SAPIENZA
UNIVERSITÀ DI ROMA

HPC Techniques for Large Scale Data Analysis

Giancarlo Carbone

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the Degree of
Doctor of Philosophy in **Computer Science**
at the **Sapienza University of Rome**

December 9, 2015

Contents

Introduction

In the present work we apply High-Performance Computing techniques to two Big Data problems. The first one deals with the analysis of large graphs by using a parallel distributed architecture, whereas the second one consists in the design and implementation of a scalable solution for fast indexing and searching of large datasets of heterogeneous documents.

In recent years, there has been a steadily growing interest in the study of real-world networks [1], since a number of natural and artificial phenomena may be described by using networks, i.e. sets of interconnected nodes. The size of the networks under study has grown up to millions of nodes and hundred-million or even billions of connections. Examples of such huge networks are the Web with a number of pages exceeding 45 billion, or social networks like Facebook, Twitter, Google+ [2]. Large-scale networks can be found also in biology: *e.g.,* protein interaction networks, the human brain or the metabolic interaction networks [3].

Networks are often represented as *graphs* having vertices connected by edges. When applied to large graphs, even simple algorithms like Breadth First Search (BFS) require significant computing resources to provide timely results.

Most of the platforms in use to that purpose are based on parallel architectures that can be grouped in two major categories: shared memory and distributed memory architectures. Shared memory systems have many advantages from the programming point of view but are limited both in the size of the memory and in the number of processors. On the other end, distributed systems are more difficult to program but can include thousands of computing nodes with a huge amount of total memory. In principle, by leveraging a suitable distributed system, there is no limit to the size of the network that can be studied.

Generally, a distributed system is a cluster of computing nodes interconnected via a wired network. Communications are typically carried out by using Message Passing Interface (MPI) primitives. Each computing node in a distributed system can be equipped with a single- or a multi-core CPU. In GPU clusters, each node hosts also one or more GPU devices.

If the size of a graph does not fit in the memory of one single node, it is necessary to split the graph over multiple computing nodes in order to analyze it. For the purpose of the present work, we designed and implemented solutions for studying graphs on a cluster of computing nodes equipped with Graphic Processing Units (GPUs): while each GPU provides shared-memory based parallel computational power, interconnected computing nodes form a distributed memory system in which cooperation is realized by exchanging messages.

For the analysis of large networks, we developed Multi-GPUs implementations of three graph algorithms: Breadth First Search (BFS), ST-Connectivity (ST-CON) and Betweenness Centrality (BC).

Breadth First Search is a fundamental building block for solving many graph-related problems like computing the maximum-flow/minimum-cut, testing a graph for bipartiteness and, more in general, for traversing a graph, i.e., visiting all the vertices and edges from a given start vertex. Unfortunately, parallel algorithms for the BFS are difficult to implement efficiently on GPUs, due to the low arithmetic intensity and lack of spatial locality of the algorithm. Furthermore, in a distributed system, computing nodes cooperate exchanging data thus, a large fraction of the running time, is spent in communication among nodes. Moreover, size and number of messages exchanged may fluctuate during algorithm execution and communication patterns are irregular. Therefore communication is both a performance bottleneck and inhibitor of scalability [4–6].

Starting from an existing Multi-GPU BFS implementation [7], in Chapter 1 we present our enhancements to that solution [8], improving both performance and scalability by reducing data exchanges among nodes thanks to a different communication pattern and a reorganization of internal data structures.

In the same Chapter, we also employ our Multi-GPU BFS to solve the ST-Connectivity problem (ST-CON) which consists in deciding, for vertices s and t in a graph, if t is reachable from s. Such problem may be simply solved by executing a BFS starting from s and checking if t is reached. Nevertheless a more efficient solution can be explored: starting two concurrent BFS, one from s and the other from t and checking if they intersect at some point. However, that approach opens also some issues: BFS data structures need to be modified to cope with two concurrent independent searches and it is necessary to check if and when the two searches intersect. We provide two implementations to efficiently solve ST-CON [9] that highlight the impact of atomic operations in GPU.

As already mentioned, BFS is a fundamental building block for solving many graph-related algorithms and the Betweenness Centrality (BC) is one of those. BC is, by now, one of the most popular metrics used to determine the “relevance” (or the centrality) of a node in a network. It has been used in many fields like the study of the interactions in social-networks [10, 11], lethality in biological networks [12], identification of leaders in terrorist networks [13], and so on.

The betweenness centrality of a node is based on the number of all-pairs shortest paths passing through that node. Exact computation of BC scores is computationally-expensive, the fastest known algorithm for calculating BC scores has $\mathcal{O}(nm)$ time-complexity for unweighted graphs [14]. Therefore BC computation for a large scale graph is an extraordinary challenge that requires high performance computing techniques to provide results in a reasonable amount of time. Leveraging the knowledge acquired for the BFS and

ST-CON, in Chapter 2 we present the techniques we developed to speed-up the computation of the BC on Multi-GPU systems. In particular, time and space complexity is reduced by using graph topology manipulations that, although modify the graph, still guarantee a correct BC score calculation. Then we resort to both coarse- and fine-grained parallelizations to squeeze the highest computational power from the distributed system. Experimental results on synthetic and real-world graphs show that the proposed techniques are well suited to compute BC scores in graphs which are too large to fit in the memory of a single computational node. In particular, the computation time of a 117 million undirected edges graph is reduced to less than 2 hours.

In Chapter 3, we deal with the second problem faced in our work: how to reduce the time required to index and search very large sets of heterogeneous textual data [15]. Current digital forensics tools and mindsets are no longer adequate to meet the scope and complexity of today's threats. As a matter of fact, due to the steadily growing size of data managed by IT centers and the rapid growth of Cloud services providers, seizures, for forensic purposes, of very large storage devices are expected in the next future. Therefore our work aims at building a cost-effective solution for analyzing large unstructured data sets so providing the ability to quickly retrieve information to investigators. We exploit High Performance Computing (HPC) techniques to index and search huge amount of data that may include (but are not limited to) emails, documents, plain text files, web pages, etc. The final product is a technology for indexing and searching digital forensics data and a "proof of concept" browser-based search system that may be immediately used by an investigator.

Chapter 1

Breadth First Search and ST-Connectivity

An efficient Breadth First Search (BFS) is a fundamental building block for solving many graph-related problems like finding the diameter or testing a graph for bipartiteness as well as more sophisticated problems, such as finding community structures in networks or computing the maximum-flow/minimum-cut, problems that may have an immediate practical utility.

However, it is not easy to develop an efficient parallel implementation of the BFS algorithm because of its spatial non-locality and low arithmetic intensity.[a] On shared memory systems, a large fraction of the running time is spent accessing memory while on distributed memory systems, it is spent communications among computational nodes.

Despite these facts, many authors have demonstrated that, by carefully addressing these problems, it is possible to implement high performance parallel and distributed BFS [6, 16–21].

Optimization strategies strongly depends on the properties of the graph under study. Regular graphs, with high diameter and regular structures (like those used in physical simulations), present fewer difficulties with respect to

[a]Arithmetic intensity is the ratio of floating point and/or integer operations to memory accesses.

graphs that have short diameter and skewed degree distributions. In the latter case, a small fraction of vertices has a very high degree whereas the majority has only few connections. This characteristic complicates load-balancing among computational nodes because, in many graph algorithms, like BFS, work units are associated with vertices degrees. Due to the limited amount of memory available on a GPU, the graph size is constrained to few millions of vertices, neverthless using recent K80 GPU with up to 24 GB shared memory if is possible to handle up to few hundreds millions of vertices.

In this chapter we describe a parallel distributed implementation of BFS algorithm over multiple GPUs, then we discuss how to use the BFS implementation to solve ST-Connectivity problem.

For the BFS on Multi-GPUs, we started from the solution proposed in [7]; in that work two main issues are tackled: imbalance of the workload among threads and the communication of duplicated data. We enhanced that work with the following main contributions:

1. the implementation of a modified Compressed Sparse Row (CSR) data structure which allows for a faster and complete filtering of already visited edges;

2. the optimization of data communication by using new communication patterns;

3. testing our BFS solution over real-world graphs.

To understand the overall algorithm and its implementation on Multi-GPU, in Section 1.1.3 we introduce the parallel and distributed BFS problem and algorithms, while in Section 1.2 we review the Multi-GPU BFS implementation described in [7] whereas Sections 1.2.3 and 1.2.4 describe new proposed implementation and its optimizations. In Section 1.2.5 performance on synthetic and real graphs is reported.

After that, in Section 1.3 we discuss the ST-Connectivity problem and how the BFS Multi-GPU solution can be efficiently used to solve it.

1.1 Background and related work

1.1.1 Graph Definitions

A network can be represented using a graph $G = (V, E)$ where vertices V corresponds to network entities and edges E to their relations. Two vertices $u, v \in V$ are directly connected if $\exists\ e = (u, v) \in E$, while $N = |V|$ is the number of vertices in G whereas, $M = |E|$ is the number of edges.

G can be:

- **directed**: if the pair (u, v) is oriented, meaning there is a relation from u to v
- **weighted**: if each edge e has associated a scalar value p_e.
- **multi-graph**: if there are multiple distinct edges between two vertices.
- **simple**: unweighted, undirected graph containing neither graph loops nor multiple edges, where a loop is an edge (u, u) .

Since our work is focused on simple graphs, whenever possibile a pre-processing step is applied to transform the input graph into a simple one: multiple edges or loops are removed and directed graphs are transformed into indirected by adding required edges.

A path between two vertices $s, t \in V$ is a sequence of edges $\langle e_0, e_1, \ldots, e_d\rangle$ such that $e_0 = (s, u_1), e_{d-1} = (u_{d-1}, t)$, where the path has length d. G is said to be **connected** if there exist a path between any two vertices in V. Two vertices s, t are reachable if there is a path between s and t in G. The connected components of a graph are the equivalence classes of vertices according to the relation "is reachable from". A graph can have many **connected components**, i.e., subsets of nodes that are connected. If G is simple and every pair of distinct vertices is connected by a unique edge, then G is **complete**. A closed path in which some vertices are repeated is called **cycle**. A **tree** is a connected graph with no cycle. The degree of a vertex $deg(v)$ is the number of edges incident on it.

The shortest path between two vertices s, t is a sequence of unique vertices of minimum length. Given any two nodes s, t in G, the **distance**, is measured as the length of shortest path between s and t. The diameter D of a unweighted undirected graph $G = (V, E)$ is the maximum pairwise distance $D = max_{s,t \in V} d(s, t)$ between reachable vertices, where d is the number of the edges along the shortest path from s to v [22].

1.1.2 GPU

Graphics Processing Units (GPUs) are specially designed processors typically used for computer graphics, that in recent years have been strongly employed to perform parallel computations. GPUs have dedicated memory with many-core processors specifically designed to perform data-parallel computation in a shared memory system, so that can easily process naturally parallel tasks. A programmer may choose to offload these parallel tasks to a GPU to free up the CPU for more serial parts of an algorithm.

In 2006, NVIDIA launched CUDA, a general purpose parallel computing architecture that allows using C programming language to code algorithm for NVIDIA GPU devices. The Multi-GPU solutions described in this dissertation have been realized using CUDA.

GPUs have been successfully used in accelerating many regular applications specially those involving dense matrix operations and, more recently, irregular and low-arithmetic intensity applications like graph traversal-based algorithms [23, 24]. Workload imbalance and uncoalesced memory accesses are major bottlenecks for GPU-based traversal algorithms. However, in recent years several authors have proposed efficient parallel and distributed implementations for the Breadth First Search (BFS) [4, 16, 21, 25] because it represents a building-block for the solution of more sophisticated problems on unweighted graphs like minimum-cut [22], ST-connectivity [9, 26] and betweenness centrality as well [14].

1.1.3 Level-synchronous parallel BFS

Given a simple graph *G(V, E)* with V vertices and E edges, the BFS algorithm starts from a root vertex $s \in V$ and traverses all the vertices that can be reached from s, yielding a tree rooted in s.

In a classic queue-based BFS algorithm, a graph is traversed iteratively where at each level the hop distance from the root increases. For each level, a current queue CQ is used to store the current set of vertices (a.k.a. current frontier), all its neighbors compose the so called *Next Level Frontier Set* (NLFS). Among the vertices in the NLFS, only those not visited yet are selected for the *next level set* of vertices and stored in a next level queue NQ. At the end of the iteration the next level queue NQ becomes the current queue CQ. Graph traversal terminates when no more vertices are added to NQ.

To provide the resulting tree rooted in s, it is necessary to keep track of parent-child relationships discovered during the graph traversal; this can be accomplished by using a predecessors array to store the parent vertex. The same array can also be used for checking if a vertex has already been visited.

Algorithm 1 Level-synchronous parallel BFS Algorithm

```
    Input: graph G(V,E), starting vertex $s$
    Output: array of predecessors $pred$
    Data: $CQ$ and $NQ$; $nlfs\_array$
 1: $pred[v_j] = -1, \forall\ v_j \in V$
 2: $CQ, NQ \leftarrow \emptyset$
 3: enqueue $s \rightarrow CQ$
 4: $pred[s] = s$
 5: while $CQ \neq \emptyset$ do
 6:     for all $u_i$ in $CQ$ in parallel do
 7:         for all $v_j$ neighbor of $u_i$ in parallel do      ▷ Expand the NLFS and visit all the neighbors
 8:             if $p[v_j] == -1$ then
 9:                 $pred[v_j] = v_i$
10:                 enqueue $v_j \rightarrow NQ$
11:             end if
12:         end for
13:     end for
14:     $CQ \leftarrow NQ$
15:     $NQ \leftarrow \emptyset$
16: end while
```

This method is referred in literature as Level Synchronous BFS [6, 26, 27] because, to ensure the correctness of the computation in a parallel implementation, a synchronization is required at the end of each level. Algorithm 1 shows the pseudo-code for an implementation of a level-synchronous parallel BFS.

In “real world” graphs, the number of levels is of the same order of the diameter of the graph, that is short (~ 10). As reported by various authors [27, 28] the computation is dominated by only two or three levels.

During the BFS visit, at each iteration, the size of both the queues and the NLFS vary greatly. This irregular behavior makes it harder to correctly balance the load among the concurrent working threads (on both the GPU and multi-core CPU). The problem is exacerbated on *real-world* graphs with skewed distributions of vertex degrees.

To address the work unbalance among threads, a good solution is to have as many active threads as the number of elements in the NLFS. In this way, each thread is in charge of only one vertex and the entire NLFS can be processed in parallel.

In a distributed memory implementation, the graph *G(V, E)* is partitioned among computational nodes, thus each node is in charge of only a subset of the total number of edges and has direct access to them. Local edges can be processed as in a shared memory based implementation: for each local vertex in the NLFS it is easy to check if it has already been visited or not, and the computation of the predecessor vertices array can be carried out locally. However, information about edges belonging to other nodes have to be explicitly exchanged.

As a matter of fact, most of the execution time is spent sending and receiving data as shown in [7]. Moreover, the communication patterns required by graph algorithms are irregular and both the size of the messages and the couples of senders/receivers vary during the execution.

It is not surprising therefore, that, as reported by many authors, [4–6], the communication is the bottleneck of a distributed BFS.

Algorithm 2 shows the pseudo-code of a distributed BFS. Given the root

Algorithm 2 Distributed memory BFS

```
    Input: graph G(V,E), starting vertex $s$
    Output: array of predecessors $pred$
    Data: $CQ$ and $NQ$; $nlfs\_array$; total number of vertices in the current queues $totlen$; arrays to
    send/receive vertices $sendarray, recvarray$
 1: $CQ, NQ \leftarrow \emptyset$
 2: $pred[v_j] = -1, \forall\ v_j \in V$
 3: if $s$ is local then
 4:     $pred[s] = s$
 5:     enqueue $s \rightarrow CQ$
 6: end if
 7: while $totlen > 0$ do
 8:     $sendarray \leftarrow []$
 9:     $recvarray \leftarrow []$
10:     for all $u_i$ in $CQ$ in parallel do
11:         for all $v_j$ neighbor of $u_i$ in parallel do
12:             if $v_j$ is local then
13:                 if $pred[v_j] == -1$ then
14:                     $pred[v_j] = u_i$
15:                     enqueue $v_j \rightarrow NQ$
16:                 end if
17:             else
18:                 append $(u_i, v_j) \rightarrow sendarray$
19:             end if
20:         end for
21:     end for
22:     $SEND(sendarray)$                              ▷ Send vertices to other nodes
23:     $RECV(recvarray)$                              ▷ Receive vertices from other nodes
24:     for all $(z_k, w_k)$ in $recvarray$ in parallel do
25:         if $pred[w_k] == -1$ then
26:             $pred[w_k] = z_k$
27:             enqueue $w_k \rightarrow NQ$
28:         end if
29:     end for
30:     $CQ \leftarrow NQ$
31:     $NQ \leftarrow \emptyset$
32:     $totlen = allreduce(size(CQ))$
33: end while
```

vertex s, the BFS starts locally on the computing node in charge of the root and propagates to other computing nodes as the NLFS expands through the graph. Computing nodes with vertices in the NLFS perform a local frontier advancement, exchange information about other vertices with the corresponding owner nodes and update parents, if needed.

The predecessors array update and the parallel enqueue operation (lines 13-15 and 24-26) have to be carefully implemented: the first gives rise to a benign race condition, that is a race condition that cannot produce a harmful result, whereas the latter requires special care to ensure correctness and to achieve good performance.

Note that on line 18 both the vertex v_j and its parent u_i are appended to the *sendarray* so that each computing node can build the predecessors array. This solution causes communication overhead since two vertices are exchanged for each new vertex visited. Furthermore, if v_j is visited again in another iteration, it is sent again because the originating computing node does not keep track of visited vertices it does not own.

1.1.4 Related work

On a single GPU, the main problem running algorithms like the BFS is to find the best data-to-threads mapping so that the data-parallelism capabilities of the GPU may be exploited at their best.

Another issue is the amount of global memory available on the GPU, which on latest NVIDIA K80 GPU is limited to 24 GBytes divided into 12 GBytes memory pools (whereas commodity CPUs can have hundreds of GBytes); this contraints the size of the graph that can be visited.

Furthermore the naive assignment, in which each thread is assigned to one element of the BFS queue, may determine a dramatic imbalance and poor performance [7]. It is also possible to assign one thread to each vertex of the graph but, as showed by Harish *et al.* [24], the overhead of having a large number of unused threads results in poor performance. To solve this problem Hong *et al.* [17] proposed a warp centric programming model. In their implementation each warp is responsible for a subset of the vertices in the BFS queue. Another solution has been proposed by Merrill *et al.* in [28] where a chunk of data is assigned to a CTA (a CUDA block). The CTA works in parallel to inspect the vertices in its chunk.

While we inherit an original data mapping described by Bernaschi and Mastrostefano in [7], to reduce the work, we use an integer map to keep track of visited vertices.

Many papers on distributed systems rely on a linear algebra based representation of graph algorithms [6, 20, 21, 25]. That approach, along with a 2-D partitioning of the graph among computational nodes, reduces the overall communication time by limiting the number of nodes involved in the communication.

Satish *et al.* [29] implemented independently a technique to exchange predecessors very similar to our method presented in Section 1.2.4.1. They also implemented a bitmap to exchange vertices during the BFS search that reduces the amount of exchanged data.

Several authors ([5,21,29,30]) propose to use compression during communication to achieve higher performance. We reduce the amount of exchanged data as described in section 1.2.3.2. We ran some tests by compressing the remaining exchanged data but the compression ratio is so low (not surprising since we eliminated any data redundancy) that the time required for data compression/ decompression is greater than the time saved by exchanging compressed data.

Ueno *et al.* [30] presented a hybrid CPU-GPU implementation of the Graph500 benchmark, using the 2-D partitioning proposed in [4]. Their implementation uses the technique introduced by Merrill *et al.* [18] to create the edge frontier and resort to a novel compression technique to shrink the size of messages. They also implemented a sophisticated method to overlap communication and computation in order to reduce the working memory size of the GPUs.

A completely different algorithm that uses a direction-optimizing approach has been proposed by Beamer *et. al* [19] and extended in [25] for a cluster of CPUs. The direction-optimizing method switches between the *top-down* and the *bottom-up* traversal. The *bottom-up* search dramatically reduces the number of traversed edges during the most expensive computational levels of the BFS by searching the parent in the frontier starting from the sub-set of unvisited vertices. The predecessor search implies a serialization in order to minimize the required work. However, on shared memory systems (including single GPUs [31, 32]) the BFS performance increases significantly.

A recent work [33] demonstrates the chance of having an effective implementation of a distributed direction-optimizing approach on the BlueGene/P by using a 1-D partitioning. That partitioning simplifies the parallelization of the bottom-up algorithm but it may require a significant increase in the number of communications. Their results show that the combination of the underlaying architecture and the SPI interface is well suited to the purpose. The authors report that replacing the SPI with MPI incurs a loss of performance by a factor of nearly 5 although the MPI-based implementation cannot be considered optimal. This suggests that the scalability of a distributed implementation may be worse on different network architectures.

1.2 BFS on a Multi-GPU system

Our work strictly follows the Graph 500 [34] benchmark specifications: hereafter, when necessary to explain our choices, we describe some of the features and restrictions imposed by the benchmark but, for the full specifications, we refer to the Graph 500 website `www.graph500.org`.

The benchmark requires to generate in advance a list of edges with a R-MAT graph [35] generator. Then the actual benchmark consists of two parts: *i)* Kernel1 corresponding to the generation of the data structure representing the graph; *ii)* Kernel2 corresponding to the distributed BFS on the graph.

To double check the result of our algorithm, we resort to the same validation function provided with the reference code of the Graph 500. The validation ensures that: *i)* the BFS is a tree and does not contain cycles; each tree edge connects vertices *ii)* whose BFS levels differ by exactly one or *iii)* both vertices are out of the BFS tree; *iv)* the BFS tree spans all vertices of an entire connected component, and *v)* a vertex and its parent are joined by an edge of the original graph.

1.2.1 Graph generation and data structure

To be compliant with the Graph 500 specs, each vertex of the graph is represented by a 64-bit integer. On GPUs, where memory is a limited resource, this requirement imposes a severe limitation. More in detail, we generate $N = 2^{SCALE}$ vertices and $M = EF \times N$ edges, where EF is the edge factor and is equal to 16; now, considering that a single NVIDIA GPU is equipped with a maximum of 12 GBytes of *global* (*i.e.,* main) memory, the theoretical upper limit $SCALE$ would be 24. However since additional data structures are needed by the algorithm, the highest $SCALE$ of the Graph 500 benchmark that we are able to run on each device is 21.

Vertices are partitioned across computing nodes using 1-D partitioning; it consists in dividing the graph among computational nodes according to the following rule: edge $(u_i, v_j) \in P_k$ if $(u_i \bmod \#P == k)$, where $\#P$ is the number of nodes. With this decomposition each computational node holds a vertex together with all its outgoing edges.

The data structure is created directly on the GPU. We use the well known Compressed Sparse Row (CSR) data structure to represent the graph because it is simple and has reduced memory requirements. The CSR data structure is composed by two arrays, an array of offsets (*Offset*) and an array (*Adjacency Lists*) that contains the adjacency list of all the vertices in the graph (see Figure 1.1).

1.2.2 A first BFS implementation on Multi-GPU

Since our BFS implementation is grounded on the solution described by Bernaschi and Mastrostefano in [7], here we quickly review it. They follow the Algorithm 2, but provide a specific solution for building the NLFS and the next level queue.

To solve the imbalance of the workload among threads on a single GPU, they assume that a good solution is to have as many active threads as the number of elements in the NLFS so that each thread is in charge of only one

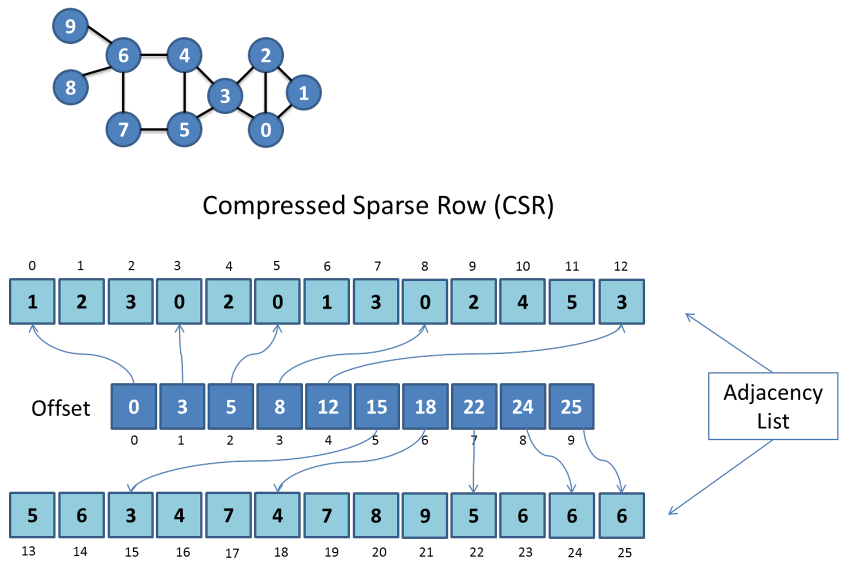


Figure 1.1. Compressed Sparse Row data structure for a toy graph with 10 vertices and 13 bidirectional edges. To obtain the adjacency list of vertex i, one looks up the entry i of the *Offset* array which contains the starting index in the *Adjacency List* array. E.g, for vertex 4 the adjacency list starts at offset 12.

vertex and the whole NLFS can be processed in parallel.

This data-thread mapping technique employs a prefix-sum operation and a binary search: given the current queue array $CQ[] = [u_0, u_1, \dots, u_{n-1}]$ and their degrees $degree[] = [d_0, d_1, \dots, d_{n-1}]$ the first step consists in using a prefix sum to calculate the cumulative degree array:

$$cumulDegree[] = [0, d_0, (d_0 + d_1), (d_0 + d_1 + d_2), \dots, (d_0 + d_1 + d_2 + \cdots + d_{n-2} + d_{n-1})]$$

In this way they determine m, the total number of elements in the NLFS, which corresponds to the threads required to process in parallel all the neighbors of vertices in CQ. This value is stored in the last element of $cumulDegree$ array.

In order to assign each thread to one NLFS vertex and its corresponding parent, a binary search is executed over the cumulative degree array $cumulDegree$: each thread uses its global thread identifier $threadId$ (which is a number between 0 and $m-1$), and a binary search operation to determine which is the index i in the *cumulDegree* array such that

$$cumulDegree[i] \leq threadId < cumulDegree[i+1]$$

so the thread identified by *threadId* is in charge of processing one of the neighbor of vertex $u_i = CQ[i]$. To determine which is the specific neighbor vertex v_j assigned to the thread, the following formula is used:

$$v_j = adj[offset[u_i] + threadId - cumulDegree[i]]$$

where *adj* and $offset$ are the CSR *Adjacency List* and *Offset* arrays respectively. Note that $adj[offset[u_i]]$ corresponds to the first neighbor of vertex u_i, while $threadId - cumulDegree[i]$ computes which is the specific neighbor assigned to the thread. In this way each thread processes a specific edge (u_i, v_j). Figure 1.2 shows how this data-thread mapping technique works for a toy graph.

Since NLFS is explored in parallel, the same vertex may be enqueued to NQ multiple times, this occurs when that vertex is connected to more vertices in CQ. To remove vertex duplicates, a sort followed by a compact-unique operation is performed. This solution can be very computing intensive when the NQ is very large. In Section 1.2.3 we will show how our enhanced solution solves this issue.

1.2.2.1 Results of the first BFS implementation

In figure 1.3 the sort-unique algorithm proposed in [7] is compared with the reference Multi-CPU implementation. The results were obtained on the CINECA PLX cluster that features 2 six-cores Intel Westmere 2.40 GHz and two Tesla M2070 Fermi GPU for each computational node (with a total of 274 nodes and 548 GPUS). Nodes are connected by QDR Infiniband.

The algorithm and results described so far were presented in [7], and are reported here to provide a comprehensive picture of our enhancements

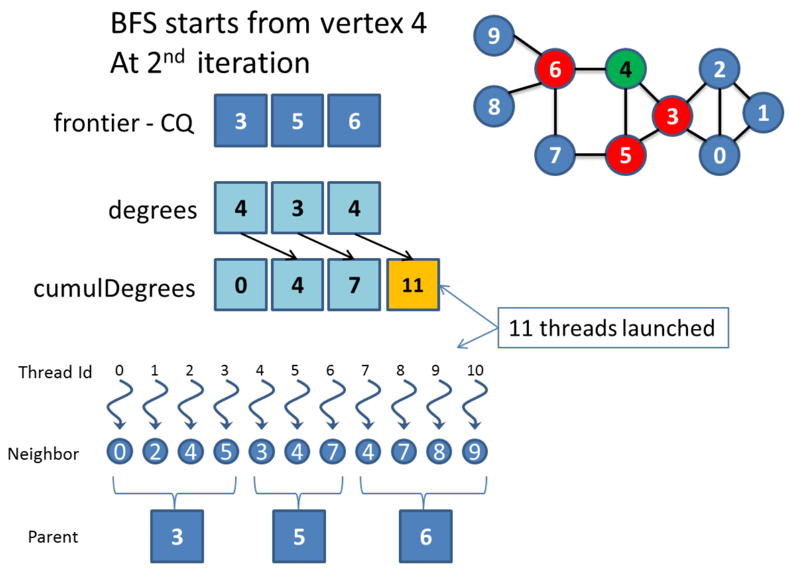


Figure 1.2. Example of data-thread mapping technique applied to a toy graph. The BFS starts from vertex 4 and the frontier CQ refers to the second iteration.

and improvements with respect to that work. Hereafter we describe these improvements and the new results we obtained.

1.2.3 Enhanced BFS

In the previous section we described how a combination of sort and unique operations allows to reduce data exchange by pruning the NLFS. That procedure, however, is very time consuming. In the following sections we describe a new approach to reduce both computation and communication.

1.2.3.1 Marking visited vertices using modified CSR data structure

In a distributed memory system, the graph is partitioned among computing nodes so that each node holds only a subset of the overall vertices and edges. For instance, using the 1-D partitioning described in Section 1.2.1 each node stores a CSR with $N \div \#P$ vertices (where N is the total number of vertices

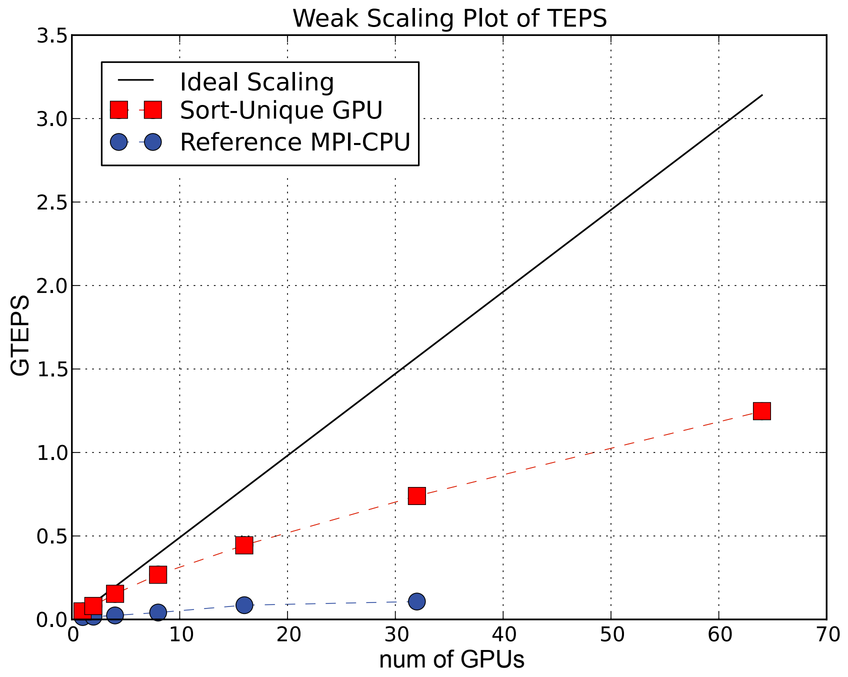


Figure 1.3. Performance of the sort-unique algorithm. We report the harmonic mean of the Traversed Edges Per Seconds (TEPS) over 64 runs.

in the graph and $\#P$ is the number of computing nodes in the distributed memory system), while in the *Adjacency List* array, each node stores all the vertices connected to the local vertices.

We observed that each computing node only deals with a subset of overall vertices, namely, local and remote vertices that are stored in its *Adjacency List* array. Therefore, it appears sufficient to allocate, on each computing node, a mask array capable to keep track of those vertices only. Building such data structure is not simple since: vertices stored on one computing node do not form a contiguous set of labels, and, the same vertices may appear in many adjacency lists.

To solve those issues and build a mask to keep track of visited vertices, we need to set up a contiguous set of labels, without multiplicities, to represent all vertices that are in the *Adjacency List*, regardless if they are local or remote. For that purpose, the CSR data structure is extended with an additional array as depicted in Figure 1.4.

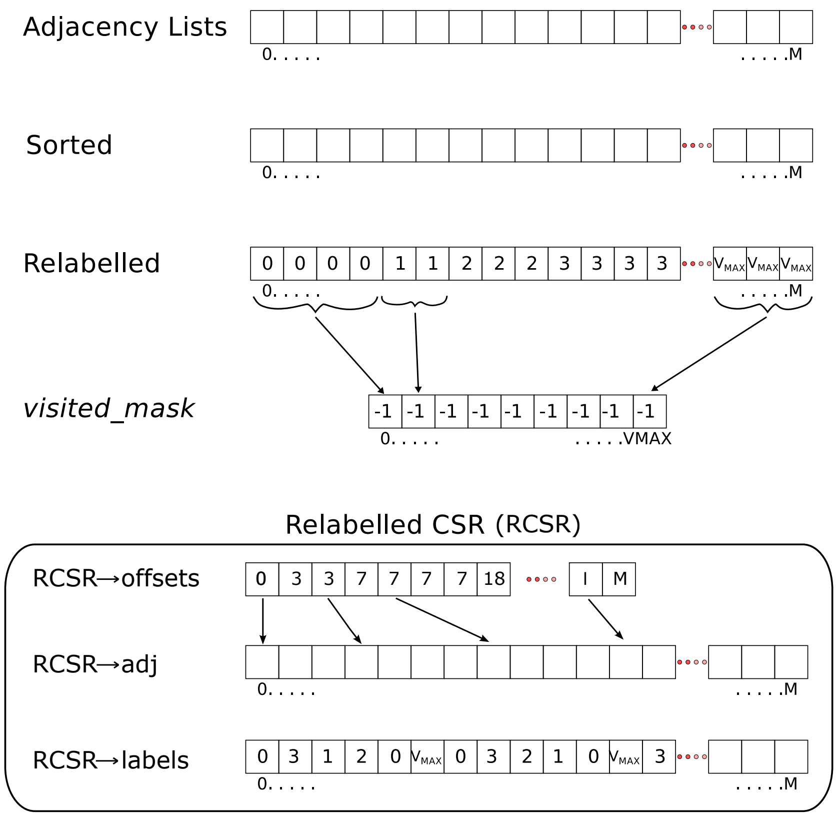


Figure 1.4. Relabeling of the CSR data structure that allows for the building of an ordered *local mask* of visited vertices

The resulting data structure, Relabeled-CSR (RCSR), is composed by three arrays: the array of offsets (*RCSR*→*offsets*), the array of adjacency lists (*RCSR*→*adj*), and the relabeled array of edges (*RCSR*→*labels*); note that the first two arrays corresponds exactly to the original *Offset* and *Adjacency List* arrays defined in the CSR structure.

Each element of the *RCSR*→*adj* array corresponds to one element of the *RCSR*→*labels* array but, while the first stores the original vertex identifiers, the latter stores a label or index that is a number between 0 and $V_{MAX} - 1$, where V_{MAX} counts how many unique vertices are stored in the CSR. The *RCSR*→*labels* array is used to access a *visited_mask* array of $V_{MAX} - 1$

elements.

1.2.3.1.1 Building the modified CSR data structure

To build the RCSR data structure, we start from the *Adjacency List* array of the CSR (see Figure 1.4).

First we sort the *Adjacency List* array in ascending order, then we assign a label to each unique vertex in the array (*Relabeled* array in Figure 1.4). The labels are consecutive numbers from 0 to $V_{MAX} - 1$, where V_{MAX} corresponds to the total count of unique vertices. Finally we build the *RCSR→labels* by sorting back the *Relabeled* array to the original ordering, that is the same order of the *Adjacency List* array from which we started.

At this point we can use a *visited_mask* array with V_{MAX} elements for keeping track of the status (visited or not) of all vertices in the local adjacency lists, indeed using the *RCSR→labels* array we can access the corresponding element in the *visited_mask* array.

1.2.3.2 Optimization of data communication

A careful analysis of the algorithm described in Section 1.2.2, shows that a newly discovered vertex v_l, that is local, is added to NQ only if it is unvisited, whereas a newly discovered vertex v_r, that is remote, is sent to the destination computing node regardless if it has been already visited or not because the computing node originating v_r does not store information on discovered vertices that are remote. In this way the same vertex v_r may be sent multiple times from the same computing node to its owner node; in the worst case, this can happen once for each BFS level.

Using the RCSR data structure previously defined, it is possible to remove this redundancy, indeed the *visited_mask* array is used to keep track of all visited vertices in the adjacency lists, regardless if they are local or remote, thus a computing node sends the same vertex only at most once during the whole BFS. This solution does not prevent the same vertex from being received multiple times by the owner, since it may be discovered by different computing nodes and each of them will send it. If N_r is the mean number of remote vertices included in an edge list (i.e., the number of remote vertices connected with local vertices), and S is the mean number of computing nodes that owns those vertices, than the total number of vertices received by a single node will be $S \times N_r$.

This first improvement has been implemented modifying the solution described in Section 1.2.2. In order to assign each thread to one vertex in the NLFS and its corresponding parent, a binary search is executed over the cumulative degree array *cumulDegree* and the index i is calculated, then *RCSR→offset* is used to find the index v_{index} of the *RCSR→adj* to get the value of the neighbor v_j as described in the following code snippet:

$$i = binsearch(cumulDegree, threadId, m);$$
$$t_{off} = threadId - cumulDegree[i];$$
$$u_i = CQ[i]$$
$$v_{index} = RCSR- > offset[u_i] + t_{off}$$
$$v_j = RCSR- > adj[v_{index}]$$

where $threadId$ is the global thread identification index, m is the number of elements in the $cumulDegree$ array, v_j and u_i are respectively the neighbor vertex and its parent. The *binsearch* function returns the index of the entry in the $cumulDegree$ array whose value is lower, or equal, to $threadId$.

At this point we need to determine if the neighbor vertex v_j has already been visited (regardless if it is local or remote). To that purpose, instead of checking if v_j is local and if it has already been visited (lines 12-19 of Algorithm 2), we use the RCSR data structure: the value stored in *RCSR→labels*[v_j] corresponds to the index in *visited_mask* array, that needs to be checked-and-set. Furthermore, if we store in the *visited_mask* array a flag that indicates if the owner of v_j is local or remote, this can be used to implement a new communication pattern to exchange vertex predecessors, as described in Section 1.2.4.1.

This solution prevents adding the same vertex (local or remote) multiple times to NQ and also that the same remote vertex is transmitted multiple times to its destination computing node. As a consequence, there is no more the need for pruning the NLFS array by using the sort-unique operation as described in Section 1.2.2.

1.2.4 Other optimizations

1.2.4.1 Predecessor vertices

In Algorithm 2 we already highlighted that on line 18 both the vertex v_j and its parent u_i are appended to the *sendarray* so that each computing node can build the predecessors array. This solution causes communication overhead since two vertices are exchanged for each new vertex visited.

In this section we present a solution to exchange predecessors only after the graph traversal terminates, thus reducing communication costs during the search by (almost) a factor of two. This enhancement requires modifying the algorithm in two different places: within each BFS iteration and at the end of BFS. More in detail:

1. within each BFS iteration:

    a) when a process sends a vertex v_r, it stores the predecessor u locally in the *visited_mask* array

    b) when a process receives vertex v_r from process P_{id}, if vertex v_r has not been visited yet ($pred[v_r] == -1$), then the identifier of the sending process is stored into the *pred* array ($pred[v_r] = P_{id}$);

    where *pred* is the predecessors array. Note that the process identifier P_{id} is stored because, later on, each process needs to recover predecessors information asking to the process which first sent v_r.

2. in the end of BFS, each process:

a) sends the list of its own vertices for which predecessor's data are needed: this is done by looking into *pred* array where information on process identifier is stored;

b) receives vertices for which it is asked to provide predecessor's data: for each vertex, its predecessor is retrieved by using the *visited_mask* array, those predecessors are then sent back to requester processes;

c) each requester receives the predecessors and updates *pred* array;

To reduce communication latency, vertices destined to a single remote process are aggregated locally, and are sent in one MPI send operation.

1.2.4.2 Exchange vertices by using 32-bits identifiers

According to Graph500 specs *"an implementation may use any set of N distinct integers to number the vertices, but at least 48 bits must be allocated per vertex number"*. Therefore although vertices are identified by using 64 bits integers, only up to 48 bits are used. In a distributed memory system, each process holds a portion of the full graph structure. We recall that we use the 1-D partitioning described in Section 1.2.1:

if $\#P$ is the number of processes involved and they are numbered from 0 to $\#P-1$, then $u_i \in P_k$ if $ui \bmod \#P == k$. This partitioning scheme allows to introduce a *local* representation or *local* index of the vertex $= u \div \#P$. In summary:

a) a vertex is identified by the 64-bits integer value u (called *global* representation or *global* index);

b) $P_k = (u \bmod \#P)$ is the process identifier that owns u;

c) $loc(u) = (u \div \#P)$ is the vertex *local* index.

With, at least, 16 processes the vertex *local* index will never exceed 32-bits. So, the local representation can be used for sending vertices among processes. This halves the amount of transmitted data. When required, it is simple to calculate back the vertex *global* representation.

Vertices are transmitted both during each BFS iteration and in the end of BFS. This optimization is applied in both cases. In detail, during each BFS iteration

1. when next level vertices are discovered, we determine both their owner processes (in order to send remote vertices to the owner processes), and their 32-bits *local* values;

2. 32-bits representations are then used to send/receive vertices;

3. when vertices are received, they are already in local representation and no specific action is required.

In the end of BFS, vertices are exchanged by using the same logic. However, before updating the *pred* array, it is necessary to recover the global 64-bits representation.

1.2.5 Performance analysis

Weak scaling experiments results shown in Figures 1.5 and 1.6 were obtained on R-MAT graphs with edge factor (EF) 16 and scale from 21 (for 1 GPU) up to 31 (for 1024 GPUs); results are in GTEPS (10^9 Traversed Edges Per Second).

The results indicated as “Mask K20X Aries” were obtained on the “Piz-Daint” Cray XC30 of the Centro Svizzero di Calcolo Scientifico (CSCS), equipped with NVIDIA K20x GPUs interconnected by a Cray custom high performance network (Aries routing and communications ASIC with Dragonfly network topology).

The results indicated as “Mask K20X Gemini” were obtained on the “Todi” Cray XK7 of the Centro Svizzero di Calcolo Scientifico (CSCS), equipped with NVIDIA K20x GPUs connected by the previous generation Cray network (Gemini).

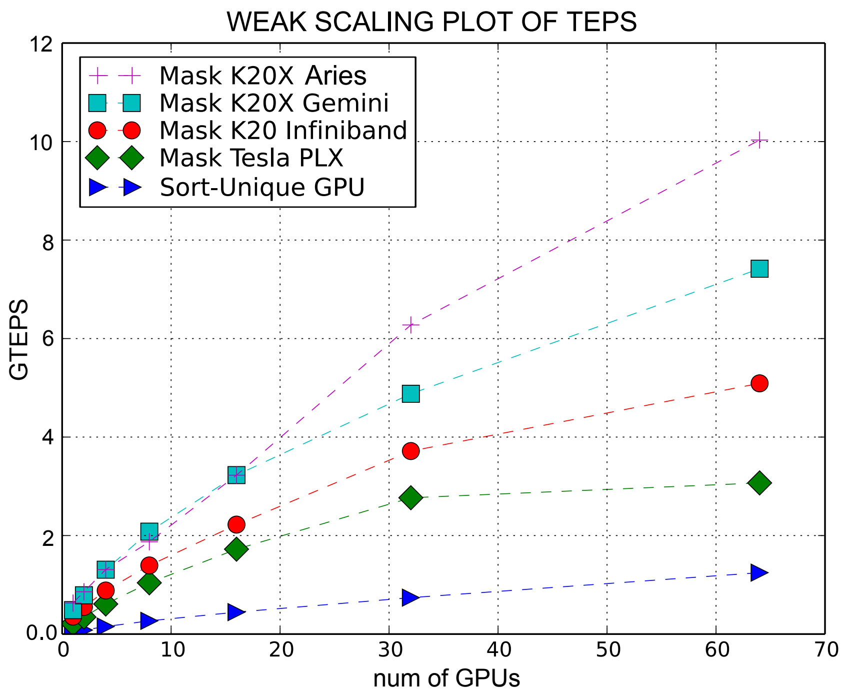


Figure 1.5. Performance of the enhanced BFS solution. We report the harmonic mean of the TEPS over 64 runs.

The results indicated as "Mask K20 Infiniband" were obtained on a cluster of the University of Southern California with 264 nodes equipped with two NVIDIA K20 GPUs while nodes are connected by FDR Infiniband.

The results labeled as "Mask Tesla PLX" were obtained on the same system where we ran the first BFS solution based on "Sort-Unique" (results already presented in Section 1.2.2.1) and are reported here for comparison.

It is apparent how the combination of our enhancements, new GPUs and interconnection technology provides a dramatic advantage with respect to the original "Sort-Unique" based implementation. On 64 GPUs the performance increase by a factor 4 although we did not use any specific new feature of the Kepler GPUs. Comparing the "Sort-Unique" series with the "Mask Tesla PLX" series we can notice the performance improvement gained only with the our enhancements (the hardware platform is the same) whereas the difference

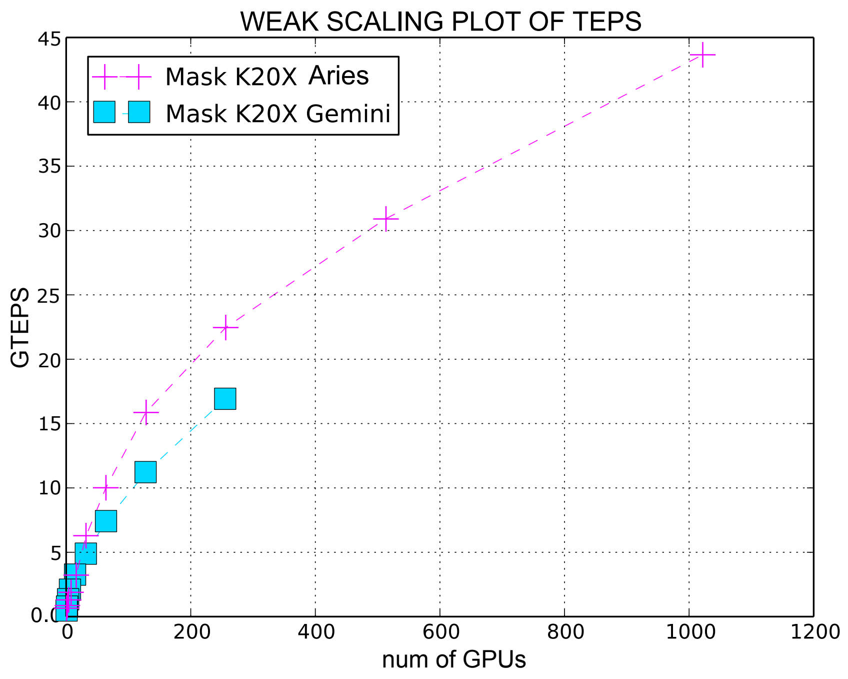


Figure 1.6. Performance of the enhanced BFS solution. We report the harmonic mean of the TEPS over 64 runs.

between the "Mask K20 Infiniband" and the "Mask Tesla PLX" is due to the different technology (Kepler *vs.* Fermi and FDR *vs.* QDR Infiniband) since the implementation is the same. The main difference between "Mask K20X Gemini" and "Mask K20X Daint" is the interconnection network. This result shows clearly the impact of a better communication technology.

1.2.5.1 Real Graph

We also applied our BFS algorithm to some real-world graphs obtained from the Stanford Large Network Dataset Collection [36]. Among them we selected undirected graphs with the highest number of edges and nodes and Twitter graph [37].

Results shown in Table 1.1 were obtained on the CSCS "Piz-Daint" system: the first column reports the name of the data set; the second and third columns are the scale and edge factor. Last two columns report the result obtained in terms of GTEPS and how many iterations are required to explore the whole graph.

Data Set Name	Scale	EF	GPUs	GTEPS	Levels
com-LiveJournal	~ 22	~ 9	2	0.43	14
soc-LiveJournal1	~ 22	~ 14	2	0.47	13
com-Orkut	~ 22	~ 38	4	1.43	8
com-Friendster	~ 25	~ 27	64	5.55	24
Twitter-2010	~ 25	~ 35	64	5.83	16

Table 1.1. Results of BFS solution on real graphs.

Our solution provides good performance even on the com-Friendster data set where we achieve 5.5 GTEPS on 64 GPUs, although 24 iterations are required (while R-MAT generated graphs require 6-8 iterations). This confirms that GPUs can be a valid platform for the analysis of large-scale graphs having practical relevance.

1.3 ST-Connectivity problem

ST-Connectivity (ST-CON) is the problem of deciding whether there is a path between vertices s and t in a graph. Undirected ST-CON is the version of ST-CON where the graph is undirected and it is known as U-STCON.

The BFS solution discussed in Section 1.2 can be clearly used to solve the U-STCON problem: we can determine if there is a path from vertex s to t by starting a BFS from s and terminating it when t is visited or the entire graph is visited. Moreover the path found is a shortest path.

A different strategy would be to run two BFS concurrently, starting from both vertices s and t and expanding their frontiers. The algorithm ends when the same vertex is in both frontiers. Vertices on both frontiers represent the ST-CON Matching Set (ST-CON MS).

Using the BFS algorithm in this way to solve the ST-CON problem poses the following main issues:

- in a BFS there is a single frontier, whereas in ST-CON there are two frontiers: one of the vertices from s and another of the vertices from t. This doubling increases the memory requirements;

- in a BFS, a vertex can be either unvisited or visited, whereas in ST-CON, a vertex can be unvisited, visited from s, visited from t, or visited from both s and t (in which case, a matching vertex is found and the problem is solved);

- in a BFS, a vertex can have at most one predecessor, whereas in ST-CON, vertices within the matching set have two predecessors, one from the s path and another from the t path.

1.3.1 Solution to ST-CON on Multi-GPU

In order to cope with two frontiers, when building the NLFS, vertices coming from s are colored RED whereas those coming from t are colored BLUE. The *coloring* can be done by using a color mask to set the second Most Significant Bit (MSB)[b] of the vertex to 1 for BLUE vertices and 0 for RED ones.

To keep track of vertices visited from s and t, we use RED and BLUE colors both in the predecessor array *pred* and in the *visited_mask* array. A matching vertex can be found either when NLFS is expanded locally or when the remote vertices are received. They are found *locally* if the new vertex has been already visited from a parent with a different color; they are found *remotely*, if the vertex received has been already visited from a parent with a different color. In either cases, a matching vertex is found and we store it and its two predecessors.

[b]We use the second MSB because the first MSB is already used to mark a vertex as visited.

Algorithm 3 Parallel ST-CON with atomic operations

```
    Input: graph G(V,E), $s$, $t$
    Output: matching node $v$, its parents $u$ and $w$, path $s$ to $t$
    Data: $CQ$ and $NQ$; $pred$
    Macro: $GetColor(u)$ get the color of $u$, $SetColor(c,u)$ set the color $c$ to $u$
 1: $CQ, NQ \leftarrow \emptyset$
 2: $pred[v_j] = -1, \forall\ v_j \in V$
 3: $enqueue(CQ, SetColor(Red, s), SetColor(Blue, t)$
 4: $pred[s] = s$; $pred[t] = t$
 5: while $CQ \neq \emptyset$ do
 6:     for all $u_i$ in $CQ$ in parallel do
 7:         for all $v_j$ neighbor of $u_i$ (in parallel) do
 8:             ——critical section——
 9:                 if $pred[v_j] == -1$ then
10:                     $pred[v_j] = u_i$
11:                     $MyColor = GetColor(u_i)$
12:                     $enqueue(NQ, SetColor(MyColor, v_j))$
13:                 else if $GetColor(pred[v_j])! = GetColor(u_i)$ then
14:                     return $v_j, u_i, w_i$
15:                 end if
16:             end ——critical section——
17:         end for
18:     end for
19: end while
```

In Algorithm 3 we present the pseudo-code for the ST-CON. For the sake of simplicity, the description is based on the original Level-synchronous BFS (Algorithm 1), although the actual implementation uses the enhancements discussed in Section 1.2

At the beginning of the algorithm, both s and t are enqueued in the same queue, the first colored RED and the second BLUE, and their predecessors are set (lines 3-4). The frontier is then expanded and each new vertex, for which the predecessor is not set (line 9), is enqueued with the color of its parent (lines 11-12). If a vertex has already been visited from another color, then the algorithm ends, returning the matching vertex along with its parents (lines 13-14).

The whole section between lines 8-16 is critical for concurrency. A straightforward way to maintain the coherence is based on the usage of atomic operations. More precisely, it can be implemented through the use of a compare and swap operation that resolves the race condition among threads accessing the predecessor array (we refer to this implementation as *atomic-stcon*). For the simple BFS, this race condition is benign because the predecessors are *idempotent.*

Algorithm 4 Parallel ST-CON without atomic operations

```
    **Input**: graph G(V,E), $s$, $t$
    **Output**: Matching Node Set (MNS), each path from $s$ to $t$
    **Data**: $CQ$ and $NQ$; $pred_s$, $pred_t$
    **Macro**: $GetColor(u)$ get the color of $u$, $SetColor(c, u)$ set the color $c$ to $u$, $UnSetColor(u)$
    return $u$ discolored
 1: $CQ, NQ, MNS \leftarrow \emptyset$
 2: $pred_s[v_j] = pred_t[v_j] = -1, \forall\ v_j \in V$
 3: $enqueue(CQ, SetColor(Red, s), SetColor(Blue, t))$
 4: $pred_s[s] = s\ pred_t[t] = t$
 5: while $CQ \neq\ \emptyset\ \ \&\ \ MNS == \emptyset$ do
 6:     for all $u_i$ in $CQ$ in parallel do
 7:         for all $v_j$ neighbor of $u_i$ (in parallel) do
 8:             if $GetColor(u_i) == RED$ then
 9:                 if $pred_s[v_j] == -1$ then
10:                     $pred_s[v_j] = UnSetColor(u_i)$
11:                     $enqueue(NQ, SetColor(Red, v_j))$
12:                 end if
13:             else if $GetColor(u_i) == BLUE$ then
14:                 if $pred_t[v_j] == -1$ then
15:                     $pred_t[v_j] = UnSetColor(u_i)$
16:                     $enqueue(NQ, SetColor(Blue, v_j))$
17:                 end if
18:             end if
19:         end for
20:     end for
21:     for all $v_i \in V$ in parallel do                    ▷ Find matching node
22:         if $pred_s[v_i] \neq -1\ \ \&\ \ pred_s[v_i] == pred_t[v_i]$ then
23:             $enqueue(MNS, v_i)$
24:         end if
25:     end for
26:     $CQ \leftarrow NQ$
27:     $NQ \leftarrow \emptyset$
28: end while
```

The need to control access to the critical section can be avoided by removing the race condition. To that purpose, it is necessary to use distinct memory locations to store the predecessors and visited vertices for the two subset of vertices with additional usage of the GPU memory. We refer to this implementation as *no-atomic-stcon* (see Algorithm 4).

More in detail, to keep track of vertices visited from s and t, we use different arrays. Predecessors and visited vertices from s are stored in $pred_s$ and $mask_s$, respectively, whereas predecessors and visited vertices from t are stored in $pred_t$ and $mask_t$, respectively. Filtering already-seen vertices depends on the vertex color (lines 8-17) and thus the critical section is removed.

At the end of each BFS level (lines 21-25), we compare $pred_s$ and $pred_t$ to determine whether a matching vertex has been found. In this way, we also calculate all vertices in the matching set and all paths between s and t.

1.3.2 How to evaluate ST-CON performance

Most of recent work uses the TEPS metric to evaluate and compare BFS performance. Papers dealing with different graph algorithms such as Betweenness Centrality or All-Pairs Shortest-Path still use the running time [24, 38–40].

There is, at least, one major drawback to using the TEPS metric to evaluate the performance of a solution to the ST-Connectivity problem and, more generally, for other graph algorithms (as far as we know TEPS has been used only for BFS). The problem is that it counts all the edges in the connected component that includes the starting vertex (root) in addition to those actually traversed by the algorithm. By doing this the TEPS metric does not account for the actual work done.

We argue that for ST-CON, a simple but effective metric can be represented by the mean value of the number of s-t paths (NSTPS) found in one second, averaged over a suitable set of extracted pairs:

$$< NSTPS >_{NE}= \frac{1}{< \text{s-t } time >_{NE}}$$

where NE is the Number of Extracted pairs. The number NE and set of s-t pairs must be selected carefully.

For the special case of R-MAT graphs, it turns out that it is possible to choose at random a relatively small set compared to the total number of nodes in the graph ($NE << N$). This is a consequence of two properties of a R-MAT graphs: the power law distribution and small diameter. Such properties give rise to a sharp distribution of the shortest path lengths. For example, we computed the mean and variance of the shortest path lengths found by extracting 100, 1000 and 10000 s-t pairs for different instances of an

R-MAT graph (mean and variance were computed only for connected pairs). The results are shown in Table 1.2.

R-MAT GRAPHS								
S=25, NP=16, d=8			S=27, NP=64, d=9			S=29, NP=256, d=9		
NE	Mean	Variance	NE	Mean	Variance	NE	Mean	Variance
9989	1.9994	0.0280	9946	2.110	0.1006	9983	2.022	0.0311
999	2.002	0.0240	995	2.110	0.1024	999	2.024	0.037
100	2.01	0.01	100	2.090	0.0827	100	2.01	0.01

Table 1.2. Mean and variance of the length of shortest paths for three different instances of an R-MAT graph with different values of SCALE S. NP is the number of GPUs and d the diameter of the graph. For each instance, NE is the number of extracted vertices.

REAL GRAPHS					
Live-journal1 S=22, EF 14, d=15			com-Orkut S=22, EF 38, d=8		
NE	Mean	Variance	NE	Mean	Variance
9988	3.027	0.298	10000	2.361	0.240
1000	3.021	0.302	1000	2.365	0.244
100	2.84	0.297	100	2.25	0.209

Table 1.3. As in Table 1.2, we report the mean and variance of the length of the shortest paths for two real graphs.

It is apparent that the mean and variance are reasonably stable with respect to the number of extractions. By invoking the Law of Large Numbers, we can state that 1000 pairs are a representative set of the whole graph. We can repeat the same argument for the real graphs that we analyzed. In those cases, the variance is higher and thus we decided to increase the number of pairs to 10000 (although 1000 would suffice, see Table 1.3).

1.3.3 Performance analysis

We report performance evaluation for both *atomic-stcon* and *no-atomic-stcon* solutions for the ST-Connectivity problem on a Multi-GPU.

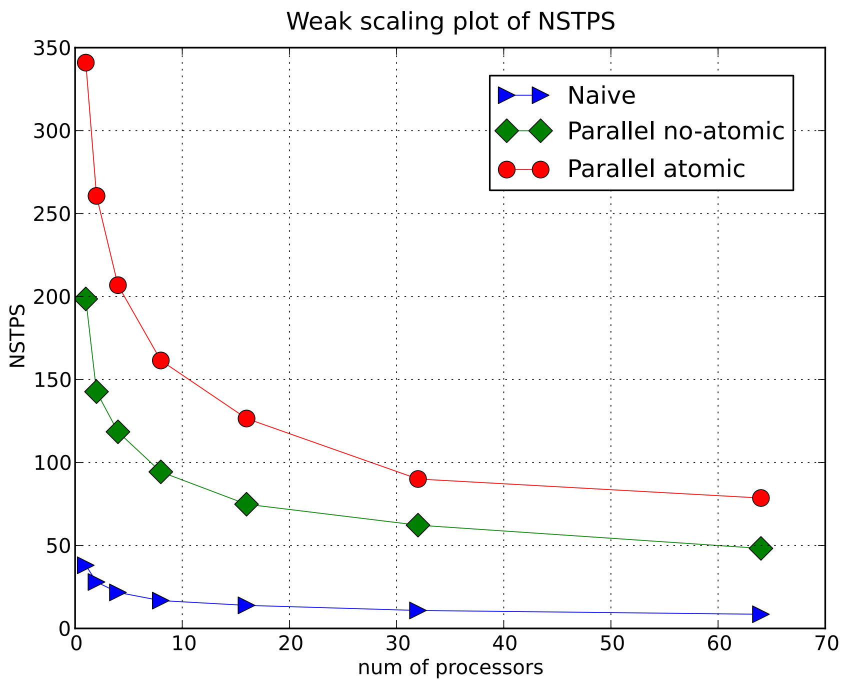


Figure 1.7. Weak scaling plot of the number of ST-CON problems solved within a second (*NSTPS*) for the two implementations described in the text: *atomic-stcon* and *no-atomic-stcon*. For comparison, we also show the performance of the naive (single BFS) implementation. The SCALE of the R-MAT graph ranges from $21 - 27$ for $1 - 64$ GPUs, respectively, with EF equal to 16.

Data Set Name	V naive	V atomic	V no-atomic	SCALE	EF	MV-lvl
R-MAT	31.59%	<1%	1.94%	22	16	1.98
RANDOM	81.77%	<1%	1.24%	22	16	2.76
soc-LiveJournal1	68.27%	< 1%	6.05%	$\sim$ 22	$\sim$ 14	3.0
com-Orkut	74.52%	<1%	9.75%	$\sim$ 22	$\sim$ 38	2.36

Table 1.4. Columns 2, 3, and 4 show, for different graphs, the percentage of vertices in the graph visited by the *naive*, *atomic*, and *no-atomic* implementations, respectively. Columns 4 and 5 specify the size of each graph in terms of SCALE and EF (for real datasets, SCALE and EF are approximated from the number of vertices and edges). Column 6 shows at which BFS level (MV-lvl) the matching vertex is found. The level is computed as the average with respect to 10000 random instances of the ST-CON problem for the same graph.

Figure 1.7 shows the performance, in NSTPS, of our implementations. For

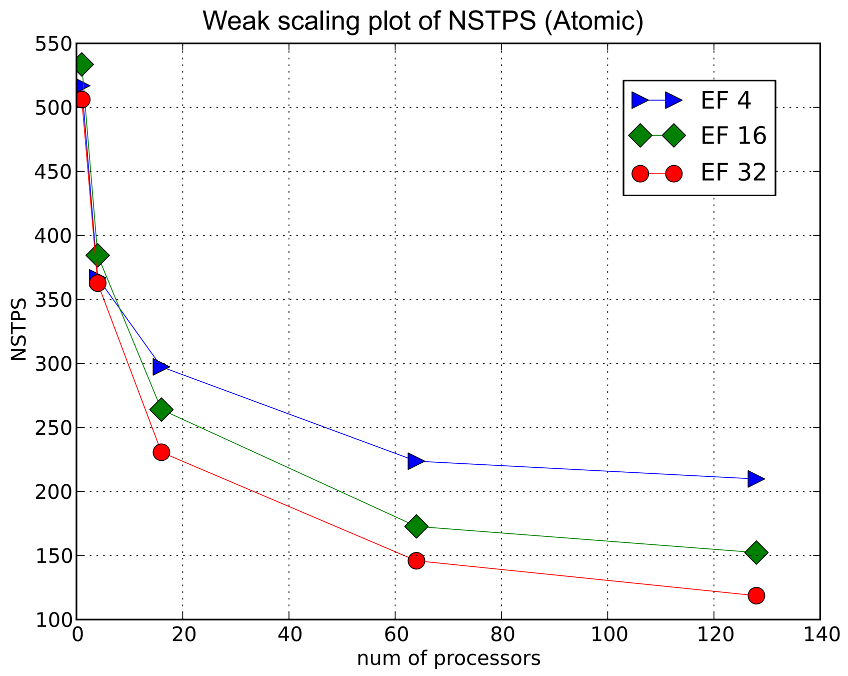


Figure 1.8. Weak scaling plot of *NSTPS* using the *atomic-stcon* implementation for three different values of EF. The SCALE of the R-MAT graph ranges from $19 - 26$ for $1 - 128$ GPUs, respectively.

comparison, the performance of a *naive* implementation is also plotted. The *naive* implementation simply starts a BFS from s and stops if t is reached. As expected, both the *atomic-stcon* and *no-atomic-stcon* outperform the *naive* implementation (Figure 1.7). The weak scaling plot is consistent with our basic intuition: the ST-Connectivity problem is harder when the scale of the graph is larger. Searching a path within a larger component using a BFS algorithm requires the traversal of more edges. This is also apparent in Figure 1.8, where by varying the EF parameter, we change the number of edges given a number of vertices. The code performs better when there are fewer edges to be traversed (the plot refers to the *atomic-stcon* implementation, but we obtained the same behavior for the *no-atomic-stcon* implementation).

It is apparent that the *atomic-stcon* implementation performs better than the *no-atomic-stcon*. This is mainly because the *atomic-stcon* implementa-

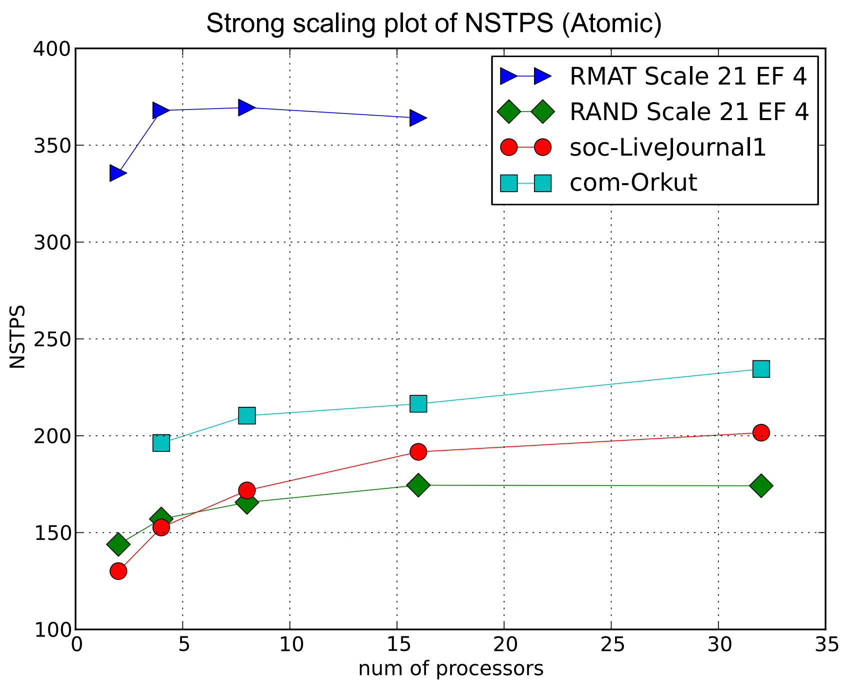


Figure 1.9. Strong scaling plot of *NSTPS* using the *atomic-stcon* implementation. 1

tion visits a very small fraction of the vertices in the graph, as shown in Table 1.4. Moreover, atomic primitives have been significantly improved in the latest Nvidia GPU "Kepler" ([30]).

A strong scaling plot of the *atomic-stcon* implementation is shown in Figure 1.9. We gain some benefits only by using a small number of computing nodes. This is because there is not enough work to be done in parallel, as becomes apparent by looking at Table 1.4. For the graphs under investigation, the matching vertex (MV-lvl) is found after, at most, three levels. At that level, hub vertices are usually enqueued but not yet visited [19, 28]. As a consequence, only a small fraction of vertices are actually visited (columns "V naive", "V atomic", and "V no-atomic"). This under-utilizes the CUDA threads and, in turn, explains the lack of scalability.

Our implementation can also output the path (or the set of paths) between *s* and *t*. This part has been implemented in a straightforward way by collecting all the predecessors on one computing node and then traversing the whole set backward from the matching vertex and its predecessors. We did not include the time needed for that task in the reported NSTPS.

One of the first papers to deal with a parallel implementation for the ST-Connectivity problem is [26]. The authors exploited the idea of starting two BFS concurrently from both s and t. The proposed solution relied on atomic operations and achieved a performance of 0.3s to solve an ST-CON problem on a scale-free graph with 134 million vertices and 805 million edges (EF $\sim$ 6) on a Cray MTA-2. On a problem of comparable size, an R-MAT graph with 134 million vertices and 4 billion edges (EF=16), our *atomic-stcon* implementation running on 64 GPUs is about 25 times faster (it takes 0.012s).

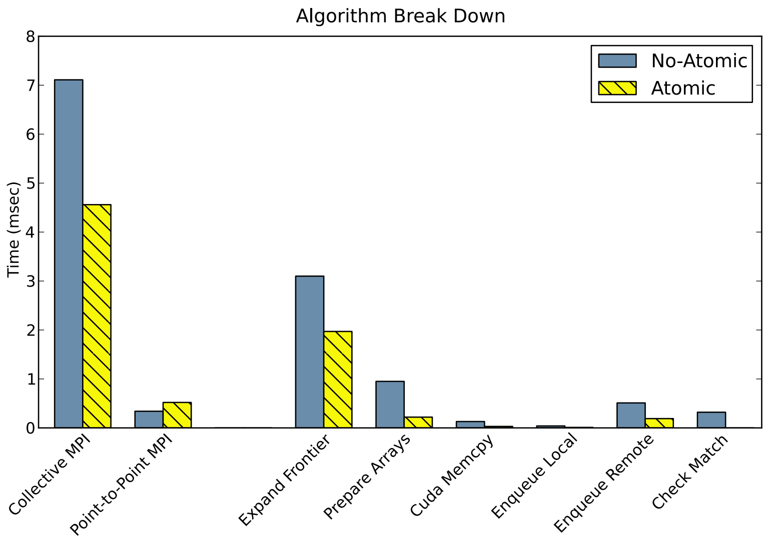


Figure 1.10. Time breakdown for *no-atomic-stcon* and *atomic-stcon* on 16 GPUs. The scale of the R-MAT graph is equal to 25, and EF is equal to 16.

Figure 1.10 shows a time breakdown of the main computational and communication components of the two implementations for a R-MAT graph, while in Figure 1.11, we compare the overall computation and communication time for three different graphs. Data were collected, using 16 GPUs

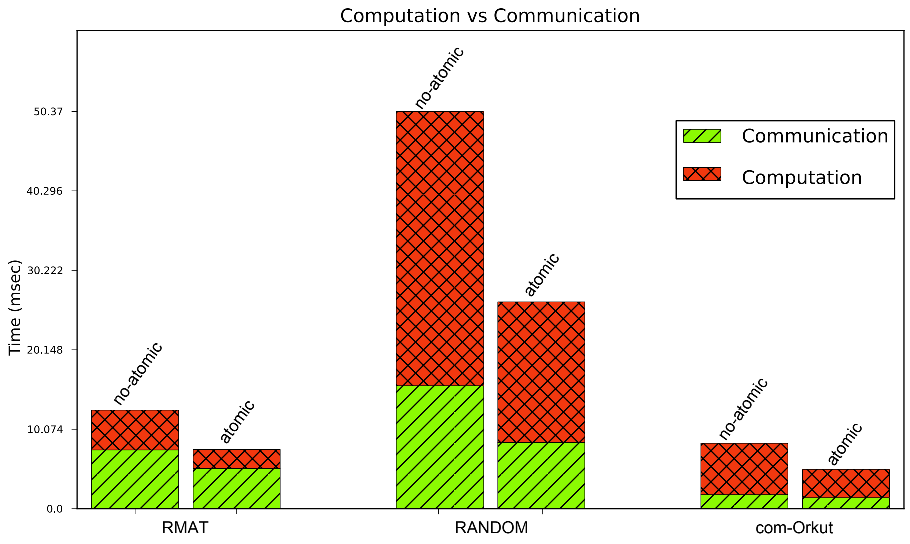


Figure 1.11. Computation and communication time breakdown for *no-atomic-stcon* and *atomic-stcon* for three graphs. The scale of the R-MAT and RANDOM graphs is equal to 25, and EF is equal to 16. The scale of the com-Orkut graph is approximately 22, and EF is approximately 38.

to solve 2000 s-t pairs for an R-MAT graph and a RANDOM graph with SCALE equal to 25 and EF equal to 16, and using only 4 GPUs for the *com-Orkut* real dataset, since it is smaller than the synthetic graphs. Each timing is averaged over the 2000 runs and over the number of GPUs.

We first discuss the computational parts: *expand frontier*, *prepare arrays*, *enqueue local*, *enqueue remote*, and *check match*. The expansion of the frontier, where the NLFS is built starting from the *Current Queue*, is the most time consuming part. The *atomic-stcon* implementation performs better because it stops exactly when the first matching vertex is found. Thus, on average, only a subset of the vertices in the frontier are actually visited. The *prepare array* part, which organizes data for communications, is more expensive in the *no-atomic-stcon* implementation, because of the redundancy of the data structure used. However, at this point of the BFS, there are fewer elements to be processed, because of the filtering in the previous step, and thus the time required by *prepare array* is only a small fraction of the time required by *expand frontier*. Once again, as the number of visited vertices is, on average, smaller in the *atomic-stcon* implementation, it is less expensive to use atomic primitives to build the queue. The *check matching* is present only in *no-atomic-stcon*, where the check for a matching node is performed right before starting a new iteration, whereas in *atomic-stcon*, this check is performed in both *expand frontier* and *enqueue remote*.

MPI communications can be further divided into collective and point-to-point primitives. The first group includes the `MPIallgather` and `MPIallreduce`, operations. Collective MPI operations require a significant amount of time because they are implicit synchronization points, that is, all processes wait for the slowest computing node. In the first BFS levels, computation may be unbalanced among computing nodes because queues contain relatively few elements not uniformly distributed among them. This unbalance occurs both in the ST-Connectivity and BFS algorithms, but the former terminates after very few levels (see Table 1.4), and therefore the unbalance is more evident.

1.4 Discussion

In this chapter we discussed the Breadth First Search algorithm and how it can be used to solve the ST-Connectivity problem. We have seen the Multi-GPU BFS solution described in [7] and we found how it can be enhanced. In particular we described the following main improvements:

1. a modified CSR data structure that, along with a mask array, allows to keep track of both local and remote vertices already visited, the use of this array greatly reduces the computation and the communication during the BFS;

2. a new communication pattern among computing nodes that allows to send predecessors only once at the end of the BFS;

3. exchanging 32-bits vertices instead of 64-bits, halving communication message size.

Our experiments show that the ratio between the time spent in computation and the time spent in communication reduces by increasing the number of tasks. When the graph size increases we use more GPUs and the number of messages exchanged among tasks increases accordingly. To maintain a good scalability by using, say, thousands GPUs we need to further improve the communication mechanism that remains, in the present implementation, quite simple. To that purpose, many studies propose a 2-D partitioning of the graph to reduce the number of computing nodes involved in communication [41].

After the Breadth First Search, we discussed the ST-Connectivity problem, in particular we presented two solutions to ST-CON. Both are based on concurrent BFS operations running on multiple GPUs. On a single GPU, we can solve, in one second, about 340 ST-CON problems on a graph having about 2 million vertices and 32 million edges using an implementation that resorts to atomic operations for the control of critical sections where data structures, shared by the concurrent BFS, need to be accessed. The efficiency of the atomic primitives available using "Kepler" NVIDIA GPUs is crucial from this viewpoint. Some tests we carried out on previous generation ("Fermi") GPUs show that atomic operations may have a dramatic impact on performance when running on multiple GPUs, up to the point that, on the Fermi architecture, the other solution we implemented, based on a duplication of some critical data structures, may provide better performance. In this situation, our choice of a Relabeled-CSR for the data structure limits the amount of additional memory required. In general, if the structure of the graph is such that ST-CON is solved within the first few (say three) levels of the BFS, the efficiency of a GPU implementation remains quite limited, because only few threads are in use on each GPU.

Chapter 2

Betweenness Centrality

One of the main goals of network analysis is finding out the most “relevant” nodes in a network according to a centrality measure. In general, centrality measures play an important role in several graph applications including transportation [42], wireless sensor networks [43], beyond the aforementioned social and biological networks [10, 44]. Regarding social networks, the centrality of a node (often representing an entity or an individual) states a degree of influence within a social domain. Several measures have been proposed in literature in order to assess the influence of a node in a network [45, 46].

In particular, one of the most popular metrics is the Betweenness Centrality (BC) [10]. The betweenness centrality of a node is based on the number of all-pairs shortest paths passing through that node. Despite of the simplicity of the definition, the computation of BC scores of all the nodes in a network is expensive. The fastest known algorithm for calculating BC scores has $\mathcal{O}(nm)$ time-complexity for unweighted graphs [14]. Serial implementation of Brandes' algorithm takes too long for graphs like Twitter with 41.7 million user profiles and 1.47 billion social relations [37]. Several authors [47,48] proposed alternative algorithms based on the parallelization of Brandes' algorithm for the exact computation of BC score. However, in those solutions the size of the graph is limited by the space complexity of Brandes' algorithm.

Since BC computation on unweighted graphs employs BFS, parallel and distributed BC implementations present similar issues. On the other hand, fast BFS implementations do not guarantee a fast BC computation due to Brandes' algorithm time-complexity.

Hereafter, we present two complementary solutions for the computation, on distributed-memory parallel systems, of Betweenness Centrality for unweighted graphs. We propose a fully distributed solution based on a two-dimensional (2-D) decomposition of the sparse adjacency matrix of the graph on Multi-GPU systems. To the best of our knowledge, this is the first fully distributed Betweenness Centrality implementation over a GPU cluster, since in previous implementations the graph was replicated on each GPU. Additionally, we also support graph replication over multiple GPUs, in other words we split the GPU cluster in sub-clusters of one or more GPUs, which work on the same graph, so that each sub-cluster can process a disjoint subset of all vertices independently.

Since computation of BC score can be really expensive, we apply graph topology manipulations to reduce time and space complexity of BC computation. In particular, our solution is enhanced by a distributed graph preprocessing task based on 1-degree reduction technique [49, 50]. We also provide an analysis of the impact of 1-degree reduction. Other contributions are:

- we describe an efficient algorithm for betweenness centrality computation that in most cases outperforms previous single GPU implementations by exploiting a threads-data mapping technique already introduced in [7, 18] based on prefix-sum operations;
- we introduce two optimizations: pipelining of network communication with GPU-CPU data transfer, and reuse of data computation results that reduces computation time to the price of an increase in memory utilization;
- we extend 1-degree reduction to any vertex of a graph and not only to the largest connected component, providing a distributed implementa-

tion.

The rest of this chapter is organized as follows: in Section 2.1 and Section 2.1.2, we introduce a description of Brandes's algorithm and the state-of-the-art as well; our main contributions are described in-depth in Section 2.2. In Section 2.3, we provide comprehensive experimental results to validate our study. Finally, in Section 2.3 we discuss our results.

2.1 Background and related work

2.1.1 Betweenness Centrality in a nutshell

The first formal definition of the betweenness centrality metric was originally introduced in [10] (see also [2] for further details).

Let $\sigma_{s,t}$ be the number of shortest paths between vertices s, t whereas $\sigma_{s,t}(v)$ represents the number of those shortest paths that pass through v with $s, t, v \in V$. We define the pair-dependency on v of a pair s, t, the ratio $\delta_{st}(v) = \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$. The betweenness centrality of a vertex v is defined as the sum of the pair-dependencies of all pairs on v,

$$BC(v) = \sum_{s \neq t \neq v} \delta_{st}(v) \tag{2.1.1}$$

Before Brandes' work, a simple algorithm computed the BC score by solving the all-pairs-shortest-path problem and then by paths counting. This solution requires $\mathcal{O}(n^3)$ time by using FloydWarshall algorithm and $\Theta(n^2)$ space of pair-dependencies. In order to remove the explicit summation of all pair-dependencies and thus exploiting the natural sparsity of real-world graphs, Brandes introduced the dependency of a vertex v with respect to a given source vertex s:

$$\delta_s(v) = \sum_{w : v \in pred(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_s(w)) \tag{2.1.2}$$

The 2.1.1 can be redefined as sum of dependencies:

$$BC(v) = \sum_{w:v \in pred(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_s(w)) \tag{2.1.3}$$

Algorithm 5 Brandes' algorithm

```
    Input: graph G(V,E)                                        ▷ $G$ is unweighted graph
    Output: betweenness centrality values $BC[v], v \in V$
    Data: $Q$, $S$, $\sigma$, $\delta$, $pred$, $d$
 1: $BC[v] \leftarrow 0$
 2: for $s \in V$ do
 3:     $S \leftarrow$ empty stack                             ▷ Stack of visited vertices
 4:     $pred[v] \leftarrow$ NULL $\forall v \in V$
 5:     $\sigma[v] \leftarrow 0, \forall v \in V, \sigma[s] = 1$
 6:     $d[v] \leftarrow -1, \forall v \in V, d[s] = 0$        ▷ $d$ array of distances from $s$
 7:     $Q \leftarrow \emptyset$
 8:     enqueue $s \rightarrow Q$
 9:     while $Q \neq \emptyset$ do                            ▷ Path counting via BFS
10:         dequeue $v \leftarrow Q$
11:         push $v \rightarrow S$
12:         for all $w$ neighbor of $v$ do
13:             if $d[w] < 0$ then
14:                 enqueue $w \rightarrow Q$
15:                 $d[w] \leftarrow d[v] + 1$
16:             end if
17:             if $d[w] = d[v] + 1$ then
18:                 $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
19:                 append $v \rightarrow pred[w]$
20:             end if
21:         end for
22:     end while
23:     $\delta[v] \leftarrow 0, \forall v \in V$               ▷ Dependency
24:     while $S \neq \emptyset$ do
25:         pop $v \leftarrow S$
26:         for $v \in pred[w]$ do
27:             $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \times (1 + \delta[w])$
28:         end for
29:         if $w \neq s$ then                                 ▷ Update BC
30:             $BC[w] \leftarrow BC[w] + \delta[w]$
31:         end if
32:     end while
33: end for
```

This makes it possible to determine the BC score by solving Single-Source-Shortest-Paths (SSSP) problem for each vertex in the graph. Brandes' algorithm, shown in Algorithm 5, computes BC scores in $\mathcal{O}(nm)$ on unweighted graphs [14] and consists in:

1. computing the single source-shortest-path σ from a single root vertex

s (lines $9 - 22$);

2. summing all dependencies δ from s (lines 24-28) and update BC score (line 30);

3. repeating steps 1. and 2. for each vertex in G.

Sometimes, in the rest of this chapter we refer to the first part as "forward step" since it starts from a single root vertex s and traverses the whole graph, while the second part is also called "backward step" since it starts from the leaves and traverses back the graph till the root.

2.1.2 Related Work

Several authors have tackled the problem of speeding up the exact computation of betweenness centrality. A quite common strategy consists in the parallelization of Brandes' algorithm. This approach requires a fast and memory-efficient traversal algorithm for unweighted graphs. As mentioned above, BC computation on GPU suffers from both the irregular access pattern and the workload unbalance due to traversal steps of the graph (counting of shortest paths and dependency accumulation).

Jia et al. [51] evaluated two types of thread-data mapping: *vertex-parallel* and *edge-parallel*. Briefly, the former approach assigns a thread to each vertex during graph traversal. The number of edges traversed per thread depends on the out-degree of the vertex assigned to each thread. The difference in the out-degree among vertices causes a load imbalance among threads. In particular, since the out-degree distribution of typical scale-free networks (like the social networks) follows a power law [52], the load imbalance is the main reason of poor performance obtained with that approach on GPU systems.

The *edge-parallel* approach solves that problem by assigning edges to threads during the frontier expansion. However, this assignment of threads can also result in a waste of work because the edges that do not originate from vertices in the current frontier do not need to be inspected. The *edge-parallel* approach is not well-suited for graphs with low average degree as well as dense graphs [51]. Shi and Zhang presented GPU-FAN [53] and reported a significant speed-up (11%-19%) with respect to Jia et al. by avoiding data structure duplication and using a different distribution of threads to units of work on graphs with a number of vertices in the range 10000-50000. The vertex-based parallelism is affected by workload unbalance whereas the edge-based parallelism uses more memory and more atomic operations [39, 51].

In [48] and [39, 54] the authors proposed different strategies in order to exploit the advantages of both methods. In detail, Mclaughlin and Bader discussed two hybrid methods for the selection of the parallelization strategy. Their sampling method performs 2.71 times faster, on average, with respect to the edge-parallel approach by Jia et al.

Saryüce et al., in [39, 54] introduced the vertex virtualization technique based on a relabeling of the data structure (e.g. CSR, Compressed Sparse Row). The technique replaces a high-degree vertex v with $n_v = \lceil \texttt{adj(v)} \rceil / \Delta$ virtual vertices having at most Δ neighbors. In other words, the neighbors of high-degree vertices are divided (according to the input parameter Δ) in several groups and each of them is assigned to a virtual vertex. Vertex virtualization technique is not very effective for graphs with low average degree. Moreover, it requires a careful tuning of its parameters. The authors also proposed a coarse-grained approach in which a single GPU executes multiple BFSs at the same time with an increase of the memory requirements.

The strategy employed in [55] is based on [18] and [56]. In [18], the workload due to a single vertex is mapped, depending on the number of its outgoing edges, to a thread, a warp or a cooperative thread array (CTO).

Davidson et al. [56] introduced two strategies aimed at improving the workload balance: the first one splits the frontier into chunks, then the neighbor lists of the vertices in one chunk are assigned to one block of threads, therefore all neighbors of a vertex are processed by the same block; the second one splits the edge lists into chunks and assigns each chunk to one block of threads, therefore edges originated from the same vertex may be processed by different blocks. In both cases the size of chunks is fixed.

Mastrostefano et. al. showed that prefix-sum and binary search operation can be used to define a data-thread mapping that makes possible to achieve a perfect load balancing among threads [7]. We described such approach earlier in Section 1.2.2.

Madduri et al. [47] introduced a technique of checking successors instead of predecessors in the dependency accumulation step. In this way, the dependency accumulation procedure can start from one depth level closer to the root vertex of the BFS tree. Moreover this technique does not require atomic operations during dependencies update. Each predecessor can update its own dependency based on its successors without the need for resolving race conditions. Unfortunately, in an edge-parallel implementation the successor approach still requires atomic operations because multiple threads could be assigned to the same predecessor [48].

In [57], Green and Bader proposed a solution which reduces the memory requirements of local data structures from $\mathcal{O}(m)$ to $\mathcal{O}(n)$ by discarding predecessors array. In the dependency accumulation step, instead of traversing the predecessors, all of the neighbors of a given vertex are traversed.

To speed-up computation of betweenness centrality a totally different approach consists in approximating the values of BC by using extrapolation and sampling methods [58–61]; our work is focused exclusively on exact BC computation while vertex sampling and total time extrapolation is used in some experiments (Section 2.3).

As sketched before, betweenness computations can be parallelized in two ways: coarse- and fine-grained. The coarse-grained parallelism on distributed memory is implemented by duplicating the entire graph, and additional data structures, on each computational node where the computation is carried out for a subset of the vertices. Since each root vertex can be processed independently, this approach requires only one *Reduce* operation in order to update the final BC scores.

For graphs that have a single connected component, the amount of work to perform will be balanced among computational nodes. In this case, a nearly perfect scaling can be expected [48]. However, this approach does not work in case of large scale graphs which cannot be stored in the memory of a single processing unit.

On the contrary, in the fine-grained approach all processing units are involved concurrently on the same computation starting from a single root vertex. On distributed systems, this requires a partitioning of the graph and data structures among the computational nodes. The inter-processor communication phase during shortest path counting and dependency accumulation steps is, in general, the main performance bottleneck.

In [62] was proposed a space efficient distributed algorithm where vertices are randomly partitioned; although such decomposition achieves a good load-balance, it can dramatically reduce data locality increasing the communication costs. On unweighted R-MAT graphs [35], the authors showed a linear scalability up to 8 nodes. Gunrock library also provided an implementation of Brandes' algorithm on an "one-node" Multi-GPU [63]. The authors also evaluated several partitioning strategies. Their best BC implementation is 2.5 times faster than Single-GPU implemented in [55] by exploiting 6 GPUs.

2.1.3 2-Dimensional Decomposition

A careful decomposition of the graph is instrumental in order to increase performance and achieve satisfactory scalability on parallel graph algorithms. In Section 1.2.1 we describe the 1-D partitioning adopted for the BFS Multi-GPU implementation, nevertheless on graph traversal algorithms this technique suffers for poor scalability since it requires all-to-all communications among the P computing nodes [4, 6, 8]

In [4, 6, 29], authors proposed different strategies to reduce the communication cost. Hereafter we recall the 2-D partitioning strategy introduced by Yoo *et al.* in [4] where the computing nodes are arranged as a logical grid with R rows and C columns and mapped onto the adjacency matrix $A_{N\times N}$

(partitioning it into blocks of edges). The processor grid is mapped once horizontally and C times vertically thus dividing the columns in C blocks and the rows in RC blocks, as shown in Figure 2.1.

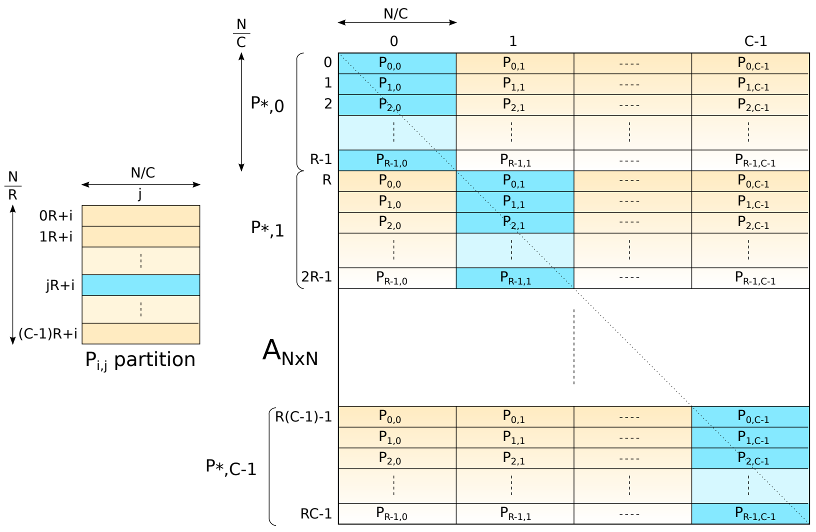


Figure 2.1. Two-dimensional partitioning of an adjacency matrix A with an $R\times C$ processor grid. The matrix is divided into C consecutive groups of $R \times C$ blocks of edges along the vertical direction. Each block is a $N/(RC) \times N/C$ sub-matrix of A. Different groups of blocks are colored with different yellow gradients. For each block, the column of processors owning the corresponding vertices (row indexes) are shown in blue. On the left part, it is shown the sequence of blocks, from top to bottom, assigned to the generic processor (p_i, p_j). The colors correspond to the blocks assigned to the processor in each group of blocks.

Processor P_{ij} handles all the edges in the blocks $(mR + i,\ j)$, with $m = 0, ..., C - 1$. Vertices are divided into RC blocks and processor P_{ij} handles the block $jR + i$. Considering the edge lists represented along the columns of the adjacency matrix, this partitioning can be summarized as follows:

(i) the edge lists of the vertices handled by each processor are partitioned among the processors in the same grid column;

(ii) for each edge, the processor in charge of the destination vertex is in the same grid row of the edge owner.

The 2-D partitioning is well-suited for graph traversal algorithms like the BFS; in [41], authors describe an optimized implementation of BFS on GPUs using 2-D decomposition. In details, 2-D BFS requires two communication phases, called *expand* and *fold*. The first one involves the processors in the same grid column whereas the second those in the same grid row. Algorithm 6 shows a pseudo code for a parallel BFS with 2-D partitioning.

Algorithm 6 Parallel BFS with 2-D partitioning

```
    Input: graph G(V,E), starting vertex $s$, processor $P_{ij}$
    Output: array of predecessors $pred$
 1: $d[v] \leftarrow -1, \forall v \in V$                      ▷ Distance vector from root vertex
 2: $Q \leftarrow \emptyset$
 3: $lvl \leftarrow 0$                                          ▷ BFS level or depth
 4: $pred \leftarrow -1, \forall v \in V$
 5: $bmap \leftarrow 0, \forall v \in V$                        ▷ Bitmap array
 6: if $s \in P_{ij}$ then
 7:     $d[s] \leftarrow 0$
 8:     $pred[s] \leftarrow s$
 9:     $bmap[s] \leftarrow 1$
10:     enqueue $s \rightarrow Q$
11: end if
12: while true do
13:     $lvl \leftarrow lvl + 1$
14:     gather $Q$ from column $j$                              ▷ Expand vertical communication
15:     $Q_r \leftarrow$ expandFrt $(lvl, bmap, Q, d, pred)$    ▷ Expand frontier
16:     exchange $Q_r$ for row $i$                              ▷ Fold horizontal communication
17:     append updateFrt $(lvl, bmap, Q_r, d, pred) \rightarrow Q$   ▷ Update frontier
18:     if $Q = \emptyset$ for all processors then
19:         break
20:     end if
21: end while
```

At the beginning of each step, each processor has its own subset of the *frontier* set of vertices (initially only the root vertex). The search entails the scanning of the edge lists of all the frontier vertices so, due to property *(i)*, each processor gathers the frontier sets of vertices from the other processors in the same processor-column (expand vertical communication, line 14). In the **expandFrt** procedure, new vertices w_i are discovered by inspecting all outgoing edges (v, w_i) of current frontier Q.

The unvisited vertices are marked visited using the *bmap* array, while predecessors array *pred* and distance array *d* are updated with the predecessor vertex and the current level *lvl* respectively. At the end, new discovered vertices w_i are enqueued on a processor based queue Q_r in order to be sent to their owner processor.

Due to property *(ii)* this exchange only involves processors in the same processor-row (fold horizontal communication, line 16). In the **updateFrt** procedure, each received vertex w that has not been visited yet, is marked as visited and its predecessor and distance are updated. Finally, it is enqueued on the frontier Q for the next level expansion.

The graph traversal ends when the frontiers for all processors are empty, meaning that the whole connected component of the root vertex has been visited.

The main advantage of the 2-D partitioning is to reduce the number of communications. For a given number of processors p, 1-D modulo-based partitioning requires $\mathcal{O}(p)$ data transfers at each step whereas the 2-D partitioning only requires $2 \times \mathcal{O}(\sqrt{p})$ communications.

It is worth noting that in Algorithm 6 a bitmap is used to keep track of visited vertices in order to reduce memory footprint as described in [29].

2.1.4 Compressing Networks

An exhaustive evaluation of betweenness centrality requires solving the SSSP problem starting from each vertex. For large-scale graphs with millions of vertices, finding all SSSPs is unfeasible. Nevertheless, in some cases the betweenness centrality of defined sub-structures of the graph, or vertices with specific properties can be analytically computed with no need of resorting to Brandes' algorithm [49, 50, 64]. For example, vertices with exactly one neighbor (degree-1 vertices) have BC score 0, since they are endpoint and cannot be crossed by any shortest path. As a matter of fact, a careful handling of degree-1 vertices improves overall performance of Brandes' algorithm

- by skipping the execution of Brandes' algorithm rooted from degree-1 vertices;
- by reducing the number of vertices to explore.

It is enough clear that the performance improvement depends on graph characteristics, indeed the more degree-1 vertices are in the graph, the greater will be the improvement (see for example Table 2.1). Since degree-1 vertices "influence" the betweenness value of their neighbor, they cannot be just removed from the graph, but it is necessary to calculate their contribution to the BC values of other vertices.

Formally, let $G = (V, E)$ be an undirected and unweighted graph with $N = |V|$ vertices and $M = |E|$ unordered pairs, let $(u, v) \in E : deg(u)$. Since all the shortest paths terminating into a degree-1 vertex have to go through its neighbor, the contribution $\delta_{sv}(w)$ could be not necessarily equal to 0. From the algorithm point of view, 1-degree reduction extends Brandes' algorithm by adding a preprocessing procedure and by employing a different formulation for dependencies computation.

In detail, the preprocessing step computes $\forall(u, v) \in E : deg(u) = 1$:

$$\begin{aligned} \omega(v) &= \omega(v) + 1 \\ BC(v) &= BC(v) + 2 \cdot (N - \omega(v) - 2) \end{aligned} \tag{2.1.4}$$

where $\omega(v)$ represents the contribution of u to v and initially is set to 0. When a degree-1 vertex u is detected, the value $\omega(v)$ of its neighbor v is incremented, and u is removed from the graph. When u is removed from the graph, the value $BC(v)$ needs to be updated in order to consider the contribution of paths starting from all other vertices connected to v and terminating in u (to be precise, in the formula 2.1.4, N does not correspond to the number of vertices in the graph, but to the number of vertices in the connected component of v).

After the preprocessing step, Brandes' algorithm is executed over the residual graph $G'(V', E')$ obtained by 1-degree removal. Concerning dependency accumulation, 2.1.2 and 2.1.3 can be re-defined as follows:

$$\begin{aligned} \delta_s(v) &= \sum_{w:v\in pred(w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1+\delta_s(w)+\omega(w)) \\ BC(v) &= \sum_{w:v\in pred(w)} \delta_s(w)\cdot(\omega(s)+1) \end{aligned} \tag{2.1.5}$$

After applying a first time the 1-degree preprocessing, in the residual graph $G'(V', E')$ there may appear new degree-1 vertices due to the removal process, therefore the preprocessing may be called iteratively until no additional vertex can be removed (tree vertices removal).

The 1-degree reduction algorithm described in [65] only enables skipping the execution of Brandes' algorithm rooted from degree-1 vertices, augmenting the BC contributions for degree-1 vertices during the accumulation stage from its neighbor vertex: it does not remove the vertex from the graph, therefore degree-1 vertices are still explored. In [49, 50] is presented an algorithm for 1-degree reduction that, by means of a preprocessing step, removes degree-1 vertices from the graph reducing the number of vertices to explore. Finally in [39, 54] authors provided a GPU implementation of 1-degree reduction on the biggest connected component of the graph. If G is not connected, the preprocessing step can be repeated for each connected component separately [49].

Graph	Scale	EF	1-degree(%)
R-MAT	16	4	15.15
R-MAT	16	16	13.12
R-MAT	16	32	10.09
R-MAT	18	4	14.53
R-MAT	18	16	13.50
R-MAT	18	32	11.70
soc-LiveJournal1 [36]	~ 22	~ 14	21.39

Table 2.1. Percent of degree-1 vertices on different graphs

2.2 Betweenness Centrality Computation on GPUs

Our Multi-GPU Betweenness Centrality implementation (**MGBC**) is based on Brandes algorithm introduced in Section 2.1.1. Exactly as Brandes' algorithm, MGBC is composed by three main steps: (1)shortest paths counting, (2)dependency accumulation and (3)update of BC scores.

The implementation reuses some techniques adopted for the Multi-GPU BFS implementation described in Chapter 1, since shortest path counting on unweighted graphs can be performed by using a BFS, whereas for the dependency accumulation, the BFS tree is explored in reverse order from the leaves up to the root. Furthermore we use the 2-D partitioning strategy as described in 2.1.3.

2.2.1 Shortest Path Counting

Algorithm 7 describes our shortest path counting implementation on Multi-GPU.

In lines 8-13 root vertex is enqueued and variables are initialized, after that the BFS loop starts; at the beginning of each step, each processor has its own subset of the frontier, due to 2-D partitioning, processors on the same column exchange frontier vertices (vertical communication), so all processors on the same column share the same frontier. Note that in Brandes' algorithm, σ (the shortest path counting) depends on the shortest path counting of its predecessors (see line 18 in Brandes Algorithm), therefore σ values are exchanged together with vertices (line 16).

Comparing the Algorithm 7 with Brandes' Algorithm 5, we can notice that predecessor array has been removed, this is because we use the technique known as neighbor traversal approach [57]: we keep track of all visited vertices using one single array Q to store and accumulate the BFS frontiers, and use Q_{off} to save the offset index at which each frontier starts per BFS

Algorithm 7 Shortest Path Counting on Multi-GPU

```
    Input: graph G(V,E), starting vertex $s$, processor $P_{ij}$
 1: $BC[v] \leftarrow 0, \forall v \in V$
 2: $\sigma[v] \leftarrow 0, \forall v \in V$
 3: $d[v] \leftarrow -1, \forall v \in V$
 4: $Q \leftarrow$ empty queue
 5: $lvl \leftarrow 0$                                        ▷ BFS level or depth
 6: $nq \leftarrow 1$
 7: $Q_{off}[0] \leftarrow 0$
 8: if $s$ belongs to $P_{ij}$ then
 9:     $\sigma[s] \leftarrow 1$
10:     $bmap[s] \leftarrow 1$
11:     $d[s] = 0$
12:     enqueue $s \rightarrow Q$
13: end if
14: while true do
15:     $lvl \leftarrow lvl + 1$
16:     gather $Q$ and $\sigma$ from column $j$               ▷ Vertical communication
17:     $Q_{off}[lvl] \leftarrow Q_{off}[lvl-1] + nq$
18:     $nq \leftarrow 0$
19:     $Q_r \leftarrow$ expandFrt $(lvl, bmap, Q, Q_{off}, d, \sigma)$     ▷ Expand frontier
20:     exchange $Q_r$ and $\sigma$ for row $i$            ▷ Horizontal communication
21:     append $Q_j \rightarrow Q$
22:     append updateFrt $(lvl, bmap, Q, Qoff, d, \sigma) \rightarrow Q$    ▷ Update frontier
23:     $nq \leftarrow$ number of vertices added to $Q$
24:     if $nq = 0$ for all processors then
25:         break
26:     end if
27: end while
```

level. In the dependency accumulation part, instead of traversing the predecessors, all the neighbors of a vertex are analyzed but only those discovered in a previous level are considered.

Predecessor array removal brings another benefit for the Multi-GPU implementation since it is not necessary to exchange predecessors information at each level.

The **expandFrt** procedure implements frontier expansion, where all frontier's neighbors are traversed, mark as visited and σ values are updated accordingly; after that, newly discovered vertices are sent to their owner processes (horizontal communication) together with partial σ values. Notice that in this case σ values are partial since they correspond to the number of shortest paths counted on that processor, the final value is aggregated on the processor which owns the vertex.

Finally, the **updateFrt** procedure is used to update both current frontier

Algorithm 8 expandFrt

```
1: for all $v \in CQ$ in parallel do                ▷ $CQ$ is the current frontier
2:     for all neighbor $w$ of $v$ in parallel do
3:         if $bmap[w] = 0$ then
4:             $bmap[w] \leftarrow 1$
5:             $d[w] \leftarrow lvl$
6:             $r \leftarrow$ row of $w$'s owner
7:             atomically enqueue $w \rightarrow Q_r$
8:         end if
9:         if $d[w] = lvl$ then
10:            atomically $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
11:        end if
12:    end for
13: end for
```

and σ values: for all received vertices, those not visited yet are marked and σ is updated according to the Brandes algorithm. If no more vertices have been enqueued (line 24) for all processors, the loop ends.

The **expandFrt** function is described in Algorithm 8: for each vertex in the frontier, all neighbors are visited by employing the data-thread mapping technique described in 1.2.2 that uses as many CUDA threads as the number of neighbors, with the main difference that $\sigma[w]$ values need to be calculated as well.

Starting from the vertices in the current frontier CQ, all their neighbors are traversed and those not visited yet are marked, σ is updated according to the Brandes algorithm. In this step, in addition to the above mentioned data-thread mapping technique, we use the bitmask solution described in [41].

Note that in line 7 we use different queues to append newly discovered vertices, one for each processor in a row.

Figure 2.2 provides an example of the data-thread mapping technique described in 1.2.2 together with the BFS frontiers accumulation.

2.2.2 Dependency Accumulation

Algorithm 9 describes our dependency accumulation on Multi-GPU. After the shortest path counting stage, processors on the same column share the same accumulated frontier and corresponding offsets per BFS level respec-

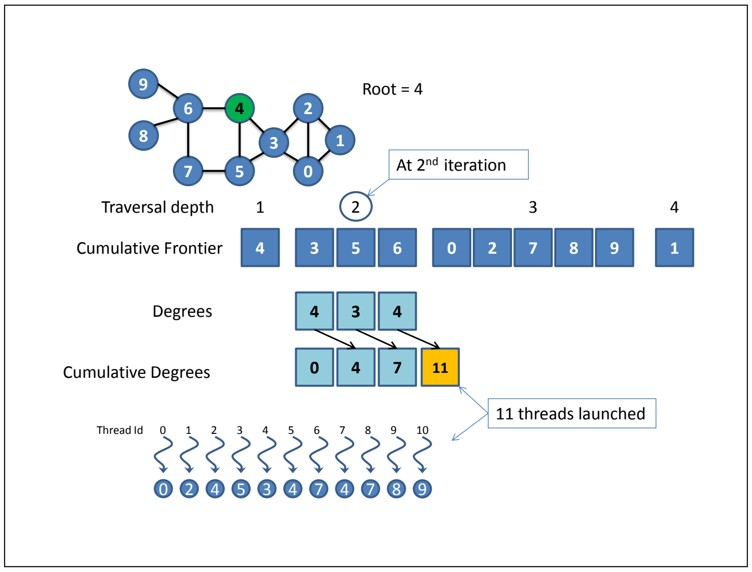


Figure 2.2. Example of data-thread mapping and BFS frontiers accumulation technique

tively in $Q[]$ and $Q_{off}[]$. The accumulated frontier is used for the neighbor traversal approach, furthermore we adopt also the checking successor technique developed by Madduri [47], which consists in checking successors rather than the predecessors of each vertex.

Algorithm 9 Dependency Accumulation on Multi-GPU

```
1: exchange $d$ and $\sigma$ for row $i$                      ▷ Horizontal communication
2: $\delta[v] \leftarrow 0, \forall v \in V$
3: $depth \leftarrow lvl - 1$
4: while depth > 0 do
5:     accumulateDep $(depth, Q, Q_{off}, d, \sigma, \delta)$   ▷ Accumulate dependencies
6:     all reduce $\delta$ among column $j$                    ▷ Vertical communication
7:     updateDep $(lvl, Q, Qoff, d, \sigma)$                   ▷ Update dependencies
8:     exchange $\delta$ among row $i$                         ▷ Horizontal communication
9:     $depth \leftarrow depth - 1$
10: end while
```

Since leaves of the BFS tree do not have successors, we start our algorithm one level closer to the root and since the root does not contribute to its BC value, we stop the calculation at level 1. Note that on line 1 in Algorithm 9, both vertices depth d and sigma σ are exchanged among computing nodes in the same row before starting the iteration steps.

According to Brandes algorithm, the dependency $\delta[w]$ is calculated from the shortest path count $\sigma[v]$ and dependency value $\delta[v]$ of all its predecessors. With 2-D decomposition, each processor has to accumulate the contribution to $\delta[w]$ from those predecessors v for which the processor holds the edge (v, w): this first accumulation step is performed in the function **accumulateDep** on line 5.

After that, the dependency contributions are exchanged and summed up through the `All Reduce` MPI communication procedure involving the processors on the same column (vertical communication line 6) since the processors of the same column hold all the edges of w. This allows each processor to calculate the final dependency for those vertices w it owns, multiplying the accumulated dependency value by $\sigma[w]$ (function **updateDep**).

Finally $\delta[w]$ values are exchanged among processors on the same row (horizontal communication line 8) since they are required for the next iteration.

The kernel function **accumulateDep** is described in the pseudo-code 10: first we select the vertices in the accumulated frontier Q based on the *depth* (line 1), then we check if their neighbors are successors, in which case we update the accumulated dependency $\delta[w]$. The data-thread mapping technique employed in the **expandFrt** procedure is used here as well, so we start one CUDA thread per neighbor.

Algorithm 10 AccumulateDep

```
1: $CQ \leftarrow Q[Q_{off}[depth]]...Q[Q_{off}[depth-1]]$
2: for all $w \in CQ$ in parallel do
3:     for all neighbor $v$ of $w$ in parallel do
4:         if $d[v] = d[w] + 1$ then
5:             atomically $\delta[w] \leftarrow \frac{1+\delta[v]}{\sigma[v]}$
6:         end if
7:     end for
8: end for
```

2.2.3 1-Degree Reduction

Algorithm 11 1-Degree Preprocessing

```
    Input: $G(V,E)$                                              ▷ G is undirected
    Output: $\omega[v]$, $G'(V',E')$
 1: $R \leftarrow$ empty List
 2: $E' \leftarrow$ empty List
 3: if $u \bmod \#P == P_i : (u,v) \in E$ then                  ▷ $P_i$ is processor $i^{th}$
 4:     assign (u,v) to $E_i$
 5: end if
 6: sorting $E_i$ by $u$
 7: for $(u,v) \in E_i$ do
 8:     if $\nexists (w,z) \in E_i : u == w$ then    ▷ (w,z) the successor or predecessor in E
 9:         append $(v,u) \rightarrow R$
10:         $\omega[v] = \omega[v] + 1$
11:     else
12:         append $(u,v) \rightarrow E'$
13:     end if
14: end for
15: if $v \bmod \#P == P_i : (v,u) \in R$ then
16:     assign (v,u) to $R_i$
17: end if
18: sorting $R_i$ by $v$
19: for $(w,z) \in E_i$ do
20:     if $\nexists (v,u) \in R_i : v == w$ then
21:         append $(u,v) \rightarrow R$
22:     else
23:         append $(w,z) \rightarrow E'$
24:     end if
25: end for
```

In this Section, we discuss our algorithm for the removal of degree-1 vertices. Unlike previous approaches, we provide a distributed preprocessing algorithm described by pseudo-code in Algorithm 11.

For the sake of simplicity we do not remove tree vertices from the graph by calling repeatedly the preprocessing. Moreover, the algorithm is implemented only on CPU because it is executed only one time prior the full BC calculation, it operates directly on edge lists data structures which are not shared on GPU, and finally computing time is already so short compared to the overall process that it is not worth provide a GPU implementation.

Since 1-degree reduction increases the sparsity of the adjacency matrix, a compaction of data structures is required. Alternatively, we can perform the preprocessing before CSR building procedures in order to have more dense data structures directly.

One-degree reduction requires to identify vertices with degree one and this task is easier to accomplish if each vertex, along with all its edges, is stored on the same processor. This can be easily performed with 1-D partitioning using modulo operator (see Section 1.2.1). More in detail, edges are distributed among processors according to the naive rule: $(u, v) \in P_i$ if the remainder of the integer division $u/\#P$ (where $\#P$ is the number of processors) is equal to i. After that, they are sorted by antecedent vertex u and processed sequentially: if a degree-1 vertex u is discovered, $\omega[v]$ is incremented and the edge (u, v) is added to the list R. Otherwise, all the edges from u are appended to the new edge list E' of the residual graph.

In a undirected graph, for each edge (u, v) a symmetric edge (v, u) exists, so if (u, v) is removed due to 1-degree processing, the symmetric edge has to be removed as well; with 1-D partitioning the edge (v, u) needs to be sent to the processor owning v for its removal. This task is illustrated in lines 15-24: when the edge (u, v) is removed, its symmetric edge (v, u) is inserted into the removed edges list R, then edges in R are distributed according to 1-D partitioning so that each processor can remove them from E'. At the end of the procedure, the residual graph $G'(V', E')$ is built. After preprocessing, the BC score is computed over G' only.

Unlike [39], our 1-degree preprocessing does not use the transpose of the adjacency matrix, reducing in this way the memory requirements.

In order to support graphs with multiple connected components, the contribution of degree-1 vertices to BC values cannot be computed during preprocessing: observing the formula $BC(v) = BC(v) + 2 \cdot (N - \omega(v) - 2)$, we already highlighted that N corresponds to the number of vertices in the same connected component of v including degree-1 vertices. It is apparent that N is equal to $|V|$ if G is composed by just one connected component.

For any vertex s, we can compute N_s, the number of vertices of its connected component, during shortest path counting. When a new vertex v is discovered during graph traversal from root vertex s, N_s is updated as follows: $N_s = N_s + \omega[v]$. Computing N_s is required whenever $\omega[s] \neq 0$, in other words, only if vertex s is connected to a degree-1 vertex.

There are two alternatives for the computation of N_s: a) using atomic operations during shortest path counting; b)using a parallel reduction of the distances array before betweenness score update. In the latter case, the procedure should not consider the contribution of unvisited vertices. Concerning the performance, solution(a) is well-suited for a serial algorithm. On parallel systems the best solution depends on the cost of atomic operations.

2.2.4 Optimizations

Besides the algorithms described in previous sections, we describe also three different optimizations for both distributed and shared-memory systems.

On shared-memory systems the data-thread mapping described in Section 1.2.2 is used to balance workload among multiple CUDA threads during graph traversal: a thread is assigned to each neighbor of the vertices in the current frontier CQ this technique requires to build an array deg with the degrees of the vertices in CQ and execute a prefix-sum operation to calculate the offsets CD used to map the neighbor vertex to its parent. In the BC algorithm, given a root vertex, graph traversal occurs twice: during the shortest path counting and the dependency accumulation. Effectively, we can observe that the latter is carried out along the BFS tree built in the previous forward step.

Since the data-thread mapping requires the building of the offset array CD from the vertices in the frontier and since the frontiers used during the forward shortest path counting and the backward dependency accumulation are the same, storing and accumulating the offset array CD in the same way the frontier Q is stored and accumulated during the shortest path counting, reduces computation time during dependency accumulation because the offset array is already available. By exploiting the symmetry between forward and backward step, binary search results can be reused as well. Obviously, this time-saving has an extra memory cost that is, at most, $\mathcal{O}(n)$.

The second optimization consists in combining fine- and coarse-grained approaches.

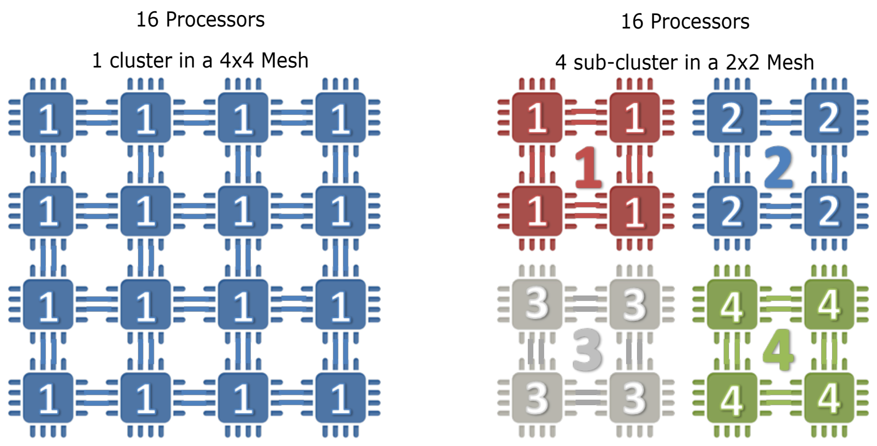


Figure 2.3. Sub clustering

A set of processors is split into sub-clusters. Each sub-cluster, in turn, is organized as a bi-dimensional grid of processors (see for example Figure 2.3). Processing nodes in the same sub-cluster work at the fine-grained level: the graph is distributed among the nodes according to a 2-D partitioning, and partial BC values are calculated starting from a subset of vertices. Independent sub-clusters work at the coarse-grained level: the whole graph and additional data structures are replicated in each sub-cluster. In the end a reduce operation updates the final BC scores. We implement this solution by creating a hierarchy among processes managed by different MPI communicators. Although the amount of work in each sub-cluster can be different when processing graphs with multiple connected components, with the sub-clusters solution it is possible to take advantage of both fine-grained and coarse-grained approach. A comprehensive analysis concerning multi-sub-cluster solution is also provided in Section 2.3.4.

Finally, the proposed distributed algorithm pipelines MPI communication and CPU-GPU data transfer. Although Nvidia provides several techniques to reduce communication overhead such as GPUDirect RDMA [66], we adopt a simple pipelining pattern between two consecutive communications, whereby the cost of the communication through the PCI bus can be partially hidden. In particular, right after the shortest path counting stage, both the distance vector d and sigmas σ values are exchanged among processors in the same grid row. Since the computation is totally delegated to GPU, usually two consecutive independent communications comply with the following pattern:

1. synchronous-copy of σ from GPU to CPU;
2. exchange of σ among processors in the same grid row;
3. synchronous-copy of σ from CPU to GPU.
4. synchronous-copy of d from GPU to CPU;
5. exchange of d among processors in the same grid row;
6. synchronous-copy of d from CPU to GPU.

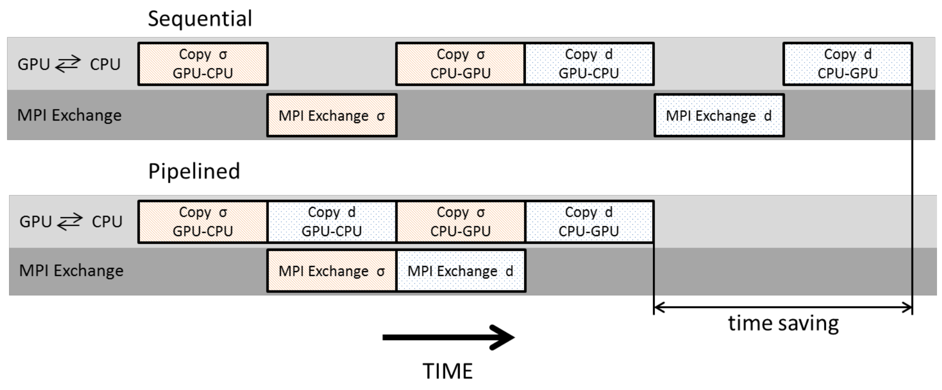


Figure 2.4. Pipelining GPU - CPU data transfer with MPI communication.

In this naive pattern, data transfer procedure ends after six synchronous steps. However, by exploiting CUDA Asynchronous Copy and CUDA Streams, the two communications can be completed in four synchronous steps (see Figure 1reffig:overlap):

1. asynchronous-copy of σ from GPU to CPU; asynchronous-copy of d from GPU to CPU;

2. exchange σ among processors in the same grid row;

3. asynchronous-copy of σ from CPU to GPU; exchange d among processors in the same grid row;

4. asynchronous-copy of d from CPU to GPU.

In general, MPI communication dominates host-device communication. As a consequence, the benefit due to this pipelining is only partial.

2.3 Experimental Results

We start by comparing our Multi-GPU Betweenness Centrality (MGBC) solution with other implementations on a single GPU, since most of them do not offer full support for a distributed Multi-GPU configuration. Actually, the Gunrock library provides a Multi-GPU implementation but only for GPUs connected to the same node. Other implementations, like [48], support only the coarse-grained parallelism where each GPU works independently on the same graph. All those solutions can not be used for very large graphs, like Friendster or Twitter, which are too large to fit in the memory of a single system. Our Multi-GPU solutions are then analyzed looking at their weak and strong scalability, showing how communication and computation change depending on the graph under study. Finally we consider the effect of degree-1 removal and we measure the speed-up it offers with respect to our heuristic-free implementation.

2.3.1 Evaluation Platforms

Numerical experiments have been carried out on two different systems: *Piz Daint* at Centro Svizzero di Calcolo Scientifico (CSCS) and on *Drake*, a server equipped with two K80s GPU available at National Research Council of Italy. Details of the two configurations are listed in Table 2.2. The code has been built with the GNU C compiler version 4.8.2, CUDA C compiler version 6.5 and Cray MPICH version 7.2.2 on Piz Daint and OpenMPI 1.8.4 on Drake. We employ the exclusive scan implemented in the Thrust Library [67]. The code uses 32-bit data structures except for graph generation. When possible, we report the time (in seconds) for total BC computation. However, for very large graphs we measure the time only for a representative subset of source vertices and then extrapolate the expected time for the whole graph (source vertices are selected randomly among not isolated vertices).

	Piz Daint	**Drake**
System	Cray XC30	–
Nodes	5272	1
Network Topology	Aries Drangonfly	–
Processor	Xeon E5-2670	Xeon E5-2640v3
Cores	8	32
Clock (GHz)	2.60	2.60
Memory (GB)	32	128
GPU	Nvidia K20x	Nvidia K80
CUDA Cores	2688	2496
Clock (MHz)	732	875
Memory (GB)	6 DDR5	12 DDR5

Table 2.2. Hardware platforms.

2.3.2 Data Sets

We measured the performance for both R-MAT [35] and real-world graphs. R-MAT is a recursive graph generator that creates networks with high variance of degree distributions and low graph diameter. We generate R-MAT graphs using parameters a, b, c, and d equal to 0.57, 0.19, 0.19, 0.05 respectively [35]. The number of vertices of a R-MAT graph is defined by a `scale` factor and it is equal to 2^{scale}. The edge factor parameter (EF) is used to set the number of edges that is equal to $2^{\text{scale}} \times \text{EF}$. For the generation of R-MAT graphs, we employ the routines available in the reference code for the Graph500 benchmark[a]. In Table 2.3, we provide the dataset used for the experiments.

Graph	**|V|**	**|E|**	**SCALE**	**EF**	d	**1-degree**
com-amazon	334863	925872	18.35	2.76	44	4.68
RoadNet-CA	1965206	2766607	20.91	1.41	849	16.27
RoadNet-PA	1088092	1541898	20.05	1.41	786	17.13
com-LiveJournal	3997962	34681189	21.93	8.67	17	19.2
com-Orkut	3072441	117185083	21.55	38.14	9	2.21
Friendster	65608366	1806067135	25.97	27.53	32	1.2
Twitter-2010	41652230	1468365182	25.3	35.25	–	4.5

Table 2.3. Dataset of real-world graphs. $|V|$ and $|E|$ represent the number of vertices and edges, respectively; d is the diameter of the graph; last column is the percentage of degree-1 vertices.

Regarding real-world graphs, we selected instances with different properties from the SNAP collection [36] while Twitter graph from [37]. We use only undirected graphs for all of our experiments except for Twitter.

Exact BC calculation requires execution of BC search starting from all vertices in the graph, this may require a long time; for this reason, whenever possible, we selected a subset of vertices and extrapolated linearly the time for all the graph. Different strategies for vertex selection can be used [58,59],

[a] `www.graph500.org`

we choose the most straightforward strategy that is uniformly at random selection. More in detail:

- when comparing with other implementations on single GPU (Section 2.3.3), we selected the first 10000 vertices of the biggest connected component, because those solutions do not support random selection;
- for sub-clustering (Section 2.3.4.3) and 1-degree reduction (Section 2.3.4.2) experiments, the BC algorithm was executed for all vertices, otherwise it would be difficult to evaluate their benefits;
- for all other experiments, vertices were selected randomly among those with positive degree, moreover, when comparing results for the same graph, the random seed used is the same.

2.3.3 Single-GPU

The Single-GPU implementation is obtained from our distributed algorithm by removing network and host-device communications. We compare our solution on single GPU (without any heuristic and optimization) to those proposed in Mclaughlin and Bader [48], Saryüce et al. [39] and Gunrock [55]. In Table 2.4, we report the mean time (seconds) for each implementation. For this first group of experiments, since other solutions do not allow to select vertices randomly, the mean time is computed over the first 10000 vertices of the biggest connected component.

Graph	**Type**	**Mclaughlin**	**Saryüce mode-2**	**Saryüce mode-4**	**Gunrock**	**MGBC**
RoadNet-CA	Road Network	0.067	0.371	0.184 (**0.023**)	0.298	0.085
RoadNet-PA	Road Network	0.035	0.210	0.114 (**0.013**)	0.212	0.071
com-Amazon	Social Network	0.008	0.009	0.006 (0.007)	–	**0.005**
com-LiveJournal	Social Network	0.210	0.143	**0.084** (0.120)	–	0.100
com-Orkut	Social Network	0.552	0.358	**0.256** (0.269)	–	0.314

Table 2.4. Comparison with other implementations on real-world graphs. On Gunrock some executions do not terminate.

Concerning Saryüce's implementations, we evaluated two of their data-mapping strategies. The first one, called *mode-2* employs edge-based GPU parallelism, instead the second one, *mode-4*, uses virtual-vertex based with strode access [39]. Concerning mode-4, we also report in parentheses the virtualization time. Experiments show that the hybrid approach of Mclaughlin performs better with respect to others on graphs with a very small edge factor like road networks. On the other hand, the vertex-virtualization technique achieves very good performance on more dense graphs. However, such approach requires an *a priori* tuning of the virtual-vertex parameter. Although the design is focused on distributed systems, our BC implementation generally achieves very satisfactory performance without any specific tuning compared to other implementations.

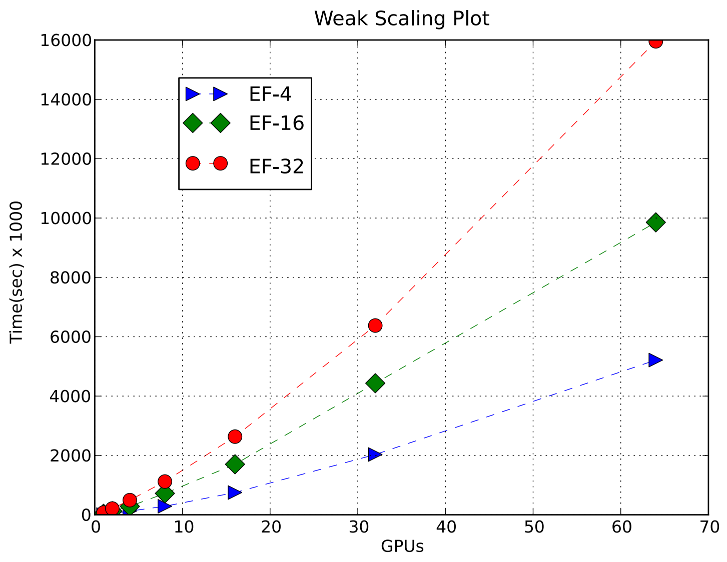


Figure 2.5. Weak scaling plot for R-MAT with SCALE from 20 to 26 and EF 4, 16, 32.

2.3.4 Multi-GPU and Sub-Clustering

We study the scalability of the Multi-GPU implementation, looking at both the weak and the strong scaling. For scaling experiments we did not exploit the 1-degree reduction heuristic and we performed 10000 BC computations selecting randomly the start vertices among those with positive degree; while for Sub-clustering and 1-degree experiments we calculate full BC (i.e. running the algorithm for all vertices in the graph).

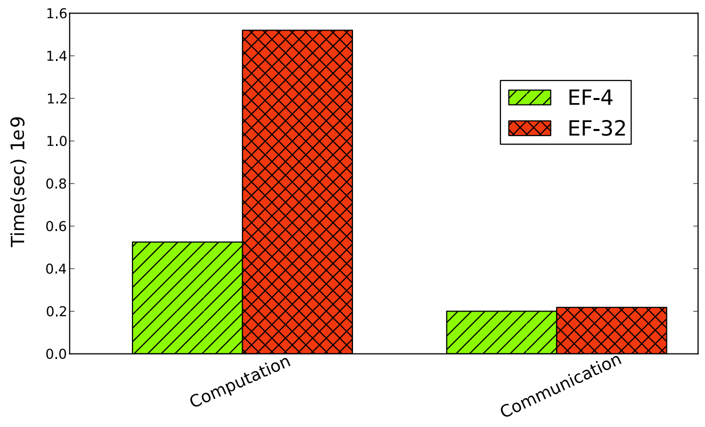


Figure 2.6. R-MAT graphs SCALE 22 on 4 GPUs.

2.3.4.1 Scaling

Weak scaling experiments show how MGBC behaves as the problem size increases. Figure 2.5 illustrates the capability to calculate betweenness centrality for graphs that a single GPU can not handle due to memory and/or time constraints.

Even if the amount of data remains the same for each computing node, the time to compute BC is not constant for R-MAT graphs with different EFs and SCALE that increases linearly from 20 up to 26. As a matter of fact, both computation and communication time increase.

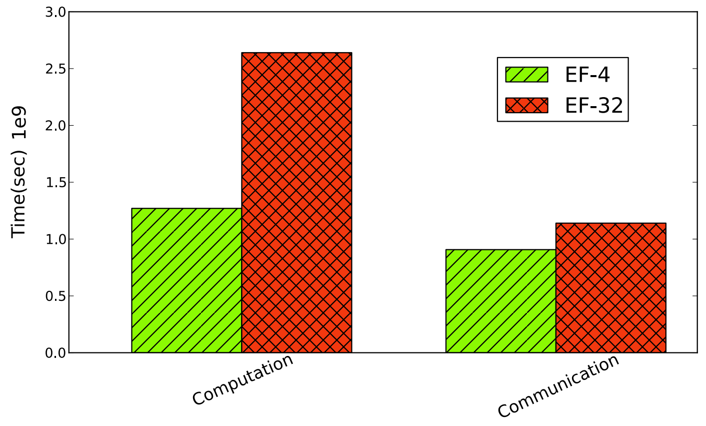


Figure 2.7. R-MAT graphs SCALE 26 on 64 GPUs.

MGBC unveils better scalability for EF 4: unlike the BFS algorithm where only one successor is considered, with BC all shortest paths need to be considered and counted, therefore, when there are more neighbors, more time is required to count shortest paths and to accumulate dependency contributions.

Moreover, by comparing the time spent for computation and communication for the R-MAT graphs with equal scale but different edge factors (see figures 2.6 and 2.7) we notice that while the communication time remains almost the same between EF 4 and 32, computation time is more than twice higher for EF 32.

By means of strong scaling experiments, we evaluate performance on fixed-size graphs while increasing computational resources. Figure 2.8 shows the strong scalability for R-MAT graphs at SCALE 24 and two different partitioning: a 1-D partitioning with only one processors column and a 2-D partitioning.

For the 1-D partitioning, scaling stops between 16 and 32 nodes, whereas the 2-D partitioning continues to scale till 64 nodes. The reason is that with a 1-D partitioning, the MPI overhead, due to the all-to-all communication pattern increases more quickly with respect to the 2-D partitioning. When

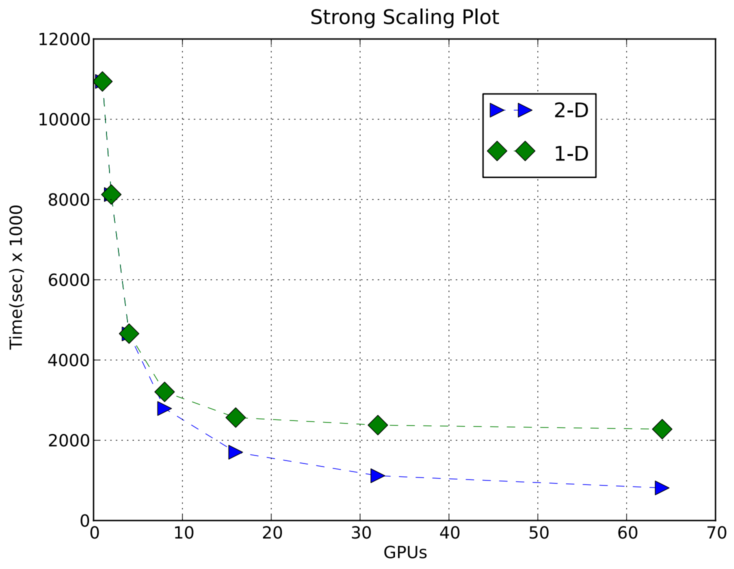


Figure 2.8. Strong scaling for R-MAT graphs with SCALE 24 and EF 16 using 1-D and 2-D partitioning

.

moving from 1 to 2 nodes, there is $\sim$ 40% improvement in spite of the communication overhead introduced.

Figure 2.9 shows the strong scaling for a R-MAT graph (SCALE 22, EF 16) and for Orkut (SCALE 21.5, EF 38). Notice that the total time required to compute the exact BC score of Orkut exploiting 64 GPUs is about 37 hours. Finally, in Figure 2.10, the strong scaling for Friendster and Twitter graphs are also evaluated. Notice that the minimum number of GPUs needed to store the graph is 16.

2.3.4.2 1-Degree

Finally, in this section we summarize the performance of the 1-degree reduction heuristic. Experiments in this section have been executed running the BC algorithm for all vertices (full BC), otherwise it would be difficult to

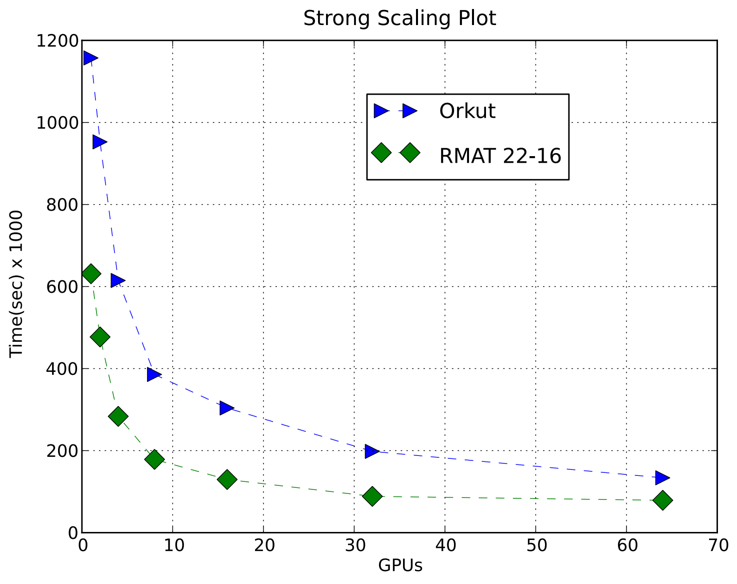


Figure 2.9. Strong scaling for a R-MAT graph with SCALE 22 and EF 16 and Orkut graph.

determine the improvement over the algorithm without 1-degree.

First of all, we evaluated the strong scalability of the pre-processing step on the Piz Daint system. Figure 2.11 illustrates the strong scaling of the Algorithm 11 applied to a R-MAT graph with SCALE 22 and EF 16. The algorithm exhibits a near-linear speedup, so that the communication does not represent a bottleneck during preprocessing step.

To figure out the impact of the preprocessing and the 1-degree reduction on exact BC computation, we computed the BC scores of all vertices for R-MAT graphs with SCALE 20 and different EFs exploiting 2x2 grid of GPUs and for some real graphs on single GPU. More in detail, Table 2.5 for R-MAT graphs and Table 2.6 for real graphs show the total time, mean time[b], and

[b]The mean time is computed by dividing the total time for the number of BC searches executed which is less than the total number of vertices.

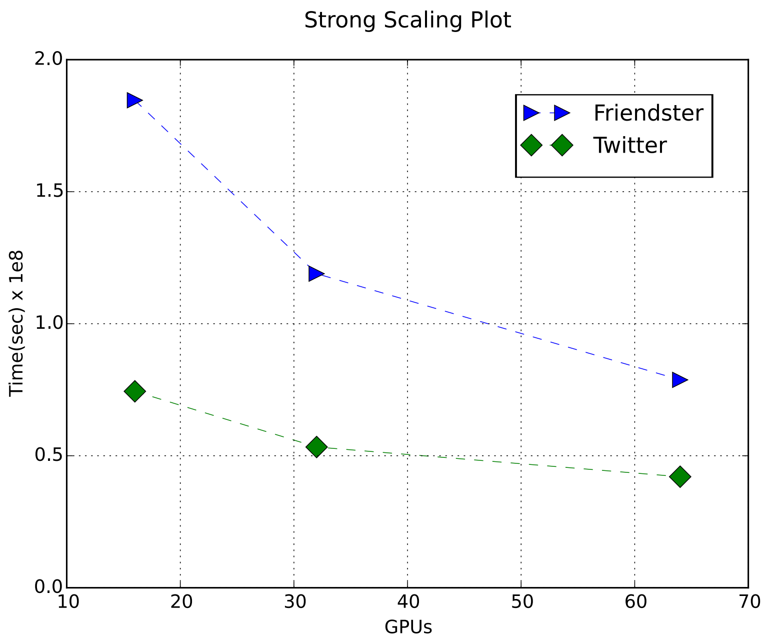


Figure 2.10. Strong scaling for Friendster and Twitter graphs.

the 1-degree preprocessing time (in parenthesis the time obtained without 1-degree reduction).

On a R-MAT graph with EF 16, the impact of the preprocessing with respect to the total time is less than 0.02% *vs.* a speed-up of 1.4 compared to the execution without 1-degree. A greater improvement is achieved when the edge factor decreases and consequently there are more degree-1 vertices. Indeed the speed-up of the 1-degree reduction strongly depends on graph characteristics, for instance com-Youtube graph has 53% of degree-1 vertices and the speed-up grows up to 2.9.

While it is not possible to compare our 1-degree reduction results with those in [65] since we are using different platforms, from a detailed analysis we observed that for the RMAT graph with Scale 20 EF 16 the BFS depth decreases by 1 for 99% of the vertices and, if we simply skip execution of BC algorithm rooted from degree-1 vertices, the total time is 10% longer than

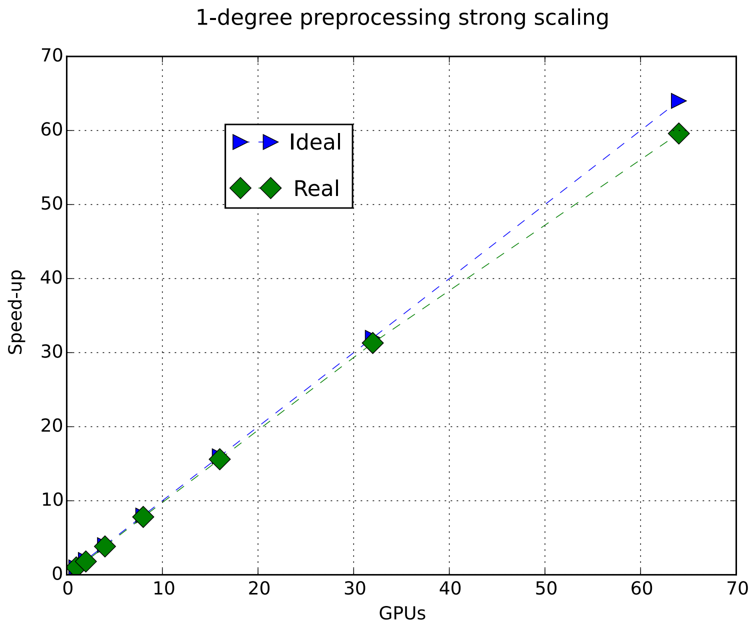


Figure 2.11. Preprocessing: strong scaling for R-MAT graphs with SCALE 22 and EF 16.

fully removing degree-1 vertices.

Graph	**EF**	**1-degree**	**Total time(hour)**	**Mean time(sec)**	**Preprocessing(sec)**	**Speed-up**
R-MAT	4	13%	1.06(1.86)	0.012(0.014)	0.312	1.7
R-MAT	16	13%	3.01(4.28)	0.021(0.023)	1.283	1.4
R-MAT	32	12%	5.38(7.29)	0.030(0.035)	2.449	1.4

Table 2.5. Impact on BC due to 1-degree reduction on RMAT graphs, running the full BC on 4 GPUs in a 2x2 mesh (in parenthesis time obtained without 1-degree reduction).

2.3.4.3 Sub-Clustering

While a Multi-GPU implementation makes possible to handle very large graphs, the overall time required can still be quite long. However, by combining coarse- and fine-grained parallelism a substantial time reduction can

Graph	1-degree	Total time(hour)	Mean time(sec)	Preprocessing(sec)	Speed-up
com-Youtube	53%	1.46 (4.23)	0.0098 (0.013)	0.62	2.9x
roadNet-CA	16%	40.6 (49.5)	0.089 (0.090)	0.55	1.2x
com-DBLP	14%	0.41 (0.51)	0.054 (0.058)	0.19	1.2x

Table 2.6. Impact on BC processing time due to 1-degree reduction on real graphs, running on single GPU (in parenthesis time obtained without 1-degree reduction).

be obtained. For these experiments we ran a full BC calculation exploiting the 1-degree reduction heuristic.

GPUs	Mesh	Time (hours)
1	1x1	≈ 258
64	2x1	≈ 6.5
128	2x1	≈ 3.5
256	2x1	≈ 1.7

Table 2.7. Total time to compute Betweenness Centrality for Orkut graph with different numbers of GPUs in a sub-cluster configuration.

Table 2.7 shows the total time required to compute BC for Orkut graph when the number of available GPUs increases: except for 1 GPU configuration, all other configurations uses a 2x1 sub-cluster processor mesh. For example in the 64 GPUs configuration, we are using 32 sub-clusters running concurrently. Note that by exploiting 256 GPUs the time drops to less than 2 hours. In order to achieve a perfect scaling, the sub-clustering technique requires that each sub-cluster had a balanced workload. The BC scores of local copies are accumulated for all of the GPUs on each sub-clusters. Finally, the scores at sub-cluster level are reduced into the global BC scores by a reduce operation. Since Orkut is composed by only one connected component, the workload among sub-clusters is pretty well balanced.

2.4 Discussion

In this chapter we discussed the Betweenness Centrality algorithm and we presented a Multi-GPU solution, to our knowledge this is the first distributed Multi-GPU implementation. This solution leverages state-of-the-art techniques to reduce computation and to achieve an optimal load balancing. Novel optimization techniques have been added to make possible reach high performance on both single and Multi-GPU systems, exploiting both fine- and coarse-grained parallelism, thus allowing to reduce the computation time of a 117 million undirected edges graph to less than 2 hours.

In order to furthermore reduce computation time, we are working on a 2-degree heuristic, more in detail if we suitably store intermediate results of graph traversal when starting from the two neighbor of a 2-degree vertex, then it is possible to compute the BC contributions from all three vertices executing only two dependency accumulation steps. With this heuristic instead of removing vertices, we reduce computation time by avoiding BC algorithm starting from 2-degree vertices.

Finally, we expect to release our code in the public domain to offer a tool able to compute BC on very large scale graphs.

Chapter 3

Large Data Forensic Analysis

For any large scale analysis of digital data, a first necessary step is the creation of an inverted index [68, 69] of all items present on the systems under scrutiny. The construction of an inverted index is a burdensome operation that can take many hours or even days depending on the size of the data and the available resources. Some previous work [70, 71] investigated on the benefits of using the MapReduce paradigm to build inverted indexes. First proposed in [72], MapReduce provides a simplified model to distribute workload over a cluster environment. It offers parallel processing of large amount of data by distributing work tasks over multiple processing machines. The core idea is that the overall processing algorithm can be split into many smaller operations, more in detail it can be divided in a *map* operation, which is a simple function over each record of the dataset emitting key/-value pairs, followed by a *reduce* operation, which collects the outputs of the map operations and merges them to build the desired result. Both map and reduce operations can be distributed over different machines.

Apache Hadoop [73] is the most popular open source implementation of the MapReduce paradigm. Hadoop requires the Hadoop Distributed File System (HDFS) to store input datasets, intermediate data, and final results. HDFS provides both high availability and data distribution, indeed data are divided in chunks, stored on local disks on each processing node and replicated so the task can be executed on another node holding a replica, if a node fails.

Hadoop and MapReduce provide significant benefits in terms of data reliability, simplified workload distribution and use. Moreover, Hadoop performs well when input data to map operation are already distributed. However, when indexing disk image files, before unleashing the full power of the framework, it is required to split and distribute the data source so that multiple map operations can run in parallel.

Hereafter we present a more cost-effective solution for analyzing large unstructured datasets that enables the user to quickly locate keywords in files stored in disk image files. The system allows the user to narrow down searches according to time-stamps, documents size, types and other attributes.

We rely on High Performance Computing (HPC) techniques to index and search huge amount of data that may include (but are not limited to) e-mails, documents, plain text files, web pages, etc. Our approach is scalable so, if more computing resources are available, the time needed to create the index decreases accordingly.

As proof-of-concept we implemented a system for indexing and searching digital forensics data that may be immediately used by an investigator. This may represent a real breakthrough since, in many investigations, time is a key factor.

This chapter is organized as follows: after a short review of related work (Section 3.1), in Section 3.2 we describe the problem of text searching; in Section 3.3 we illustrate the architecture of our solution and its software components; in Section 3.4 we focus on how data are processed by the first three components of our architecture which form the *Extract-Parse-Index* pipeline; in Section 3.5 we describe the optimizations applied during the indexing stage. Section 3.6 describes ISODAC, the *proof-of-concept* tool we developed for digital forensics investigation purposes. Finally, in Section 3.7 we discuss the results current and future perspectives.

3.1 Background and related work

Even if the proof-of-concept tool we present is focused on the digital forensics area, we propose a general solution for indexing and searching of heterogeneous textual data. Therefore, in this Section we report related work on Apache Hadoop, in-memory computing, and results previously obtained in digital forensics research.

3.1.1 Apache Hadoop

The Apache Hadoop[a] is an open source framework for distributed processing of large datasets. Hadoop mainly consists of MapReduce and HDFS modules, inspired by Google's work. Furthermore, several projects are built on top of Apache Hadoop like Flume, HBase, Hive, Mahout, Pig, Spark and others.

Hadoop performance analysis is an emerging topic. We can group work on MapReduce performance and HDFS performance in two categories.

The first category focuses on identifying the factors that affect Hadoop performance. Jiang et al. [74] carried out an in-depth study identifying five factors that affect MapReduce performance: I/O mode, indexing, data parsing, grouping schemes, and block-level scheduling. This study investigated alternative methods for tuning those factors showing that the overall performance of Hadoop can be improved by a factor between 2.5 and 3.5 for the same benchmark used in [75]. In [76], Zaharia et al. showed that Hadoop's scheduler may cause a severe performance degradation in heterogeneous environments, so they designed a new scheduling algorithm, called Longest Approximate Time to End, that is highly robust to heterogeneity. In [77], Rao et al. studied several issues that affect Hadoop performance at different levels (Cluster Hardware Configuration, Application logic related, System Bottlenecks, and Resource Under-utilization) and provided some guidelines on how to overcome these bottlenecks. In [78], Shafer et al. analyzed the per-

[a] `http://hadoop.apache.org`

formance of HDFS and uncovered several issues: architectural bottlenecks, portability limitations, and portability assumptions. In [79], Xue et al. proposed a performance monitoring tool for Hadoop Cluster named Hadoop Monitor. In [80], Lin et al. addressed an inefficient aspect of Hadoop-based processing: the need to perform a full scan of the entire dataset and showed that is possible to leverage a full-text index to optimize selection operations on text fields within records.

The second category of studies is oriented to build performance models for analyzing and optimizing Hadoop performance. For example, Lin et al. [81] provided an accurate performance model for Hadoop MapReduce. They defined the complexity metrics of Standard Process and Relative Computational Complexity to easily estimate the cost of Map function or Reduce function. In [82], the relationship between file size and HDFS Write/Read throughput, i.e., the average flow rate of a HDFS Write/Read operation, is studied to build HDFS performance models from a systematic view. For simplicity, that study focused on the Single-Input Single-Output situation instead of the more complex Multiple-Input Multiple-Output situation typical of real-world scenarios.

Finally, Mishne et al. [83] presented a case study in which they replaced a Hadoop-based system because it did not meet the latency requirements necessary to generate meaningful real-time results in Twitter. The context of their work is related to query suggestion and spelling correction in Twitter search (search assistance). They solved the problem implementing a new custom in-memory processing engine. That experience shows how Hadoop may have troubles in the low-latency processing of big data.

3.1.2 In-Memory Computing

MapReduce [72], Dryad [84], and Ciel [85] are examples of data flow models for commodity clusters that provide data sharing abstractions through permanent storage systems. For any kind of application with low-latency requirements, they work inefficiently due to the cost of data replication, I/O and serialization. To overcome those issues, several emerging cluster computing frameworks resort to in-memory processing. For example, Pregel [86] and HaLoop [87] are systems that provide high-level interfaces for iterative graph applications and iterative MapReduce runtime respectively. In particular, HaLoop extends MapReduce leveraging two simple intuitions: first it caches the invariant data to reduce the I/O cost for loading and shuffling them in subsequent iterations; then it caches and indexes the local output for each reducer.

Piccolo [88] provides a data-centric programming model for writing parallel in-memory applications across multiple systems. In Piccolo, users can run parallel functions using a set of in-memory tables whose entries reside in the memory of different nodes.

M3R (Main Memory Map Reduce) [89] is a new implementation of the Hadoop MapReduce API targeted at on-line analytics on high mean-time-to-failure[b] clusters. M3R focused on in-memory execution by storing key-value sequences in a family of long-lived JVMs, sharing heap-state among jobs. It does not offer resilience, so it fails if any node goes down.

S4 (Simple Scalable Streaming System) [90] is a distributed stream processing engine inspired by the MapReduce model. It uses a decentralized and symmetric architecture and minimizes latency using local memory on each processing node to avoid disk I/O bottlenecks. In S4, computation is performed by Processing Elements and messages are transmitted among them in the form of data events.

Most of the cluster programming models can be expressed efficiently using Resilient Distributed Datasets (RDDs) [91, 92] which are an abstraction for performing in-memory computations on large clusters. The RDDs abstraction aims at enabling efficiently two types of applications: those that reuse intermediate results across multiple computations (i.e., iterative machine learning and graph algorithms) and interactive data mining. RDDs

[b]Mean Time To Failure (MTTF) is the time a system is expected to last during operation.

are fault-tolerant, parallel data structures that allow users to control both in memory persistence of intermediate data results and data partitioning. RDDs provide an interface based on coarse-grained transformations that apply the same operation on multiple data items. This is different from existing abstractions based on fine-grained operations because RDDs provide efficiently fault-tolerance logging only the transformations used to derive an RDD from other datasets (its lineage). RDDs have been implemented in Apache Spark[c], an open source cluster computing solution that aims at speeding up data analytics. It was initially started as a research project at UC Berkeley in the AMPLab[d].

In recent years, Spark successfully inspired other projects. In [93, 94], Zaharia et al. proposed a new processing model, *discretized streams* (D-Streams), as an extension to the Spark framework called Spark Streaming, which allows users to seamlessly intermix streaming, batch and interactive queries. D-Streams enables a parallel recovery mechanism that improves efficiency over traditional replication and backup schemes, and tolerates stragglers. Shark (Hive on Spark) [95, 96] is built on top of RDDs memory abstraction. It is a new data analysis system that merges query processing together with complex analytics on large clusters, and provides fine-grained fault recovery across both types of operations.

In [97], Gu et al. reported about exhaustive experiments to evaluate the system performance for iterative operations between Hadoop and Spark. They have proven experimentally that Hadoop has better performance than Spark when there is not enough memory to store newly created intermediate results.

Spark can run on clusters managed by Apache Mesos [98]. Mesos is a light-weight resource sharing layer that enables fine-grained sharing across

[c]`https://spark.apache.org`

[d]The AMPLab is a five-year collaborative effort at UC Berkeley, involving students, researchers and faculty from a wide swath of computer science and data-intensive application domains to address the Big Data analytics problem: `https://amplab.cs.berkeley.edu/`

diverse cluster computing frameworks, providing a common interface for accessing cluster resources. It is apparent how big data processing using commodity clusters has attracted many interests and new cluster computing frameworks will continue to emerge. In this context, Mesos represents a good solution for running multiple frameworks without either partitioning the cluster and run one framework per partition, or allocating a set of virtual machines to each framework.

However, all these frameworks perform an effective in-memory data processing for specific applications when data is already distributed, whereas we may operate in a different context, where data are not necessarily stored in a distributed (e.g., HDFS) environment.

3.1.3 Digital Forensics

Despite the growing size of seized data storage, high performance solutions for forensic analysis of large datasets have received limited attention. In [99–101] authors describe distributed digital forensic solutions but none of them provides document parsing and full text indexing capabilities.

Commercial forensic tools such as Forensics Toolkit (FTK)[e] and Encase Enterprise Edition[f] run on a single workstation, whereas Encase may also run in a distributed environment primarily to support remote investigation operations, rather than to distribute the burden of analysis.

In [102], Garfinkel describes a disk forensic tool for processing disk images by using The Sleuth Kit (TSK)[g], which works on a single workstation, whereas The Sleuth Kit Hadoop Framework[h] is a project that incorporates TSK into a Hadoop cluster.

Some recent work [103, 104] describe tools and solutions focused on analyzing forensic data stored in cloud systems, whereas we describe, in the

[e] `http://www.accessdata.com`
[f] `https://www.guidancesoftware.com`
[g] `http://www.sleuthkit.org`
[h] `http://www.sleuthkit.org/tsk_hadoop`

present work, a solution to distribute the work required for building the index of a large set of forensic data.

Finally, [105,106] deal with high performance tools to triage large volumes of forensic data, but they do not provide full text indexing capabilities.

3.2 Text searching over heterogeneous unstructured documents

A traditional search engine consists of three main components: a crawler, an indexer and a query processor. The crawler follows web links, downloads documents for further processing; the indexer processes the documents and builds a specific data structure that can be easily queried; the query processors receives queries from the user, process them and presents the results to the users.

Our solution for analyzing large unstructured datasets, does not download documents from the web, since the documents are instead stored already into the datasets, therefore the crawler process is substituted by two tasks: the first to extract files from the dataset and the second to parse documents into text files.

3.2.1 File extraction

If the disk image file has been created using the Unix `dd` command, a simple but effective way to extract files on Unix systems is to use a loop device, which is a pseudo-device that makes the image file accessible as a block device. However the solution is limited in scope since it works only for ISO files and `dd` created image files.

Another option consists in using TSK library which supports multiple disk image file formats and provides C/C++ functions to go through the image file and output both the raw files and their metadata (e.g., creation time, file name, path, size, . . .). The extraction of files from a disk image is basically a serial procedure, since it is necessary to walk through the file system allocation table and reconstruct each file using its disk blocks.

3.2.2 Document parsing

Extracted files need to be analyzed to determine what kind of information they contain, in particular they could include: only textual information (e.g., html, plain text, ...), metadata information (e.g., geo-localization coordinates for image files), both textual and metadata information (e.g., pdf, *Office* files, ...), or no readable information at all (e.g., binary data files). Before proceeding with the actual file parsing, the file format has to be detected by analyzing its content in order to select the most suitable parser. While parsing documents, it is possible to skip specific file types, e.g. executable files, by using filters. Nevertheless the file format needs to be detected before applying the filter.

Among many free and commercial tools and libraries able to parse document files, we chose Apache Tika[i] because it supports the most wide variety of file formats. Tika leverages specific parsers to detect file format and to extract both metadata and textual contents from documents.

3.2.3 Document indexing

The structure most commonly used for indexing is the *inverted index* (also known as *inverted file* or *posting file*). After document parsing, the extracted text and metadata are finally used to build an inverted index.

The inverted index is composed of two elements: the *vocabulary* and the *occurrences*. The vocabulary contains the collection of all the different words appearing in the text. Moreover, for each word in the vocabulary a list of all the text positions where the word appears is stored; the set of all those lists is called *occurrences* (Figure 3.1 shows an example of inverted index).

[i]`http://tika.apache.org`

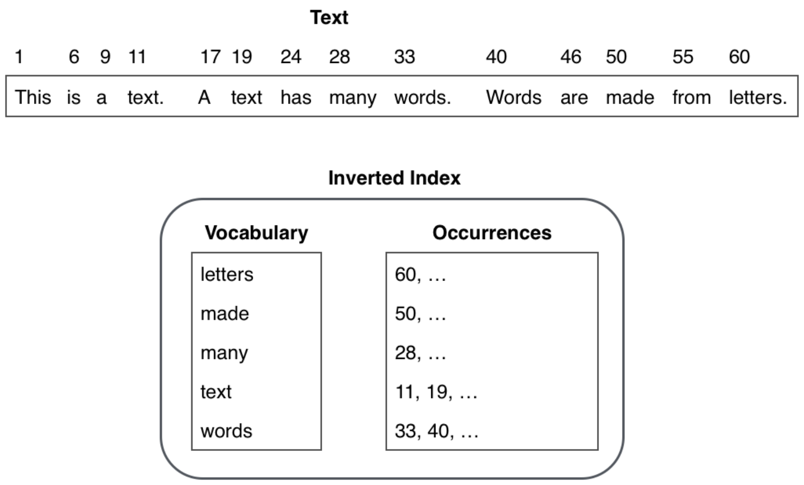


Figure 3.1. A sample text with the corresponding inverted index. The values in the occurrences point to the word positions in the text.

The text is analyzed through *a*) *Tokenization*, the document is split into a sequence of individual tokens; *b*) *Stemming*, each token is reduced to its stem; *c*) *Stop words removal*, articles, conjunctions of a given language (e.g., *and*, *in*, *the*, . . .) are removed from text; Then the vocabulary and the occurrences file are built using both the resulting set of tokens and the extracted metadata. Over the years many solutions, both commercial and free, have been developed for indexing and searching documents by building an inverted index. For instance, the Apache Lucene[j] library, one of the most widespread software for information retrieval, builds an inverted index from plain text documents.

File extraction, document parsing, and index building are executed sequentially in a pipeline that we call *EPI pipeline* (Extract-Parse-Index). To speed-up the operations in case of a large collections of documents, a parallel architecture should be employed to execute the EPI pipeline. Recently, the general purpose MapReduce paradigm has been often used for tasks like

[j]`http://lucene.apache.org`

building an inverted index and at a first glance it could be used for the EPI tasks as well. Nevertheless, we propose a different solution which turns out to be more efficient in building the inverted index specially from a disk image file.

3.2.4 Query processing

When a user submits a query, the inverted index is used to provide timely response, in particular the query processor receives the query, pre-process it, match it against the inverted index and provides back results to the user. The already mentioned Apache Lucene library provides functions that can be used to build up a query processor as well.

3.3 Architecture overview

Our solution for indexing and searching heterogeneous textual data runs on a hierarchical infrastructure (see figure 3.2) made by one Coordinator node and multiple Worker nodes. The nodes exchange control messages by using the ZeroMQ[k] API. As building blocks, we use the TSK library for raw files extraction from disk images, the Tika library for text and metadata parsing, and the Lucene library for text indexing and searching.

3.3.1 Coordinator node

Coordinator node is responsible for managing, coordinating and monitoring Worker nodes. It is logically divided into the following components:

- Job Scheduler: determines which Workers are available to execute the indexing of a disk image, allocates resources and starts required components;

[k] `http://zeromq.org`

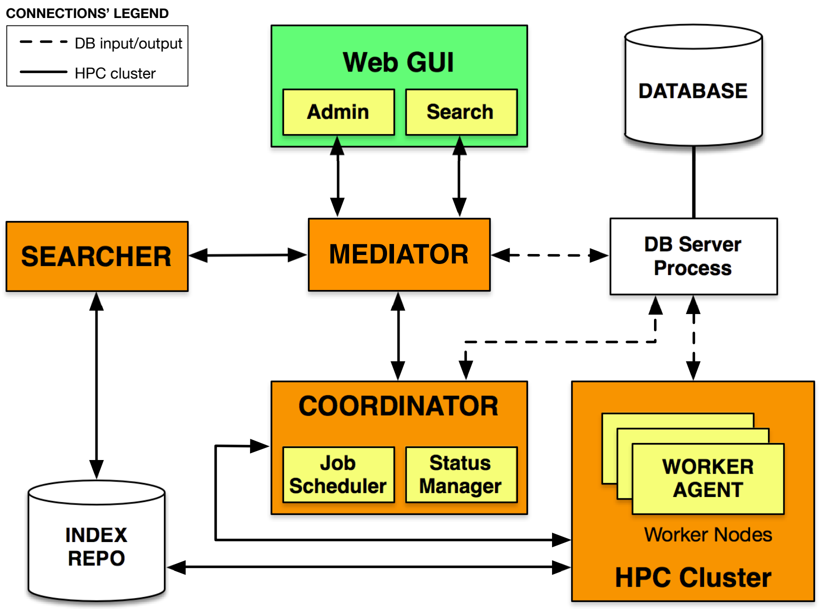


Figure 3.2. Architecture Overview.

- Status Manager: periodically monitors overall system status, in particular it checks when all tasks for a job are completed.

3.3.2 Worker nodes

Worker nodes respond to Coordinator commands carrying out the actual extraction and indexing work. Each worker node hosts one or more components coordinated by a Worker Agent, that is an agent daemon providing the following basic services:

- Heart-beat functionalities (for checking that the infrastructure works properly);
- Execution of commands issued by the Coordinator to start/stop components;
- Services for updating configurations and uploading results;

- Monitoring of components status: for each activated component, the Agent periodically checks whether it is working properly or not.

Each Worker node can execute one or more tasks using the following software components:

- Image-Extractor: extracts raw data and file metadata from forensic copies and sends them to Docu-Parsers;

- Docu-Parser: extracts plain text and document metadata from raw data and sends them to Docu-Indexers;

- Docu-Indexer: builds one or more indexes from text and metadata;

We now present these components in detail.

3.3.3 Image-Extractor

This component reads a disk image file, extracts raw files and sends them to Docu-Parsers. It works in fire-and-forget mode[1]: the Image-Extractor does not check if a Docu-Parser reports an error since error recovery is managed in the end by the Coordinator.

Image-Extractor is a process forked off by the Worker Agent. Its execution terminates after processing the whole image file. It can process all files within a single disk image or only a subset of those files as defined by the Coordinator.

Image-Extractor uses the TSK library to walk through the disk image and extract both raw files and file system metadata.

[1]*fire-and-forget* (*i.e.,* One-Way or In-Only) is a message exchange pattern for asynchronous communications. If the send operation (*fire*) completes successfully at the client end, it does not wait (*forget*) for a response from remote endpoint.

3.3.4 Docu-Parser

This component receives files that have been extracted by an Image-Extractor, it extracts plain text and file metadata and finally provides them to a Docu-Indexer along with file system metadata. This is a fire-and-forget operation as well, and the Coordinator manages error recovery in the end of the whole process. Docu-Parser is a Java process forked off by the Worker Agent.

The tasks carried out by the Docu-Parser are:

- reading a file and its metadata;
- extracting text exploiting the Tika library;
- sending file system metadata, plain text and metadata to a Docu-Indexer;
- logging parsing errors;
- providing error reporting to Worker Agent.

3.3.4.1 Docu-Indexer

This component receives parsed text and metadata from a Docu-Parser and creates a Lucene index for all documents received. This is also a fire-and-forget operation, and error recovery is managed by the Coordinator node. Docu-Indexer is a Java process forked off by the Worker Agent.

The Docu-Indexer is in charge of:

- reading the plain-text and metadata of each file;
- processing the text and metadata for indexing exploiting the Lucene library;
- logging processed files information and errors;
- providing error reporting to Worker Agent.

3.3.5 Docu-Searcher

This component is responsible for querying multiple Lucene indexes, presenting a single set of results to the user. Queries can be built according to the following query models:

- Disjunctive-keyword query: search for documents containing at least one of the words provided in the query;
- Conjunctive-keyword search: search for documents containing all the words provided in the query (the order of the words is irrelevant);
- Sentence search: search for documents containing all the words provided in the query (the order of the words is kept);
- Regular expression query: the query is interpreted as a regular expression and documents are returned if containing words that match the query (if the query is made of multiple words, they are managed with the same approach of the Sentence search);
- Range query: the user can query an index by providing a suitable range of values for the specific field (e.g., file size).

The user can browse query results and she can refine the search by applying a set of filters to prune the documents[m].

3.3.6 Database system

We use a database system to store information about:

- disk images and their indexes;
- infrastructure configuration and status;

[m]As an example, suppose that the query *foo* is submitted to an index and a set of documents *D(foo)* is returned; after a quick look at the results the user may decide to be interested only in files whose size is greater than a certain threshold.

- extraction, parsing and indexing jobs and their status.

We chose SQLite[n] as solution for database management due to its simplicity. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads from and writes directly to ordinary disk files.

Coordinator Node, Worker Agents and Mediator communicate with the Database system to store and retrieve information about indexed data.

3.3.7 WebGUI and Mediator

All the components and tasks just presented can be managed by using a Web-based user interface that is part of the proof-of-concept prototype described in Section 3.6. This user interface can interact with all the other components by means of the Mediator which is responsible to mediate requests from GUI to Searcher and Coordinator. The Mediator basically provides an abstraction layer between the framework and the other components.

3.4 Extract - Parse - Index

Image-Extractor, Docu-Parser and Docu-Indexer are the components of the EPI (Extract-Parse-Index) pipeline that provides a complete and reliable index of all documents stored within a disk image. They process the data stream in a pipe mode (see figure 3.3): Image-Extractor reads sequentially a disk image and, for each file, it extracts both file system metadata (creation date if present, last modification date, size, path, etc.) and raw data. Raw files and metadata are distributed round-robin among multiple Docu-Parser running in parallel. Image-Extractor keeps track of which files are sent to each parser. In this way, once the pipeline has been executed, it is possible to start a recovery procedure for those files whose indexing failed for any reason or that require special processing (e.g., OCR processing).

[n] `http://www.sqlite.org`

Docu-Parser, upon receiving raw file and its metadata, uses Tika library to extract additional metadata information and plain text, and finally it sends the output to a Docu-Indexer. Docu-Indexer, upon receiving file's textual content and metadata, uses Lucene library to build an inverted index.

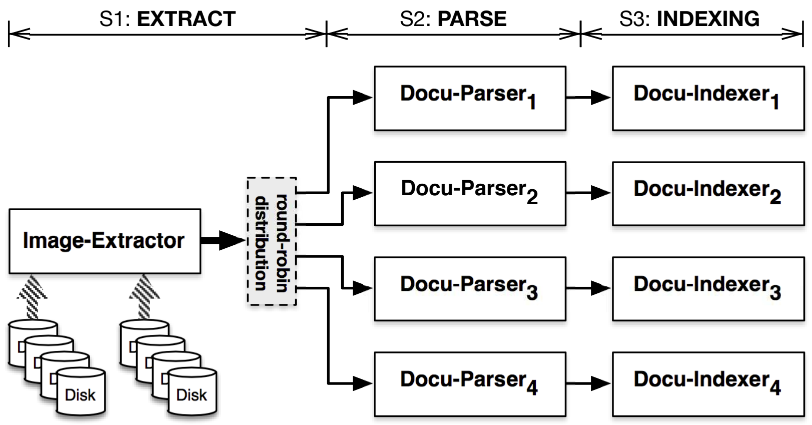


Figure 3.3. EPI Pipeline Overview. Image-Extractor extracts raw data from disk images and sends them in a round-robin fashion to each pipe.

EPI activities are managed, monitored and scheduled by the Coordinator that, based on user requests for disk images indexing, allocates required resources; in detail the Coordinator:

1. looks in the database for available computer nodes;
2. schedules an EPI pipeline job through the Job Scheduler;
3. updates workers configuration, starts and monitors EPI jobs.

3.4.1 EPI Pipeline Setup

The Job-Scheduler determines which resources to use to execute an EPI job. Currently all available Worker nodes are allocated on a best effort basis. After that, the Job-Scheduler adds a new Job to the DB and sends distinct messages to all worker agents involved, in particular:

- To Image-Extractor: job ID and optionally a list of fileIDs to extract, plus basic additional information;
- To Docu-Parser: job ID, Image-Extractor address to connect to and Docu-Indexer address to connect to;
- To Docu-Indexer: job ID.

Both Image-Extractor and Docu-Indexer accept stream connections from Docu-Parser: when all Docu-Parsers are connected to both ends of the pipeline, the indexing procedure starts.

3.4.2 Monitoring and Recovery

Since files are extracted from the Image-Extractor and sent over to Docu-Parsers and Docu-Indexers, something may go wrong causing the failure of the indexing for one or more files. In order to determine for which files the indexing failed, we employ the strategy described below.

Since a disk image file can include more than one file system, Image-Extractor always processes the disk image starting from the first file within the first file system until the last file within the last file system; during this phase, it keeps track of how many files are extracted from each file system.

Image-Extractor uses an incremental integer number, $fileId$ to count how many files are extracted; furthermore it stores the total number of files extracted from each file system within the same image file. In this way it is possible to calculate which file system contains a file, given its $fileId$.

Image-Extractor sends files to parsers according to a round-robin policy; each Docu-Parser P is assigned a number k from 0 to $N-1$, where N is the number of available parsers, furthermore is assigned a number O_k that corresponds to the order in which files are assigned to parsers within the round-robin loop, at the beginning $O_k = k$, therefore, for instance, parser number 2 is the third parser in a round.

For each parser P_k, the Extractor maintains an incremental counter C_k to keep track of how many files are sent; this counter is sent to P_k and represents the $localFileId$. Given a specific value M for the $localFileId$ on a parser P_k, it is possible to calculate its $fileId$ using the following formula:

$$fileId = M \times N + O_k \tag{3.4.1}$$

Docu-Parsers forward the $localFileId$ to Docu-Indexers. In this way the latter can determine which files are missing simply tracking if there are holes in the sequence of $localFileId$s received. When a job completes, each Docu-Indexer uploads the P_k identifier and the list of missing $localFileId$s to the Coordinator; the Extractor, in turn, for each P_k managed, uploads O_k and C_k values. Eventually the Coordinator, using Formula 3.4.1, determines the list of missing $fileId$s and schedules an additional job to retry the indexing process only for those files.

The solution described so far assumes that the number of Docu-Parsers does not change during job execution; unfortunately this is not always the case because parsers may stop working. To manage this situation, the previous solution has to be improved.

Each time a Docu-Parser stops working, a new *round* starts saving the current status of the EPI job (i.e., number of processed files, old parsers' order) and updating the round-robin schema accordingly to the number of remaining parsers.

More in detail, for each round i the Image-Extractor stores the following information:

- current number of extracted files ($sent[i]$);
- the number of remaining parsers ($N[i+1]$);
- the new parsers' order in the round-robin loop ($O_k[i]$);
- the total number of files sent to each parser so far ($C_k[i]$).

Note that:

- when Image-Extractor starts, $N[0]$ contains the number of parsers allocated to the Job;

- $N[i+1] < N[i]$ (we assume that the number of parsers can only decrease);
- the parsers' order O_k changes when a new round starts.

When a $localFileId = M$ is missing from parser P_k, then the $fileId$ can be calculated using the following formula:

$$\begin{aligned} &\text{Let } z = arg\,max_i : C_k[i] < M \\ &fileId = sent[z] + (M - C_k[z]) \times N[z+1] + O_k[z+1] \end{aligned} \tag{3.4.2}$$

3.4.3 Communication

Image-Extractor sends extracted files to Docu-Parser through a network stream organized according to the following format:

```
{<File_Metadata_1>}<Raw_Binary_Data_1>
{<File_Metadata_2>}<Raw_Binary_Data_2>
... ...
{<File_Metadata_N>}<Raw_Binary_Data_N>
```

File Metadata – written in JSON format – corresponds to metadata extracted from the image file or added by the Image-Extractor.

Docu-Parser sends extracted text and meta data to Docu-Indexer through a network connection by using the following format:

```
{<File-Start 1+ metadata >}
<TEXT 1>
{<File-End 1 + metadata }
{<File-Start 2+ metadata >}
<TEXT 2>
{<File-End 2 + metadata }
......
{<File-Start N + metadata >}
<TEXT N>
{<File-End N + metadata }
```

For example:

```
{
  "file_start": {
    "tika_uuid": "b904-e2dedf60711f",
    "imx_meta": {
      "imx:image_uuid": "D1810254CBAF",
      "imx:partition_id": "0",
      "imx:filename": "000578.pdf",
      "imx:pathname": "000",
      "imx:size": "351558",
      "imx:nlink": "1",
      "imx:atime": "2010-12-28T12:00:59Z",
      "imx:chtime": "2013-12-13T07:15:21Z",
      "imx:crtime": "1970-01-01T00:00:00Z",
      "imx:mtime": "2008-10-14T18:55:38Z",
      "imx:local_id": "0",
      "imx:pipe_id": "0"
    }
  }
}
EXTRACTED TEXT
{
  "file_end": {
    "tika_uuid": "b904-e2dedf60711f",
    "tika_meta": {
      "xmpTPg:NPages": "75",
      "Creation-Date": "2008-07-10T05:35:45Z",
      "meta:creation-date": "2008-07-10T05:35:45Z",
      "created": "Thu Jul 10 07:35:45 CEST 2008",
      "dcterms:created": "2008-07-10T05:35:45Z",
      "producer": "ECMP5",
      "xmp:CreatorTool": "VERSACOMP R05.2",
      "Content-Type": "application\/pdf"
    }
  }
}
```

Here:

- Extracted text is enclosed between `file_start` and `file_end` sections since we do not know how much time is required to extract the text before the extraction is completed;
- Metadata information are written both with the file start and file end: the first correspond to those sent by the Image-Extractor, the latter to those extracted by Docu-Parser which are available only when the whole file has been processed;
- `imx:local_id` and `imx:pipe_id` are sent to the Docu-Indexer.

3.4.4 Test Environment and Preliminary Results

We ran our tests using two clusters. The first is a *physical* cluster composed of four identical nodes, each one equipped with 4 six-core Intel Xeon X5650 CPU@2.67GHz, 48 GBytes RAM, CentOS 6.3, Management Network @ 100 Mbit/sec. (Fast Ethernet), dedicated Data Network @ 1 Gbit/sec. (Giga Ethernet). The second one is a *virtual* cluster composed of four virtual machines running on Amazon Cloud with 4 CPUs, 14GBytes RAM each and a dedicated Data Network with about 64 Mbit/sec. of measured speed.

We carried out a large set of preliminary experiments and analysis before starting the design of our architecture. First of all, we used the the Hadoop Framework and MapReduce model to run the EPI pipeline on the physical cluster. To do that, we had to preliminary copy disk image files into HDFS, a task often called *Data Ingress*. Using a Hadoop cluster with three Data Nodes and one Name Node, we found that uploading 64 GBytes of data takes about 35 minutes, whereas, we anticipate that our solution takes, on the same cluster, only 15 minutes to perform the whole EPI tasks.

After that, we built some disk images by using the Unix `dd` command over disk partitions containing documents downloaded from the *Digital Corpora website*[o]. Digital Corpora provides a corpus of (nearly) one million freely

[o]`http://DigitalCorpora.org` is a website of digital corpora for use in computer forensics education research.

Table 3.1. Disk image files used for testing: built using documents from Govdocs1 corpus, the first and second columns are the total size and number of files stored within the disk image, respectively, the third column is the time required to build the disk image file using the Unix `dd` command.

Image (GB)	Num Files	DD Time(hh:mm:sec)
32	58225	00:05:00
64	117282	00:14:50
100	186305	00:20:15
193	368856	00:33:21
296	586714	01:03:21

redistributable files named *Govdocs1* [107]. The corpus is available in several formats (ZIP files, subset "threads", and archive of JPEG files). We used the full version of Govdocs1 made by a set of 1000 directories, with 1000 files in each directory. The total size of the dataset is about 471 GBytes.

We evaluated Tika's performance against all Govdocs1's files, checking if files were correctly detected and parsed without any provision of metadata (*e.g.,* `Content-Type`). For that purpose, we used the simple miss ratio metric:

$$MISS = \frac{N_{err}}{N_{files}} \tag{3.4.3}$$

where $MISS$ is the Tika's miss ratio, N_{err} is the number of Tika errors in parsing files and N_{files} is the total number of files. We found that, using Tika 1.4 on Govdocs1 files, $MISS$ is $\sim 10\%$.

We used the disk images listed in Table 3.1 for our tests. The time to extract raw files and metadata from disk image files using one Image-Extractor is almost the same time required to build them from the disk image using the Unix `dd` command. Running one Docu-Parser on one computer of the physical cluster, we found it takes about 3 hours to parse extracted files and metadata from the 64 GBytes image. It appears that document parsing is the first bottleneck in the EPI pipeline. Therefore we decided to distribute the files extracted form one Image-Extractor over multiple Docu-

Table 3.2. Time required to extract, parse, and index disk image files: the second column is the time required for the whole EPI tasks, the third column is the size of text after parsing, while the fourth column is the total size of all the indexes.

Image (GB)	EPI Time	Out Text (GB)	Index (GB)
32	00:05:45	11	4
64	00:17:43	24	8.5
100	00:37:16	31	12
193	01:02:22	55	19
296	02:08:00	89	33

Parsers/ Docu-Indexer couples that run in parallel on multiple cores/systems as shown in Figure 3.3.

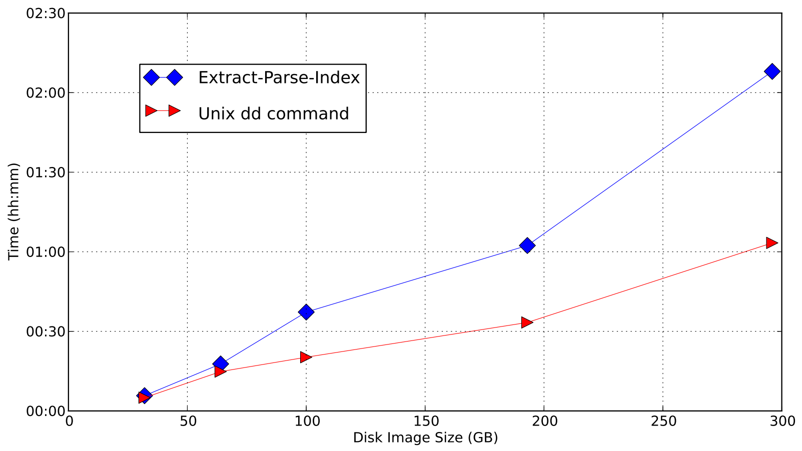


Figure 3.4. Time to Extract-Parse-Index compared to time to extract files using the Unix dd command.

In the end we processed the disk image files through a pipeline formed by one Image-Extractor and twelve Docu-Parsers/ Docu-Indexer couples running on the physical cluster, where three nodes ran 4 instances of the Docu-Parser/ Docu-Indexer couple each, whereas one node ran the Image-Extractor; results are shown in Figure 3.4 and Table 3.2.

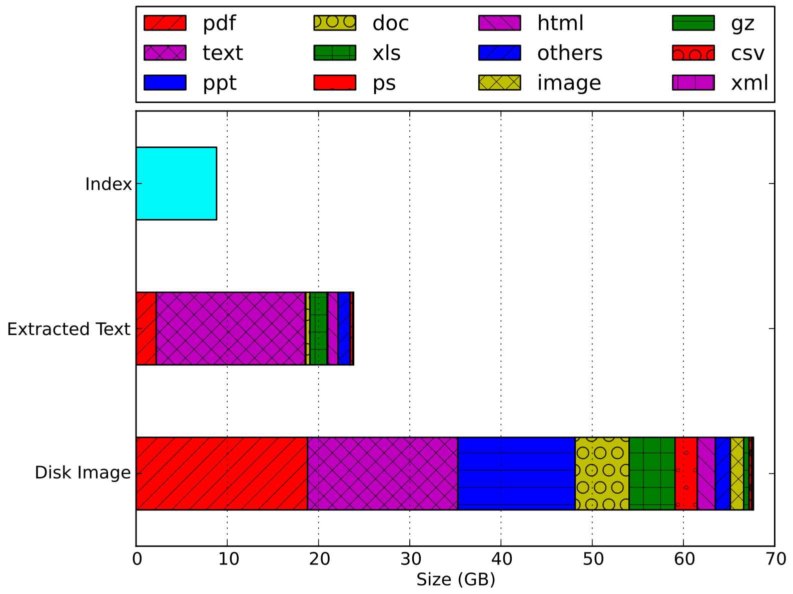


Figure 3.5. Distribution of data in the 64 GB disk image and the size of the corresponding extracted text and index.

3.4.5 Comparison with MapReduce based solutions

An interesting project that exploits both Hadoop parallel processing and TSK library capability in extracting raw files from disk images, is the TSK for Hadoop Framework.

The project, developed by Basis Technology, 42Six Solutions and Lightbox Technologies, aims at building an inverted index from a disk image file. Since this is exactly our goal, we downloaded the code available on GitHub, applied some changes and fixes, and finally tried to run it. To run TSK for Hadoop, first of all, we had to copy the disk image file into HDFS, this task requires too much time (about 35 minutes for a 64GB file), therefore we gave up with this approach.

After that we decided to write into HDFS documents extracted from disk image files and use MapReduce to run in parallel both document parsing and indexing. Documents extracted from a disk image file are streamed into

Hadoop using Flume (version 1.5) and, as soon as documents are available, map/reduce tasks start to parse and index them.

Both our ISODAC solution (see Section 3.6) and the Hadoop based solution were executed on the virtual cluster: for Hadoop we used one machine to run the Name Node and three to run the Data Nodes, whereas for the ISODAC framework we used one machine to run the Image-Extractor and three to run two Docu-Parser/ Docu-Indexer couples each (for a total of six pipes).

On the machine running the Image-Extractor or the Name Node, two disk image files, one of 40GB and the other of 100 GB, were available, both containing documents from the Govdocs1 library.

ISODAC Framework was able to build the index in 20 and 45 minutes respectively, whereas by using the Hadoop Framework it took about 38 minutes (for the 40 GB image) and 5 hours (for the 100 GB image). One of the major issues of the Hadoop solution is that Flume does not support nested directories. In that case, all files included in the image have been copied in a single directory and this configuration slows down dramatically the processing. Using a custom version of Flume it should be possible to overcome that limitation. Preliminary tests show that the time required for indexing 100 GB should reduce to 1.5 hours that remains much higher ($\sim$ a factor 2) with respect to the time obtained by using our solution.

Our ISODAC Framework performs better that Hadoop because both file extraction and parsing is performed in-memory without any time consuming disk write operation. In the Hadoop framework intermediate results (like the output of the file extraction or the result of the file parsing) are stored into the HDFS. This feature of Hadoop may be considered an advantage only if it is required to access again the output of the parsing phase.

In all situations where the final output consists only in the inverted index, our solution is much more efficient. Furthermore, storing intermediate results may become an issue when dealing with large disk images since it would require having as much disk space as the original disk image (consider also that the default configuration of HDFS replicate data 3 times). Figure 3.5 shows a comparison between the space used by a 64 GB disk image (bottom bar) and the space required for text extraction (intermediate bar) and indexing (top bar).

Building the inverted index through in-memory processing, without writing intermediate results to disk, saves both disk I/O time and space, but it makes the solution both network and memory bound. To reduce those burdens we added a buffer on each Docu-Parser that accumulates data from Image-Extractor, the buffering technique has been implemented exploiting Java Unsafe memory allocation to reuse memory and to avoid intensive Garbage Collection.

3.5 Optimizations

3.5.1 Buffering between Image-Extractor and Docu-Parser

The most time consuming task in the EPI pipeline is document parsing, which transforms input document files into plain text and metadata. The time to parse a document depends both on its size and type, indeed parsing a PDF or *Word* file requires more time than parsing an HTML or text file. The Image-Extractor extracts files from the disk image and distributes them over multiple Docu-Parsers through a dedicated network stream for each parser; if a parser consumes data at a rate slower to the rate by which the Image-Extractor sends the data, then the network pipe fills up and the whole indexing process slows down. Unfortunately this situation happens frequently because files are distributed evenly among parsers, but the time to parse each document may vary a lot. To reduce this issue, on each Docu-Parser we added a buffer where data are written as soon as they available from the network stream. The parser reads data from that buffer when it is ready for the next text extraction.

The buffer has been implemented using Java classes `PipedInputStream` and `PipedOutputStream`, but in order to overcome the JVM limitation of 2GB of memory allocation and to make memory management more efficient, we re-implemented those two classes using `sun.misc.Unsafe` class that allows the allocation of memory by-passing the JVM and the Garbage Collector.

3.5.2 CUDA tokenizer

Since extraction and parsing are typically I/O bound operations, we employed GPU devices to improve the indexing process, in particular the tokenization phase[p].

We employ CLucene [108], a C++ port of (Java) Lucene search library that represents a widely used solution for high-performance, scalable indexing and searching textual data. To fully understand its computational cost, we started with an accurate analysis of CLucene algorithms by studying the source code and by using several profilers[q]. Profiling allowed us to understand where CLucene spends its time and to define the execution flow during textual data indexing. We profiled and measured CLucene executions by using different configurations against various datasets. We found that even for indexing, the most time-consuming functions are related to I/O operations, about 60% of total time. The remaining time is used as follows: $\sim$ 32% for text tokenization and analysis, $\sim$ 7% for indexing and merging of partial indexes, and less than 2% for other operations (see fig. 3.6). Therefore, we invested in efforts to develop a new efficient tokenizer by exploiting the parallelism and the huge computational power provided by Graphics Processing Units (GPUs). CLucene scans character by character all the input text looking for tokens. When a token is found, a new token object is created. The

[p]Tokenization is the process of breaking a text into meaningful elements called tokens. Tokens may be words, phrases, acronyms, e-mail, etc depending on type of analyzer used during tokenization

[q]We used three perfomance analysis tools: GNU gprof, Oprofile, and Callgrind (profiling tool of Valgrind)

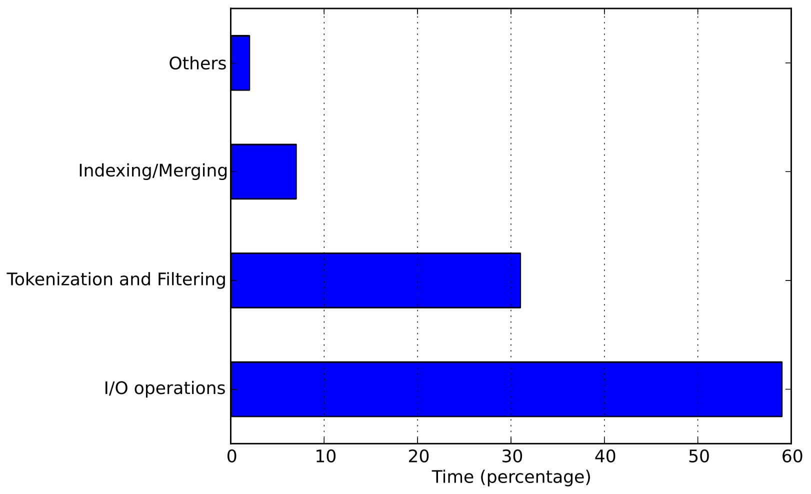


Figure 3.6. CLucene Profiling

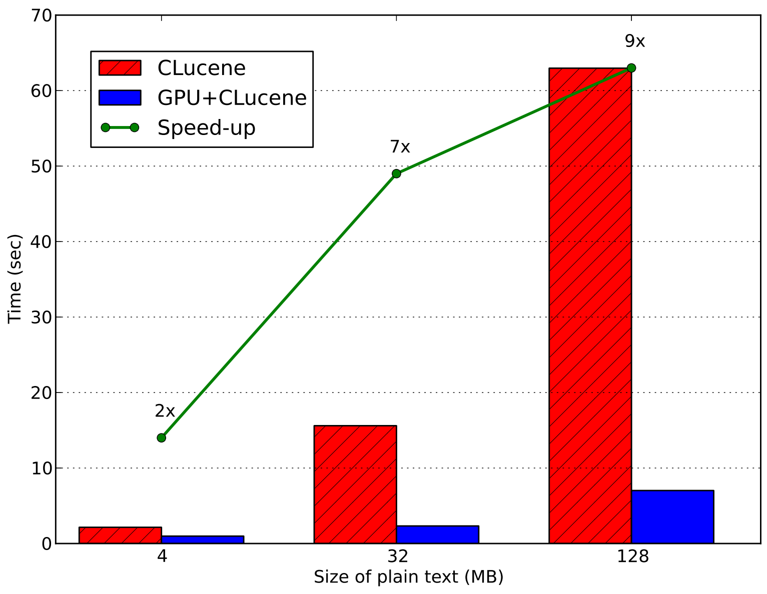


Figure 3.7. Speedup

new token passes through different filters and then it is added to the inverted index.

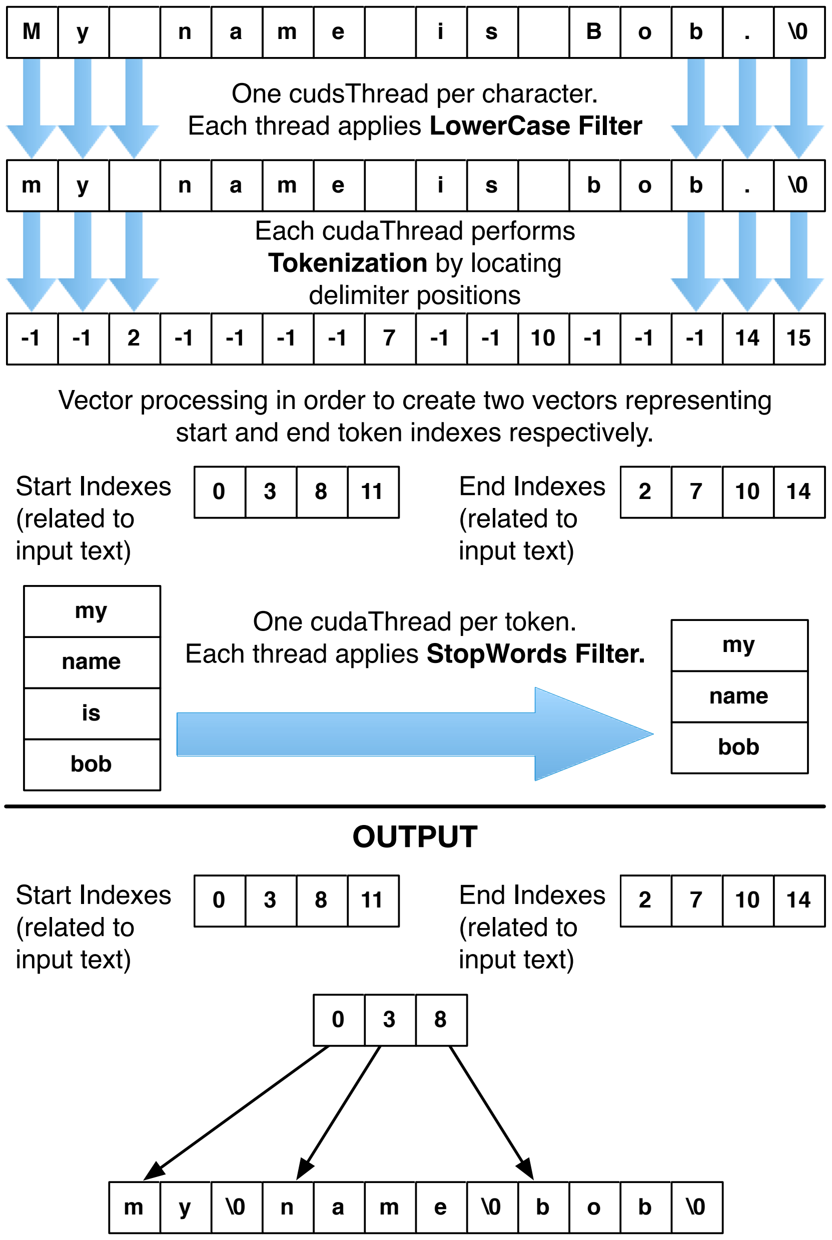

M y n a m e i s B o b . \0
One cudsThread per character.
Each thread applies **LowerCase Filter**
m y n a m e i s b o b . \0
Each cudaThread performs
Tokenization by locating
delimiter positions
-1 -1 2 -1 -1 -1 -1 7 -1 -1 10 -1 -1 -1 14 15
Vector processing in order to create two vectors representing
start and end token indexes respectively.
Start Indexes (related to input text)
0 3 8 11
End Indexes (related to input text)
2 7 10 14
my
name
is
bob
One cudaThread per token.
Each thread applies **StopWords Filter.**
my
name
bob
OUTPUT
Start Indexes (related to input text)
0 3 8 11
End Indexes (related to input text)
2 7 10 14
0 3 8
m y \0 n a m e \0 b o b \0

Figure 3.8. CudaStandardAnalyzer: Tokenization And Filtering On Sample Data

As we mentioned above, CLucene spends most of its computing time looking for tokens. The tokenization process can be parallelized, so we implemented two solutions in CUDA, one of the most widely used solutions for programming GPU, in order to improve tokenization and analysis processes carried out by CLucene. We focused our work on the StandardAnalyzer and related StandardTokenizer, defined in CLucene, for the English language. This analyzer is the most widely used. The first solution, called CudaStandardTokenizer (CudaST), performs on GPU only the tokenization process. This solution supports plain-text files up to 2 Gbytes and provides a speed-up up to 6 times compared to the original CLucene. The second solution, called CudaStandardAnalyzer (CudaSA), performs both tokenization and analysis processes on GPU. This approach requires a larger amount of memory (host and device), but it allows for reaching a speed-up to 9 times (see fig. 3.7) Figures 3.8 shows an example of tokenization and analysis against sample text performed by our second solution.

3.6 The ISODAC tool

In this section we present the ISODAC (Indexing and Searching Of Data Against Crime) tool, a prototype based on the architecture described in Section 3.3 devoted to forensic investigation purposes that is able to:

- extract raw files and metadata from forensic disk images;
- distribute those files over multiple computer systems;
- extract from every file plain text and metadata;
- index resulting text and metadata;
- provide a user interface to query indexed data;

The tool supports two different class of users: *Administrator* and *Investigator*, for both a Web Interface is available.

The main task of an Administrator is to set up investigation cases and assign to each of them one or more disk images to be indexed.

While the main task of an Investigator is to search for evidences in the seized disks.

3.6.1 Disk image add

In this use case, a new disk image is added for investigation. The disk image file is directly available in a disk that is local to a particular Worker node, or in a shared storage system, so it can be read by one or multiple Worker nodes. When that happens the following operations are carried out:

1. A Worker node detects that a new disk image file has been uploaded, reads file information and sends them to Coordinator node;
2. Coordinator stores image file information into the DB and schedules a Job, in particular:
    (a) determines one Worker to execute Image-Extractor;
    (b) determines one or more Workers to execute Docu-Parser;
    (c) determines one or more Workers to execute Docu-Indexer;
3. The Coordinator sets up the EPI pipeline and starts the Job;
4. When Job completes, information on Job execution is updated into the DB system and the disk image is available for searching.

3.6.2 Investigation cases

Since forensic investigations typically include analysis on multiple disks, it is possible to group together multiple disk image files. Later, the user can decide to search in a particular disk or in a set of disks. To add an investigation case the user connects to the UI and:

1. inserts the title and additional information;

2. looks at disk images available in the UI and assigns them to the investigation.

Later on, the user can search over all disk images included in the case.

3.6.3 User search

Docu-Indexers produce multiple indexes distributed over the Workers. Therefore the Coordinator needs to know where all indexes related to a disk image are stored. When a user, through the Search UI, submits a query related to an investigation case, the system:

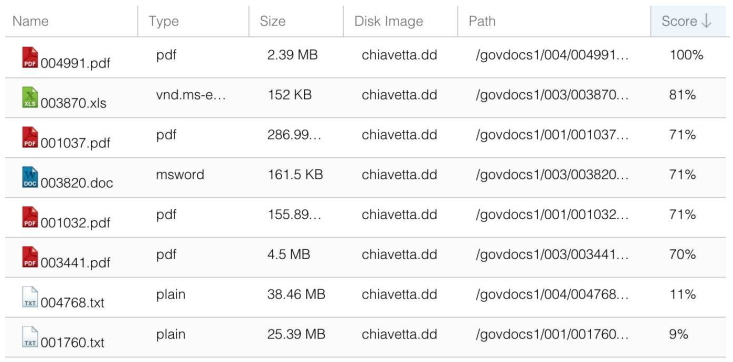
1. queries the DB to determine where all indexes related to the investigation/disk image are stored;
2. submits the query string to each index distributed over Workers;
3. merges query results;
4. returns results to the user.

3.6.4 Results visualization

When an investigator submits a query to the system, the search results should be presented in a way that makes it possible to quickly identify the most relevant documents and, possibly, to interact with displayed data. Figure 3.9 shows the first visualization option we implemented: here the results are displayed as a list of documents. Each column represents an attribute of the correspondent file which can be used to sort the list. Even though presenting results in a flat list it is certainly an efficient option when a user needs to browse documents in a given order (e.g., lexicographical, by score value, etc.) it is also important to take into account the response time and usability of the interface.

The visualizations provided should be interactive, to enable the investigator performing directly further operations on data. The same data should also be displayed in several layouts to highlight their different aspects. Finally, it is necessary to provide multiple filters for each visualization, to offer the chance of a personalized interaction with the results.

For this reason we developed an alternative visualization layout relying on D3.js [109], a JavaScript library which provides several graphical primitives to implement visualizations and uses only web standards, namely HTML, SVG and CSS. With D3.js[r] it is possible to realize multi-stage animations and interactive visualizations of complex structures.



Name	Type	Size	Disk Image	Path	Score ↓
004991.pdf	pdf	2.39 MB	chiavetta.dd	/govdocs1/004/004991...	100%
003870.xls	vnd.ms-e...	152 KB	chiavetta.dd	/govdocs1/003/003870...	81%
001037.pdf	pdf	286.99...	chiavetta.dd	/govdocs1/001/001037...	71%
003820.doc	msword	161.5 KB	chiavetta.dd	/govdocs1/003/003820...	71%
001032.pdf	pdf	155.89...	chiavetta.dd	/govdocs1/001/001032...	71%
003441.pdf	pdf	4.5 MB	chiavetta.dd	/govdocs1/003/003441...	70%
004768.txt	plain	38.46 MB	chiavetta.dd	/govdocs1/004/004768...	11%
001760.txt	plain	25.39 MB	chiavetta.dd	/govdocs1/001/001760...	9%

Figure 3.9. The results of a search can be presented as a list

After the results for a specific query are collected, data are organized in a JSON stream containing the following attributes:

- *name:* the name of the file;
- *path:* the path where the file is stored;
- *diskimage_id:* ID of the disk image containing the file;
- *type:* the file format;
- *size:* the size of the file;

[r]Technical documentation, as well as source code of D3.js, can be reached at `http://d3js.org`

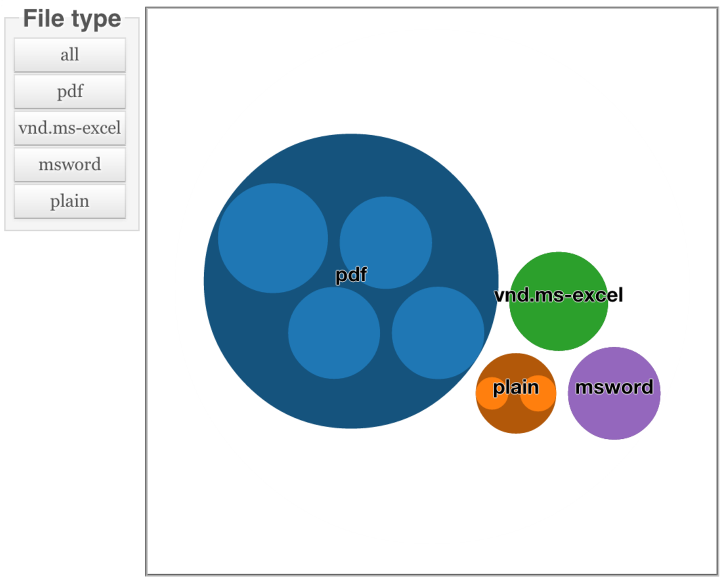
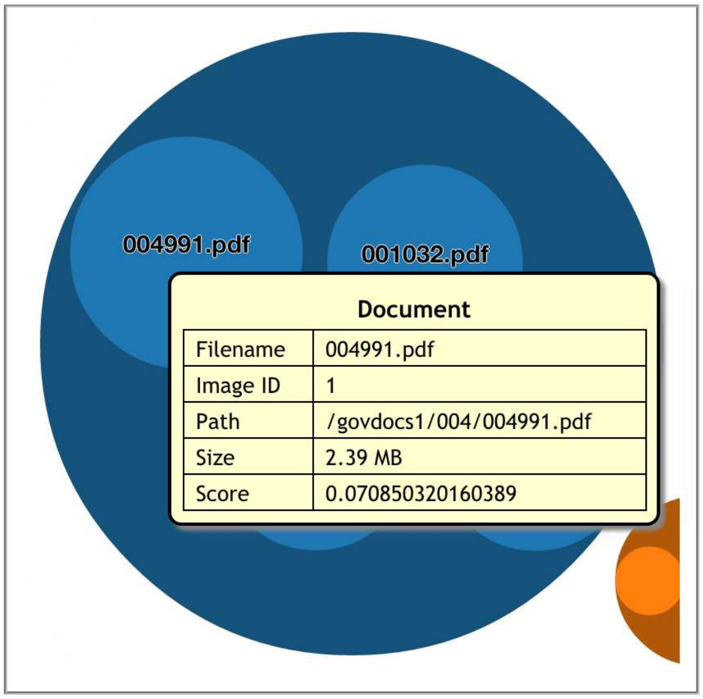


Figure 3.10. User Interface screenshot with the results of a search

- *score:* the score that Lucene assigns to the document according to the search query.

Using this schema our system is able to produce, for instance, the layout shown in Figure 3.10. The results correspond to a search that returns 8 documents divided in five different file types. Each type is represented by a circle of a specific color, as an example the blue one contains all the pdf documents, so that users can quickly focus on a subset of files. The diameter of the circles is proportional to the overall size of the documents of that specific file type (e.g., the size of `pdf` files is more than 50% of the total size of the results). The lighter and smaller circles depict the single documents of each given type and their diameter is proportional to their size. This representation can help a user to identify that half of the files (i.e., exactly 4 files) are `pdf` documents; if one looks at the size of raw data, the `msword` circle contains approximately the same amount of data than the `plain text` one, regardless of the higher number of files contained in the latter category.

If the user is interested in a specific kind of data (e.g., only `pdf` files) she can click on the corresponding circle to zoom into the desired circle as shown in Figure 3.11. The names of the files are then displayed on top of the circles and the user can obtain more information (basically retrieved by accessing the corresponding metadata) on a specific file by moving the mouse over the corresponding circle.



Document

Filename	004991.pdf
Image ID	1
Path	/govdocs1/004/004991.pdf
Size	2.39 MB
Score	0.070850320160389

Figure 3.11. A zoom of the previous screenshot focused on `pdf` files

3.7 Discussion

We presented a solution for fast indexing of large sets of textual data whose main, but non exclusive, goal is the reduction of the time required for the analysis of seized storage devices or bodies of evidence.

We do not rely on Hadoop or similar frameworks since they usually perform well when data are already distributed whereas significant I/O issues affect the performance of the whole process if ingestion of large non distributed data has to be carried out. Conversely our work aims at providing an efficient and effective solution to index data that are not already distributed, by exploiting in-memory processing.

More in detail, we implemented a new framework that performs streamed in-memory processing writing on disks only the final results of the Extract Parse Index pipeline. Our recovery mechanism is able to manage both single file failures and components failures. The same mechanism supports ad-hoc processing of special files (e.g. files requiring OCR). We introduced two different optimizations to enhance data communication and text analysis. In particular, the latter represents, to the best of our knowledge, the first work that exploits the GPU computational power to enhance performance of tokenization and text analysis. Finally we present some experimental results obtained by using the proof-of-concept tool we developed for digital forensics purposes.

In the near future we expect to test our framework with other sources of data besides disk images, for instance data produced by web crawlers.

Chapter 4

Conclusions

In the first part of this dissertation we presented three algorithms and their implementations on a Multi-GPU system.

First of all, starting from an already existing Breadth First Search Multi-GPU implementation, we improved both the data structures and the communication patterns used, in particular we modified the Compressed Sparse Row (CSR) data structure, used to store the graph, allowing to reduces both memory utilization and determine an upper bound on memory requirements. Furthermore we introduced specific communication patterns to exchange 32-bits integers instead of 64-bits and to exchange predecessors information only once at the end of the graph traversal. These optimizations allowed to perform 2 times faster than the previous implementation on a 64 GPUs cluster.

Then, our Multi-GPU BFS has been used to solve the ST-Connectivity problem. The proposed solution was implemented through two concurrent parallel BFS that are started from both terminals s and t. For this problem we provided two solutions to solve race conditions due to the parallel execution of the two searches: one using atomic operations and the other not using atomic operations. We found that the efficiency of the atomic primitives available using Kepler NVIDIA GPUs makes the solution with atomic operation performing better, some tests carried out on previous generation (Fermi) GPUs show that atomic operations may have a impact on performance when running on multiple GPUs, up to the point that, on the Fermi architecture, the solution not using atomic operations may provide better performance.

Comparing BFS and ST-CON results, we observed that GPUs are exploited efficiently when lots of vertices are processed in parallel so that the device is fully loaded, for R-MAT graphs this occurs between the third and the fifth BFS level. When the ST-CON problem is worked out by using two concurrent parallel searches on R-MAT graphs, the execution terminates after few iterations, therefore the efficiency of a GPU implementation remains quite limited.

Finally we took on the Betweenness Centrality, which is particularly interesting both for its relevance in many fields and for its computational complexity. In our Multi-GPU implementation we capitalize on knowledge and expertize acquired for BFS and ST-CON, nevertheless unlike the previous two implementations, we opted for a 2-D graph partitioning since it revealed to provide better performance and scalability. As far as we know this is the first implementation of BC over distributed Multi-GPUs, it takes advantage of multiple techniques to reduce overall computation time: fine- and coarse-grained parallelism, 1-degree reduction, pipelining of CPU-GPU data transfer with communication. All these techniques allowed to compute full BC scores for a 117 million undirected edges graph in less than 2 hours. Even if the main focus of our Multi-GPU BC was to analyze graphs too large to fit into one single system memory, our implementation on a Single-GPU system provides results comparable to other existing solutions.

In the second part of this dissertation we present a solution for fast indexing and searching of large datasets of heterogeneous documents whose main, but non exclusive, goal is the reduction of the time required for the analysis of seized storage devices or bodies of evidence.

Existing approaches are able to process huge amounts of data already distributed in a cluster environment, while provide limited performance when data is not distributed. Our tool improves, with respect to those solutions, by providing a suitable combination of HPC techniques. In particular, we implemented a new framework that performs streamed in-memory indexing, writing on disk only the final results. We developed a proof-of-concept tool for digital forensics purposes which has been successfully deployed at Raggruppamento Carabinieri Investigazioni Scientifiche (RaCIS)[a] and has been presented at the seminar "Cybercrime and terrorism threat in the Mediterranean Area" held at the Carabinieri Higher Institute for investigation techniques.

The work described in this dissertation has been published in the following papers:

- M. Bernaschi, G. Carbone, et. al., "Forensic disk image indexing and search in an hpc environment". In High Performance Computing & Simulation (HPCS), 2014 International Conference on, IEEE, 2014.
- M. Bernaschi, G. Carbone, E. Mastrostefano, F. Vella, "Solutions to the st-connectivity problem using a GPU-based distributed BFS". J. Parallel Distrib. Comput. (2014)
- M. Bernaschi, G.Carbone, et. al., "Enhanced gpu-based distributed breadth first search". In Proceedings of the 12th ACM International Conference on Computing Frontiers, 2015
- M. Bernaschi, G. Carbone, F. Vella, "Betweenness centrality on Multi-GPU systems." In Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms. ACM, 2015
- G. Totaro, G. Carbone, M. Bernaschi, et. al., "ISODAC: a High Performance Solution for Indexing and Searching Heterogeneous Data". Accepted at The Journal of Systems & Software.

[a]Department of Carabinieri Corps police performing scientific investigations.

Bibliography

[1] Ulrik Brandes and Thomas Erlebach. *Network analysis: methodological foundations*, volume 3418. Springer Science & Business Media, 2005.

[2] Stanley Wasserman. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[3] Luonan Chen, Rui-Sheng Wang, and Xiang-Sun Zhang. *Biomolecular networks: methods and applications in systems biology*, volume 10. John Wiley & Sons, 2009.

[4] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 25–, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Huiwei Lv, Guangming Tan, Mingyu Chen, and Ninghui Sun. Compression and sieve: Reducing communication in parallel breadth first search on distributed memory systems. *CoRR*, abs/1208.5542, 2012.

[6] Aydin Buluc and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[7] Enrico Mastrostefano and Massimo Bernaschi. Efficient breadth first search on multi-gpu systems. *J. Parallel Distrib. Comput.*, 73(9):1292–1305, September 2013.

[8] Massimo Bernaschi, Giancarlo Carbone, Enrico Mastrostefano, Mauro Bisson, and Massimiliano Fatica. Enhanced gpu-based distributed breadth first search. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 10:1–10:8, New York, NY, USA, 2015. ACM.

[9] Massimo Bernaschi, Giancarlo Carbone, Enrico Mastrostefano, and Flavio Vella. Solutions to the st-connectivity problem using a gpu-based distributed BFS. *Journal of Parallel and Distributed Computing*, 76:145 – 153, 2015. Special Issue on Architecture and Algorithms for Irregular Applications.

[10] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[11] A. L. Barabási, H. Jeong, Z. Néda, E. Ravasz, A. Schubert, and T. Vicsek. Evolution of the social network of scientific collaborations. *Phys. A*, 311(3-4):590–614, 2002.

[12] Hawoong Jeong, Sean P Mason, A-L Barabási, and Zoltan N Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, 2001.

[13] Valdis Krebs. Uncloaking terrorist networks. *First Monday*, 7(4), 2002.

[14] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[15] Massimo Bernaschi, Marco Cianfriglia, Antonio Di Marco, Alessandro Sabellico, Gianluigi Me, Giuseppe Carbone, and Giuseppe Totaro. Forensic disk image indexing and search in an HPC environment. In

High Performance Computing & Simulation (HPCS), 2014 International Conference on, pages 558–565. IEEE, 2014.

[16] V. Agarwal, F. Petrini, D. Pasetto, and D.A. Bader. Scalable graph exploration on multicore processors. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1 –11, nov. 2010.

[17] T. Oguntebi S. Hong, S. K. Kim and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, 2011.

[18] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[19] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[20] Koji Ueno and Toyotaro Suzumura. Highly scalable graph search for the graph500 benchmark. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 149–160, New York, NY, USA, 2012. ACM.

[21] Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, and Yogish Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12. IEEE, 2012.

[22] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. *Introduction to algorithms*. MIT press, 1990.

[23] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[24] Pawan Harish and P.J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and ViktorK. Prasanna, editors, *High Performance Computing HiPC 2007*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer Berlin Heidelberg, 2007.

[25] Scott Beamer, Aydin Buluc, Krste Asanovic, and Dean Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1618–1627. IEEE, 2013.

[26] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. *2012 41st International Conference on Parallel Processing*, 0:523–530, 2006.

[27] Sungpack Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78 –88, oct. 2011.

[28] Andrew Grimshaw Duane Merrill, Michael Garland. High performance and scalable gpu graph traversal. Technical report, Nvidia, 2011.

[29] Nadathur Satish, Changkyu Kim, Jatin Chhugani, and Pradeep Dubey. Large-scale energy-efficient graph traversal: A path to efficient data-

intensive supercomputing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 14:1–14:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[30] Koji Ueno and Toyotaro Suzumura. Parallel distributed breadth first search on gpu. In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 314–323. IEEE, 2013.

[31] Dan Zou, Yong Dou, Qiang Wang, Jinbo Xu, and Baofeng Li. Direction-optimizing breadth-first search on CPU-GPU heterogeneous platforms. In *High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC EUC), 2013 IEEE 10th International Conference on*, pages 1064–1069, Nov 2013.

[32] Yang You, David Bader, and Maryam Mehri Dehnavi. Designing a heuristic cross-architecture combination for breadth-first search. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 70–79, Sept 2014.

[33] Fabio Checconi and Fabrizio Petrini. Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 425–434, Washington, DC, USA, 2014. IEEE Computer Society.

[34] Brian W. Barrett James A. Ang Richard C. Murphy, Kyle B. Wheeler. Introducing the graph 500. 2010.

[35] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[36] Jure Leskovec. Stanford large network dataset collection (http://snap.stanford.edu/data/index.html). `http://snap.stanford.edu/data/index.html`.

[37] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[38] Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[39] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Betweenness centrality on gpus and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 76–85, New York, NY, USA, 2013. ACM.

[40] Kamesh Madduri, David A. Bader, Jonathan W. Berry, and Joseph R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances, 2007.

[41] Mauro Bisson, Massimo Bernaschi, and Enrico Mastrostefano. Parallel distributed breadth first search on the kepler architecture. *IEEE Transactions on Parallel & Distributed Systems*, 2015.

[42] Jiaoe Wang, Huihui Mo, Fahui Wang, and Fengjun Jin. Exploring the network structure and nodal centrality of chinas air transport network: A complex network approach. *Journal of Transport Geography*, 19(4):712–721, 2011.

[43] Alfredo Cuzzocrea, Alexis Papadimitriou, Dimitrios Katsaros, and Yannis Manolopoulos. Edge betweenness centrality: A novel algorithm for qos-based topology control over wireless sensor networks. *Journal of Network and Computer Applications*, 35(4):1210 – 1217, 2012. Intelligent Algorithms for Data-Centric Sensor Networks.

[44] Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186–198, 2009.

[45] Paolo Boldi and Sebastiano Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014.

[46] Glenn Lawyer. Understanding the influence of all nodes in a network. *Scientific reports*, 5, 2015.

[47] Kamesh Madduri, David Ediger, Karl Jiang, David A Bader, and Daniel Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[48] Adam McLaughlin and David A Bader. Scalable and high performance betweenness centrality on the gpu. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 572–583. IEEE Press, 2014.

[49] Miriam Baglioni, Filippo Geraci, Marco Pellegrini, and Ernesto Lastres. Fast exact computation of betweenness centrality in social networks. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, pages 450–456. IEEE Computer Society, 2012.

[50] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Umit V Catalyürek. Shattering and compressing networks for betweenness centrality. In *SIAM Data Mining Conference (SDM)*. SIAM, 2013.

[51] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C Hart. Edge vs. node parallelism for graph centrality metrics. *GPU Computing Gems: Jade Edition*, pages 15–28, 2011.

[52] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[53] Zhiao Shi and Bing Zhang. Fast network centrality analysis using gpus. *BMC bioinformatics*, 12(1):149, 2011.

[54] Ahmet Erdem Saryce, Erik Saule, Kamer Kaya, and mit V. atalyrek. Regularizing graph centrality computations. *Journal of Parallel and Distributed Computing*, 76(0):106 – 119, 2015. Special Issue on Architecture and Algorithms for Irregular Applications.

[55] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. *CoRR*, abs/1501.05387, 2015.

[56] A. Davidson, S. Baxter, M. Garland, and J.D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 349–359, May 2014.

[57] Oded Green and David A Bader. Faster betweenness centrality based on data structure experimentation. *Procedia Computer Science*, 18:399–408, 2013.

[58] Ulrik Brandes and Christian Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

[59] Mostafa Haghir Chehreghani. An efficient algorithm for approximate betweenness centrality computation. *The Computer Journal*, page bxu003, 2014.

[60] David A Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality. In *Algorithms and Models for the Web-Graph*, pages 124–137. Springer, 2007.

[61] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In *ALENEX*, pages 90–100. SIAM, 2008.

[62] Nick Edmonds, Torsten Hoefler, and Andrew Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[63] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. Multi-gpu graph analytics. *CoRR*, abs/1504.04804, 2015.

[64] Rami Puzis, Yuval Elovici, Polina Zilberman, Shlomi Dolev, and Ulrik Brandes. Topology manipulations for speeding betweenness centrality computation. *Journal of Complex Networks*, 3(1):84–112, 2015.

[65] David A Bader, Christine E Heitsch, and Kamesh Madduri. Large-scale network analysis. *Graph Algorithms in the Language of Linear Algebra*, 253:253, 2010.

[66] Davide Rossetti. Benchmarking GPUDirect RDMA on modern server platforms. `http://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms/`, October 2014.

[67] Jared Hoberock and Nathan Bell. Thrust CUDA library (http://thrust.github.com/). `http://code.google.com/p/thrust/`.

[68] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[69] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[70] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[71] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: A flexible data processing tool. *Communications of the ACM*, 2010.

[72] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51 (1):107–113, 2008.

[73] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 3rd edition edition, 2012.

[74] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: an in-depth study. In *Proceedings of the VLDB Endowment*, volume 3, pages 472–483, 2010.

[75] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[76] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[77] B. Thirumala Rao, N. V. Sridevi, V. Krishna Reddy, and L. S. S. Reddy. Performance issues of heterogeneous hadoop clusters in cloud computing. *Global Journal of Computer Science and Technology*, 11(8), 2011.

[78] Jeffrey Shafer, Scott Rixner, and Alan L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In IEEE, editor, *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 122–133, 2010.

[79] Chongyang Xue, Feng Liu, Honghui Li, Jun Xiao, and Zhen Liu. Research and design of performance monitoring tool for hadoop clusters. In Springer India, editor, *Proceedings of International Conference on Computer Science and Information Technology*, pages 809–816, 2014.

[80] Jimmy Lin, Dmitriy Ryaboy, and Kevin Weil. Full-text indexing for optimizing selection operations in large-scale data analytics. In ACM, editor, *Proceedings of the second international workshop on MapReduce and its applications*, pages 59–66, 2011.

[81] Xuelian Lin and Zide Meng. A practical performance model for hadoop mapreduce. In IEEE, editor, *Proceedings of the 2012 IEEE International Conference on Cluster Computing Workshops*, pages 231–239, 2012.

[82] Bo Dong, Qinghua Zhenga, Feng Tiana, Kuo-Ming Chaoc, Nick Godwinc, Tian Maa, and Haipeng Xua. Performance models and dynamic characteristics analysis for hdfs write and read operations: A systematic view. *Journal of Systems and Software*, pages 132–151, 2014.

[83] Gilad Mishne, Jeff Dalton, Zhenghua Li, Aneesh Sharma, and Jimmy Lin. Fast data in the era of big data: Twitter's real-time related query suggestion architecture. In *Proceedings of the 2013 international conference on Management of data*, pages 1147–1158. ACM, 2013.

[84] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, pages 59–72. ACM, 2007.

[85] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *NSDI*, volume 11, 2011.

[86] Grzegorz Malewicz, Matthew H. Austern, Aart J. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In ACM, editor, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[87] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. In *Proceedings of the VLDB Endowment*, volume 3(1-2), pages 285–296, 2010.

[88] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, pages 1–14, 2010.

[89] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. In *Proceedings of the VLDB Endowment*, volume 5(12), pages 1736–1747, 2012.

[90] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In IEEE, editor, *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 170–177, 2010.

[91] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets.

In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[92] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In USENIX Association, editor, *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[93] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In USENIX Association, editor, *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, 2012.

[94] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In ACM, editor, *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438, 2013.

[95] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 689–692, 2012.

[96] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In ACM, editor, *Proceedings of the 2013 international conference on Management of data*, pages 13–24, 2013.

[97] Lei Gu and Huan Li. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In IEEE, editor, *Proceedings*

of the 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), pages 721–727, 2013.

[98] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, 2011.

[99] Vassil Roussev and Golden G Richard III. Breaking the performance wall: The case for distributed digital forensics. In *Proceedings of the 2004 Digital Forensics Research Workshop*, volume 94, 2004.

[100] Lodovico Marziale, Golden G Richard III, and Vassil Roussev. Massive threading: Using gpus to increase the performance of digital forensics tools. *digital investigation*, 4:73–81, 2007.

[101] Vassil Roussev, Liqiang Wang, Golden Richard, and Lodovico Marziale. A cloud computing platform for large-scale forensic computing. In *Advances in Digital Forensics V*, pages 201–214. Springer, 2009.

[102] Simson L Garfinkel. Automating disk forensic processing with sleuthkit, xml and python. In *Systematic Approaches to Digital Forensic Engineering, 2009. SADFE'09. Fourth International IEEE Workshop on*, pages 73–84. IEEE, 2009.

[103] Josiah Dykstra and Alan T Sherman. Acquiring forensic evidence from infrastructure-as-a-service cloud computing: Exploring and evaluating tools, trust, and techniques. *Digital Investigation*, 9:S90–S98, 2012.

[104] ChaeHo Cho, SungHo Chin, and Kwang Sik Chung. Cyber forensic for hadoop based cloud system. *International Journal of Security & Its Applications*, 6(3), 2012.

[105] Simson L Garfinkel. Digital media triage with bulk data analysis and bulk_extractor. *Computers & Security*, 32:56–72, 2013.

[106] Adrian Shaw and Alan Browne. A practical and robust approach to coping with large volumes of data submitted for digital forensic examination. *Digital Investigation*, 10(2):116–128, 2013.

[107] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. *Digital Investigation*, 6:S2–S11, 2009.

[108] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications Co., Greenwich, CT, USA, 2010.

[109] M. Bostock, V. Ogievetsky, and J. Heer. D^3 Data-Driven Documents. *IEEE TVCG*, 17(12):2301–2309, Dec 2011.