SiftD: A CPU & GPU Distributed Hybrid System For SIFT

Mahdi Soltan Mohammadi
Electrical and Computer Engineering Department
Yazd University
Yazd, Iran
m.s.mohammadi.k@stu.yazduni.ac.ir

Mehdi Rezaeian
Electrical and Computer Engineering Department
Yazd University
Yazd, Iran
mrezaeian@yazduni.ac.ir

Abstract— Using distributed and parallel computing systems have become a de facto for implementing scientific and industrial applications, which require tremendous amount of computing resources. As a widely used approach, general purpose distributed frameworks, like Hadoop, have provided us with many facilities to develop a distributed computing system for our applications. These General-purpose frameworks are flexible but their flexibility can only take us so far. There are many applications, which not all of their requirements can be met by these frameworks. Image matching using SIFT algorithm can be a good example of these applications. SIFT is a highly complex algorithm for extracting robust features from pictures. This paper outlines most important motivations and challenges for implementing specialized distributed systems. We present SiftD, an application for distributing and parallelizing SIFT algorithm. It uses networked computers to distribute the algorithm. Inside each system, multi-core processors and Graphical Processing Units (GPUs) are used to parallelize execution. SiftD’s performance and capability for utilizing different computing resources has been evaluated. Results show its performance is generally higher than 93%, which is a fairly appropriate performance. Furthermore, it can utilize broad range of hardware platforms.

Keywords— distributed systems; distributed computing; distributed implementation; feature extraction; GPU programming; parallel processing; SIFT;

I. INTRODUCTION

We are witnessing an ever-growing demand for computing resources. This is the case for both, scientific and industrial applications. Clearly, traditional system architectures with single-core or multi-core processors could not answer these demands. Consequently, using distributed and parallel architectures for such applications have become more and more popular. For proper utilization of these architectures, we need to design and implement software systems that best suits them. In order to achieve this goal both hardware requirements and intended application requirements should be considered. There are general purpose distributed frameworks, which can be customized and used for different applications.

Most of the general purpose distributed frameworks are implemented using a popular programming model called MapReduce. MapReduce is a programming model and associated implementation for processing and generating large data sets [1]. MapReduce has flexible design and its popular implementations, like Hadoop [2], have further increased its flexibility with smart implementation choices, but their flexibility can only take us so far. These General-purpose frameworks can only be optimized for portion of the real world applications. Most of the computer application can be implemented through general purpose distributed frameworks. Nevertheless, final output utilization would not necessarily be optimized.

In this paper, we discuss important reasons and challenges for implementing specialized distributed systems. We present a specialized distributed system, designed and implemented for distributing SIFT algorithm. SIFT is an image feature extraction algorithm [3]. It is a very popular algorithm for image matching. It has been used in object recognition, image retrieval and many other fields. Although SIFT is very effective but it is also computationally expensive. We need to utilize distributed and parallel architectures, in order to make SIFT practical for real world applications, especially real time applications like online object tracking. Through parallel and distributed design our system, namely SiftD, utilizes both CPUs and Graphical Processing Units (GPUs).

Rest of the paper is arranged as follow: Section II briefly discusses general purpose distributed frameworks. The next section presents challenges, trends and motivations for designing and implementing specialized distributed system. SiftD’s architecture and implementation are discussed at Section IV. Section V presents evaluation and results. Finally, section VI concludes the paper.

II. GENERAL PURPOSE DISTRIBUTED FRAMEWORKS

General purpose distributed frameworks, also called distributed programming frameworks, are designed to minimize development effort on programmers part for distributing their intended algorithms. These systems usually provide programmers with high-level routines to write their programs in parallel manner; completely hiding issues concerning confusing details of parallelization, fault-tolerance, data distribution and load balancing. Recently developed general purpose distributed frameworks are mostly designed based on a programming model called MapReduce.

MapReduce is a programming model for processing and producing large data sets [1]. The general idea of MapReduce is as follow: user specifies a Map function that processes key/value pairs to generate a set of intermediate key/value
pairs, and then an optional user defined Reduce function merges all intermediate values associated with the same intermediate key. Many real world applications can be represented in this key/value format [1]. Real world implementation of MapReduce framework should be implemented so distributing tasks would be transparent to user. Codes developed using these implementations should be automatically parallelized and executed on a large cluster of nodes. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. Programmers without any experience with parallel and distributed systems should be able to easily utilize the resources of a large distributed system using these frameworks.

The most widely used and mature implementation of MapReduce is considered to be Hadoop [2]. The Apache Hadoop software library uses MapReduce programming model to distribute processing of large data sets across clusters of nodes. It is mostly implemented in Java, therefore, is highly portable. Beside from computing module it includes other modules, like Hadoop Distributed File System, to handle all distributing related task. Users developing distributed systems using Hadoop are mostly concerned with writing Map and Reduce functions.

There are other programming frameworks, which more or less follow MapReduce conceptual procedure. Piccolo is a data-centric programming model for writing parallel in-memory applications in data centers [4]. Unlike MapReduce, Piccolo allows computation running on different computing nodes to share distributed mutable state via a key-value table interface. In particular, applications can specify locality policies to exploit the locality of shared state access and Piccolo's run-time automatically resolves write-write conflicts using user defined accumulation functions. There is also Oolong that uses shared partitioned table schema same as Piccolo but it is designed for asynchronous applications; its global shared states do not need synchronization [5].

III. SPECIALIZED DISTRIBUTED SYSTEMS, MOTIVATIONS AND CHALLENGES

General purpose distributed frameworks have novel designs and they can be used in developing systems for many real world application. Not all applications can use these systems and optimally utilize the distributed and parallel architecture. As flexible as these frameworks are, they suited only for a specific class of tasks and their associated data. Tasks that are being implemented with general frameworks must have these two important properties otherwise their implementation may not be optimal:

- We should be able to easily partition large data sets corresponding to the tasks; data dependency should be limited so we can partition data sets as little or as big as desired.
- The pertaining algorithm to the task should be simple and parallel in nature. Therefore, we can accommodate it in simple functions, like Map and Reduce, to be distributed on our cluster of nodes.

Most of the tasks pertaining to text based information retrieval in search engines are suited for being distributed by general purpose distributed frameworks, e.g. crawling, indexing, sorting, simple search and other related tasks. Based on number of reasons, we cannot appropriately deploy all applications using general purpose distributed frameworks. We discuss such reasons at the next section.

A. Motivations

Proper utilization of distributed and parallel hardware architectures is the most important reason for developing a specialized system instead of using existing general systems. There are many real world applications with special properties that no general purpose distributed framework can be optimally customized to deploy their related algorithm, in section IV we introduce such application and describe its special properties.

Adapting to special hardware properties and utilizing new hardware innovation is another motivation to design our own distributed software system. For example consider Remote Direct Memory Access, also called RDMA, which is hardware facility formally found in High Performance cluster (HPC), but recently also available in 10GB Ethernet [5]. RDMA operations allow a process to read (or write) from a pre-registered memory region of a remote process without involving the CPU on the remote side. Compared to traditional message passing, RDMA achieves the smallest round-trip latency, highest throughput, and lowest CPU overhead. Although Mitchell, Geng and Li have designed Pillar [6], which is a general purpose distributed framework with similar design to Piccolo, that partially utilizes RDMA for high performance data reads but this is only one example and there many other hardware properties and innovations that can be exploited.

MapReduce is the dominant schema for designing general purpose distributed frameworks. We expect implementations based on MapReduce programming Model to have highest possible performance for its suitable tasks. This is not necessarily true. Pavlo, Paulson and Rasin showed that Parallel DBMSs can outperform Hadoop even in tasks suitable for MapReduce [7]. This is also a good reason for designing and implementing specialized distributed systems.

B. Challenges

Specialized distributed systems design and implementation challenges are mostly the same as general distributed systems that can be found in textbooks and papers [8]. Fortunately, in most cases developing specialized systems can be much easier than general systems since we can exploit application specific properties to reach simpler design and higher performance. General design and implementation challenges of distributed systems are classified in eight areas [8]: 1) heterogeneity; 2) Openness; 3) Security; 4) Scalability; 5) Failure handling; 6) Concurrency; 7) Transparency; 8) Quality of service. Depending on nature of our application, some of these challenges might be more important. We will discuss challenges related to our devised system, namely sifD, and will briefly explain how we can deal with them.
IV. SIFTD

In this section, we describe SIFTD. SIFTD is a hybrid distributed system that uses both CPUs and GPUs to distribute and parallelize SIFT algorithm. SIFT is an image feature extraction algorithm. It is a very powerful algorithm for image matching and object recognition, consequently, it has broad applications in different fields of image processing. SIFT is comprised of four major steps [3]:

**Scale-space extrema detection:** First, location and scale of candidate points that can be repeatedly assigned under differing views of the same object are detected. Initial candidates are extrema in difference of Gaussian scale space of input image.

**Keypoint localization:** Before accepting any detected candidate in first step, a detailed fit must performed to the nearby data for location, scale, and ratio of principal curvatures. Using this procedure, we can reject points that have low contrast and are poorly localized along edges.

**Orientation assignment:** In order to create invariance feature to rotation, an orientation is assigned to each keypoint so its descriptor can be computed relative to this orientation.

**Keypoint descriptor:** Finally, keypoint descriptors are created. Each descriptor is a vector, normally consisting of 128 integer numbers, which describes image’s local orientation properties in the vicinity of a keypoint.

SIFT is a very complex and computationally expensive algorithm. We cannot use SIFT in the real world tasks utilizing only a single computer station with a few number of processors. In order to make SIFT usage practically feasible, we need to harness parallel and distributed architectures.

A. Special Characteristics

SIFT has a number of special characteristics that motivate us to develop a fresh application for distributing it, instead of using already existed General-purpose frameworks. These characteristics can be classified into three categories: 1) Algorithm related characteristics; 2) Associate data characteristics; 3) Required hardware characteristics.

SIFT is much more complicated than typical algorithms deployed in general-purpose distributed systems, like sorting and word count. We cannot simply divide SIFT in Map and Reduce functions or any other similar concepts. SIFT itself includes four steps, each one involving lots of independent lengthy computations. Best option for distributing SIFT using a system like Hadoop is executing the entire algorithm inside Map function. This is not a good idea since it will decrease performance gradually.

There are three major differences between SIFT’s associated data and data like text that is usually handled in general-purpose distributed frameworks. For starters, SIFT operates on images. Atomic units in text based data are words, which are very small in size. On the contrary, a single image is fairly big. When processing images, it is preferable not to divide them into smaller pieces unless necessary. Secondly, intermediate data produced during SIFT algorithm have different properties compared to intermediate data produced in algorithms like sorting and word count. SIFT’s intermediate data are much bigger in size and complexity. Finally, the most important difference can be attributed to the great magnitude of dependencies that exists between intermediate data produces during the algorithm. That is why, we cannot simply divide SIFT into smaller parts.

SIFT includes lots computations that can be categorized in what we call stream processing computation. This kind of computation is suited for processing on GPUs, which are graphic cards’ processing units. GPUs have better performance for stream processing compared to CPUs. To maximize our system performance, we have decided to utilize both CPUs and GPUs. Although there are some limited supports for GPU programming in number of general purpose distributed frameworks including Hadoop but they are not actually optimized for utilizing both CPU and GPU. This is another reason for us to design and implement our own specialize distributed system for SIFT.

B. Implementation Considerations

We have implemented SiftD entirely in C++. To make implementation as portable as possible, only POSIX and ANSI C++ standard libraries have been used [9]. This makes siftd portable across most of Linux and UNIX platforms. SiftD uses two separate implementations of SIFT, one for utilizing CPUs and another one for utilizing GPUs. SiftCU is an optimize CUDA based implementation of SIFT algorithm that utilizes Graphical Processing Unites [10]. SiftD uses siftd for running the algorithm on GPUs. SiftD comprised of three main components and each component comprised of number of modules. In the implementation, each component is a separate process and each module is a separate thread inside.
its components. This can help improve performance since it decreases context switch time and maximizes multi-core architecture utilization. Threads are implemented using POSIX’s Pthread library. External communications between components has been carried out with POSIX sockets. POSIX condition variables have been used for internal communication between modules of a component.

C. Architecture

As it shown in Fig. 1, siftD includes three major components: Masters, Workers and Distributed File Systems (DFSS). In order to alleviate load balancing and improve flexibility, we have decided that more than one Master can exist, although, this can increase complexity. Fig. 1 shows a hypothetical system with four stations, each station can have heterogeneous hardware and software configuration. This figure shows that each station can run any combination of siftD’s components depending on its resources. One Master can assign its jobs to any worker in the system and any worker may get job from more than one master.

Fig. 2 shows system’s three major components. This figure also shows external communications between these components. Master’s internal design and communications between its modules is shown in Fig. 3. As you can see, there are two job list, one for normal jobs and one for big jobs, which could not be processed on a single station. Each module is implemented in a separate thread. Internal communications between modules is depicted with dotted lines that are accomplished using POSIX’s condition variables. Worker’s internal design is illustrated in Fig. 4. We briefly describe each module responsibilities in the system.

Master:
- Main Module: 1) Reads configuration file, jobs list, and workers list. 2) Starts other modules.
- Worker Service: 1) Receives responses to job requests and workers status. 2) Notifies Job Manager and Big Job Handler about workers status.
- Job Manager: 1) Assigns jobs to workers.
- Big Job manager: 1) Creates appropriate number of normal jobs from big jobs and place them in big job list. 2) Aggregates results from partitions of a big job.
- Big Job Handler: 1) Assigns big job partitions to workers.

Worker:
- Main Module: 1) Reads configuration file and masters list. 2) Starts other modules.
- Master Service: 1) answers to masters’ inquiries. 2) Sends out advertisement messages about worker’s stat
- Job Manager: 1) Receives job request and sends job reply. 2) Notifies Algorithm Manager about new job.
- Algorithm Manager: 1) Runs SIFT on CPU and GPU.

DFS:
- File Service: 1) Answers read and write requests. 2) Handles file partitioning and aggregation requests.
Table I. Characteristics of images used in evaluations

<table>
<thead>
<tr>
<th>category</th>
<th>Picture size in pixel</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>Less than 0.5 Mega pixel</td>
<td>35</td>
</tr>
<tr>
<td>Medium</td>
<td>0.5 - 1 Mega pixel</td>
<td>35</td>
</tr>
<tr>
<td>Big</td>
<td>1 – 2 Mega pixel</td>
<td>30</td>
</tr>
<tr>
<td>Total</td>
<td>---</td>
<td>100</td>
</tr>
</tbody>
</table>

V. EVALUATIONS AND RESULTS

In this section, we evaluate siftD’s performance and its capability to utilize different hardware resources. All tests have been completed using Ubuntu 12.04 operating system, nevertheless, implementation itself was tested on other operating systems like CentOS 6 for portability check. A set of 100 pictures with different size and properties were used in evaluations. Table I shows properties of these pictures.

A. Various Hardware Utilization Capability

SiftD was tested using numerous hardware platforms to evaluate its capacity to harness various resources. Image set used in this test is shown in Table I. Table II shows all hardware configuration properties used in the test. Notice that configuration No. 2 is actually a collection of four computer stations each with a 2 core multi-core processor (hence $2 \times 4$ core in total). Also configuration No. 4 is a combination of configurations No. 1 and No. 3. Similarly configuration No. 5 is combination of configurations No. 2 and No. 4. All systems are connected using 100Mb Ethernet network connections.

Results are shown in Fig. 5. You can see using more computing power results in smaller processing time. It is obvious that system has high capability to utilize different hardware configurations at the same time.

B. Performance Evaluation

In order to evaluate siftD’s performance, we preferred to use number of computer stations with identical hardware configuration. Using identical systems is simpler and more meaningful for performance evaluation. Stations used in this test have same configuration as configuration No. 2 in Table II, except for in this test we used only one processing core in each station. Table III. shows processing time of image set in Table I. for different number of stations running siftD.

Equation (1) was used to calculate system’s performance, notice that processing time for a single station is considered as base measure:

$$P_n = \left( \frac{t_1}{(t_n \times n)} \right) \times 100$$  \hspace{1cm} (1)

In this equation, $P_n$ and $t_n$ respectively are performance and processing time of system when using $N$ number of stations; $t_1$ is processing time for a single station. Figure 6 shows system performance for using different number of stations. General performance is higher than 93 percent, which could be considered satisfactory. The slight discrepancy in performance can be attributed to disruptions in load balancing. Higher performance could be reached using a better load balancing and faster network connections.

Table II. Various hardware configurations for hardware utilization capabilities evaluation

<table>
<thead>
<tr>
<th>Config</th>
<th>Processor</th>
<th>Number of Cores</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. 1</td>
<td>Intel core i-7 2600</td>
<td>4</td>
<td>4 GB DDR3</td>
</tr>
<tr>
<td>No. 2</td>
<td>Intel core i-3 2100 × 4</td>
<td>2 × 4</td>
<td>2×4 GB DDR3</td>
</tr>
<tr>
<td>No. 3</td>
<td>NVidia GeForce 440GT Graphic</td>
<td>1 GB DDR3</td>
<td></td>
</tr>
<tr>
<td>No. 4</td>
<td>Intel core i-7 2600 &amp; NVidia GeForce 440GT Graphic</td>
<td>4 + Graphic</td>
<td>4 GB DDR3 + 1 GB GDR3</td>
</tr>
<tr>
<td>No. 5</td>
<td>All (various)</td>
<td>various</td>
<td>various</td>
</tr>
</tbody>
</table>

Figure 5. Processing time for five hardware configuration in Table II.

Figure 6. SiftD’s performance for using different number of stations.

Table III. Processing time results when using different number of stations

<table>
<thead>
<tr>
<th>Number of Stations</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Time</td>
<td>749</td>
<td>401</td>
<td>263</td>
<td>199</td>
</tr>
</tbody>
</table>
VI. CONCLUSION

In this paper, we discussed important motivations for designing and implementing specialized distributed systems. These reasons confirm that we cannot build optimal distributed system for all real world application using existing general-purpose frameworks like Hadoop. Nonetheless, there are many tasks that can be optimally implemented with general purpose distributed framework. Characteristics of these tasks were discussed. General-purpose frameworks are better suited for simple algorithms and associated data sets that do not have boundless scale of dependency. On the other hand, for applications involving complex algorithms and high data dependency, it is better to design and implement specialized distributed system. SIFT is a popular image feature extraction algorithm. We described characteristics of SIFT algorithm and its associated data. SIFT cannot be optimally distributed using general-purpose distributed frameworks. We have devised siftD, a specialized distributed application for distributing SIFT algorithm. It is a hybrid system that utilizes both CPUs and GPUs. Architecture of siftD was presented in details. We evaluated siftD’s performance and usability. Test results showed siftD has high capacity for utilizing various hardware systems. Our system’s performance is generally higher than 93 percent. We did not use high performance network connections, therefore, this is an adequate performance. We could gain better performance utilizing more sophisticated hardware systems and better load balancing.

REFERENCES