PHISH provides multiple communication backends including MPI and sockets/ZMQ. The former means streaming computations can be run on any parallel machine which supports MPI; the latter allows them to run on a heterogeneous, geographically dispersed network of machines.

We illustrate how streaming MapReduce operations can be implemented using the PHISH communication model, and describe streaming versions of three algorithms for large, sparse graph analytics: triangle enumeration, sub-graph isomorphism matching, and connected component finding. We also provide benchmark timings comparing MPI and socket performance for several kernel operations useful in streaming algorithms.

1. Introduction

Streaming data is produced continuously, in real-time, making streaming computation different from more familiar numerical, informatics, and physical simulation computations, which typically read and write archived data to a file system. The stream may be infinite, and the computations can be resource-constrained by the stream rate or by available memory [14,10]. For example, it may not be possible for a computation to "see" a datum in the stream more than once. A calculation on a datum may need to finish before the next datum arrives, in order to "keep up" with the stream. While attributes of previously seen datums can be stored, such "state" information may need to fit in memory (for speed of access), or be of finite size, even if the stream of data is infinite. The latter constraint typically requires a mechanism for "aging" or "expiring" state information.

There are at least two motivations for computing on streaming data in parallel: (1) to enable processing of higher stream rates and (2) to store more state information about the stream across the aggregate memory of many processors. Our focus in this paper is on a distributed-memory parallel approach to stream processing, since commodity clusters are cheap and ubiquitous, and we wish to go beyond the memory limits of a single shared-memory node.

A natural paradigm for processing streaming data is to view the data as a sequence of individual datums which "flow" through a collection of programs. Each program performs a specific

* Corresponding author.
E-mail address: sjplimp@sandia.gov (S.J. Plimpton).

http://dx.doi.org/10.1016/j.jpdc.2014.04.001
0743-7315/

computation on each datum it receives. It may choose to retrieve state information for previous datums, store state for the current datum, and/or pass the datum along as-is or in altered form to the next program in the stream. By connecting a collection of programs together in a specified topology, which may be a linear chain or a branching network with loops, an algorithm is defined which performs an overall computation on the stream.

On a shared-memory machine, each program could be a thread which shares state with other threads via shared memory. For distributed-memory platforms, each program is an independent process with its own private memory, and datums are exchanged between processes via some form of message passing. This incurs overhead which may limit the stream rates that can be processed, since datums must be copied from the memory of one process into the memory of another via a message; however, distributed memory parallelism has the other advantages discussed above.

We are aware of several software packages that work with streaming data. Some are extensions to the MapReduce model [6] with add-ons to Hadoop [9] that enable incremental map reduce on streaming data with intermediate results made available continuously [5,13]. Other packages implement their own streaming framework, such as the S4 platform [15] and the Storm package [20], recently released by Twitter. The former distributes the stream to processing elements based on key hashing, similar to a MapReduce computation, and was developed for data mining and machine learning applications. The latter is advertised as a real-time Hadoop, for performing parallel computations continuously on high-rate streaming data, including but not limited to streaming MapReduce operations.

There are also commercial software products that provide stream processing capability. IBM has its InfoSphere Streams system [11], designed to integrate data from thousands of real-time sources and perform analyses on it. EsperTech has a product Esper [7] which computes on a stream of "events" to match patterns and extract meaning. SQLstream [19] is a company devoted to processing real-time data; its software enables SQL queries to be made on streaming data in a time-windowed fashion.

We also note that the dataflow model [12], where data flows through a directed graph of processes, is not unique to informatics data. The open-source Titan toolkit [23], built on top of VTK [22], is a visualization and data analytics engine, which allows the user to build (through a GUI) a network of interconnected computational kernels, which is then invoked to process simulation or other data that is pipelined through the network. Often this is archived data, but some streaming capabilities have recently been added to Titan.

In this paper we describe a new software package – PHISH – which is detailed in the next section. PHISH is a lightweight, portable framework written to simplify the design and development of parallel streaming algorithms, including but not limited to the graph algorithms described in Section 3. PHISH defines a flexible streaming communication model while leaving the details of data storage and computation to the algorithm developer. This makes PHISH ideal for the needs of research and development, rather than a turn-key or production system.

Because we needed a means of running streaming algorithms on a wide variety of parallel machines and clusters for our own research, and further wished to compare the performance of different networking technologies, the PHISH communication model has been implemented on top of two backend messaging libraries, the ubiquitous message-passing interface (MPI) library [8], and the socket-based open-source ϕMQ library [24], (pronounced zero-MQ and hereafter referred to as ZMQ). The MPI backend allows PHISH programs to run on virtually any monolithic parallel machine. The ZMQ backend can be used to run on a distributed network of heterogeneous machines, including hosts that are geographically dispersed or deployed in a cloud. The design of PHISH explicitly allows for the possibility of additional backends, to support experimentation with alternate networking technologies, e.g. sophisticated new interconnects.

Of the packages discussed above, PHISH is most similar to Storm, which also uses ZMQ for socket-based communications. Though we began work on PHISH before Storm was released, the model the two packages use for connecting computational processes via communication patterns to enable parallelism is similar. Storm has additional features for fault tolerance and guaranteeing that each datum in a stream is processed, which PHISH does not provide. For the MPI-based parallel machines we use most often, these issues are not as important, and the current MPI standard (3.0) does not have support for fault tolerance. Storm places some limitations on how data that is shared among processes is packaged, when used with certain programming languages. PHISH imposes no structure on the data exchanged among processes regardless of programming language, and can be run with alternate communication backends, as outlined above.

In the next section we give a brief description of the PHISH framework and its features, and illustrate with examples how both traditional and streaming MapReduce computations can be performed in parallel. In Section 3, we outline three graph algorithms implemented using PHISH that operate on edges arriving as a stream of data. Finally, in Section 4 we provide benchmark timings for prototypical stream operations, running on a traditional parallel HPC platform, using both the MPI and socket options in PHISH. The results are a measure of the maximum stream rates that PHISH can process.

2. PHISH pheatures

PHISH, which stands for Parallel Harness for Informatic Stream Hashing, is a lightweight framework, written in a few thousand lines of C, C++, and Python code. Aside from the acronym, we chose the name because "phish" swim in a stream (of data in this case).

The framework has two parts. The first is a library which can be used by programs written in a variety of languages (we supply bindings for C, C++, and Python). We refer to a program that uses the PHISH library as a "minnow" because such programs are typically (though not necessarily) small, performing a single, specific operation on individual datums in the stream. Of course, nothing in the library prevents minnows from growing in complexity (perhaps becoming sharks or even whales!), but we find in practice that PHISH computations work best and are easiest to develop and debug when each type of minnow focuses on doing one thing well. Writing a minnow simply requires registering one or more callback functions with the library that will be invoked when datums arrive. Additional library functions are used to unpack datums into their constituent fields, and pack new datums to be sent downstream to other minnows. The PHISH library handles the actual communication of a datum from one minnow to another, via the included MPI or ZMQ back-ends.

The second part of the framework is a pre-processing tool named "bait" which enables a computation using one or more minnows to be specified. The bait tool reads a simple input script with 3 kinds of commands. "Minnow" commands specify which minnow executables to launch, and any command-line arguments they require. "School" commands specify how many minnows of each type to launch and, optionally, how to assign them to physical processors. "Hook" commands define a communication pattern used to route datums between minnows in schools. Examples of such patterns are "paired" (each minnow sends to one receiver), "roundrobin" (send to every receiver in round-robin order), "hashed" (send to a specific receiver based on a hash operation), and "bcast" (send to all receivers). The "bait" tool converts the input script into a file suitable for launching all the minnow