SAPIENZA
UNIVERSITÀ DI ROMA

Large Graphs on multi-GPUs

Enrico Mastrostefano

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**

January 9, 2013

Introduction

A network can be defined as a set of interconnected nodes. This simple model describes many different phenomena and it is used in a wide range of disciplines from physics and biology to social sciences.
Over the years the size of the investigated networks has grown, leading to the definition of the so called large-scale networks which include hundred million or even billions of nodes. A well known and widely studied example is the World Wide Web, whose size is steadily growing: currently the number of web pages is tens of billions. Protein interaction networks, the human brain, the metabolic interaction networks, transportation networks and social networks are some other examples [48, 49, 61].

Different experimental studies, carried out in the last two decades, revealed that large-scale networks tend to form complex structures. Nodes tend to aggregate in dense clusters. Inside each cluster, several nodes, called hubs, have a large number of links whereas the most part of the other nodes have few [17, 34, 65, 74, 76].

From the theoretical point of view, large-scale networks can be described by using probabilistic graph models. They are generative models that permit to create a graph on the basis of some probabilistic rules. Through the use of these representations it is possible to study the time evolution of the network,

i.e., how nodes and links are added and removed from the network.

On the other hand, structural properties of networks are evaluated by means of common graph algorithms, like the minimum spanning tree or the Breadth First Search (BFS). Graph traversal algorithms like BFS are of fundamental importance in many practical applications (a traversal is a systematic method of exploring all vertices and edges in a graph). BFS serves as a building block for many other algorithms and is employed to compute important metrics used to characterize the network. For instance, BFS serves to identify community structures, that is, how vertices are connected each other, or to compute the *centrality* of a vertex, that is a measure of the importance of the vertex in the graph.

Due to the huge size, the traversal of large graphs is quite demanding in terms of both computational and memory resources, namely, it must be performed by using parallel computing architectures. Unfortunately, most graph algorithms are memory intensive and have irregular memory access patterns that strongly depend on the structure of the graph. These features make them ill-suited to modern high performance platforms. Nevertheless, several authors in the last few years have successfully implemented high performance graph traversals on both parallel and distributed architectures. They demonstrated that, by following appropriate strategies, graph algorithms can be accelerated by using modern supercomputers [13, 42, 30, 79].

There is a wide variety of parallel supercomputers but they 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 have thousands of computing nodes. In principle, with the

help of a distributed architecture there is no limit to the size of the network that can be studied.

We focus our work on the development of a distributed algorithm to perform a BFS visit on a large graph. For the implementation of the computational part we resort to clusters of Graphic Processing Units (GPUs).

Generally, a distributed system is a cluster of computing nodes interconnected via a wired network. Communications are typically implemented by using the Message Passing Interface (MPI) primitives. Each node is a system in itself that can be equipped with a single- or a multi-core CPU. For the GPU clusters, each node hosts also one or more GPU devices. To carry out computations, the nodes of the cluster must exchange data each other.

Graph algorithms are notoriously difficult to parallelize. They have low arithmetic intensity: the time spent in computation is less then the time spent performing memory access operations. In a distributed environment, data can be in a remote memory, thus, most of the execution time, is spent sending and receiving data over the communication network. Moreover, communication patterns are irregular, the number and the size of messages exchanged may vary during the execution of the algorithm. As a consequence the performance bottleneck is represented by the communication among nodes [79, 51, 23].

To obtain a significant improvement on a single GPU and to scale by using multiple GPUs, we developed a novel algorithm. We propose a technique for mapping threads to data that achieves a perfect load balance by leveraging prefix-sum and a binary search operations. To reduce the communication overhead, we perform a pruning operation on the set of edges that needs to be exchanged at each BFS level. The result is an algorithm that exploits at its best the parallelism available on a single GPU and minimizes communication

among GPUs. As far as we know this is the first attempt to implement a graph algorithm on multi-GPUs clusters.

Our code was submitted to the Graph 500 benchmark, a new graph-theoretical challenge. To complement the compute-intensive Top 500, Graph 500 evaluates performances of modern supercomputers on data-intensive applications. By using 128 GPUs we entered the Graph 500 list at position number 20 (November 2011 ranking).

Communication among GPUs that are located on different nodes of the cluster, involves the hosting CPU. Data must be transferred to the CPU before and after any MPI call. This requirement imposes an additional overhead to the data transfer. To overcome the issue several solutions have been proposed both at hardware and software level. One of these solutions is the APEnet+ interconnection technology. It allows for the transfer of data directly from GPU to the communication link by means of the *GPUdirect* feature, introduced in the latest NVIDIA cards.

We adapted our original algorithm to use the *GPUdirect* technology. Our results, albeit preliminary, show a clear advantage with respect to classical interconnection technology (Infiniband). This result is of great interest considering that, these technologies, are an essential part of research efforts, towards the definition of a general mechanism for direct communication among GPUs.

The present dissertation is organized as follows. In the first Chapter we describe the problem of analyzing large graphs. We summarize the basic properties of large-scale networks and introduce some probabilistic models. Then we describe the serial BFS algorithm and the level synchronous BFS, which is of considerable importance in the parallel implementation of the

BFS. In the second Chapter we illustrate the GPU features and the CUDA programming model. The third Chapter deals with parallel BFS on shared memory systems; multi-core CPU and (single) GPU fall within this category. We describe two different algorithms to perform parallel BFS and give a short review of some recent works on both multi-core CPU and GPU. In the fourth Chapter we present our original study: the development of a distributed BFS. We report the issues related to the problem and review related works. We present our solution for a multi-GPUs cluster interconnected via the standard Infiniband technology and finally we extend our work to support the APEnet+ technology.

Contents

Chapter 1

Large Graphs

A number of natural and artificial phenomena may be described by using networks, i.e. sets of interconnected nodes. Over the years, the size of the studied networks has grown. Nowadays, networks with millions of vertices and hundred million or even billions of edges are studied after being extracted starting from huge datasets. A well know and widely studied example is the Internet. Internet represents also the infrastructure of the World Wide Web, a network made of hyperlinks, and of many Social Networks, that represent relationships among individuals (or web sites where people waste their time). The size of the Internet is growing fast: currently the number of web pages may be 30 billion or more [1], and the number of connected—devices is probably more than a billion.

Social networks are very attractive for many researchers in the area of sociology, history, epidemiology and economics. Well established social networks like Facebook, MSN Messenger or Twitter have hundred million links. Protein interaction networks, the human brain and the metabolic interaction

[1] http://www.worldwidewebsize.com/

networks are some examples from biology. The human brain network is one of the most complex with its $\sim 10^{11}$ nodes[48, 49, 61].

Networks, like those mentioned above, are often described as large graphs having millions of vertices and billions of edges. The theoretical formalization of such large graphs falls within the theory of random graphs. Several probabilistic/stochastic models have been proposed to mimic their structure and evolution. New metrics have been introduced to better characterize their properties. Those metrics are computed on the basis of common graph algorithms, like single source shortest path, minimum spanning tree and Breadth First Search (BFS).

Even the execution of a simple algorithm, like BFS, on a graph with billions of edges, requires the use of a parallel computing architecture. While serial algorithms on graphs have been widely studied and can be efficiently implemented, the corresponding parallel versions are still lacking behind. Parallel graph algorithms are challenging for many reasons (as we will discuss in the following). For instance, most algorithms are memory intensive and have irregular memory access patterns that strongly depend on the structure of the graph. These features make them ill-suited to modern high performance architectures. Nevertheless, recent studies [13, 42, 30] have demonstrated that, with appropriate strategies, graph algorithms can be accelerated by using modern parallel supercomputers.

This chapter is devoted to an introduction to large graphs and the breadth first search algorithm, whereas, in the next chapter, we will introduce the main features of modern GPUs.

1.1 Random Graphs

Random graphs are powerful mathematical models to study large networks. They are generative models that permit to produce a graph on the basis of some probabilistic rules. Most often the models do not impose a certain property to the network but rather give general principles or mechanisms of edge creation that lead to the rise of a global statistical property or distribution in the network [46].

Many models aim at describing the structure and also the evolution of the network over time, i.e. how nodes and edges are added and removed from the network. In the present work, we are not interested in the evolution of the graph but only in its structure.

In the next two sections we report few basic definitions and notions on (random) graphs.

1.1.1 Basic graph notions

$G(V, E)$ is a graph G whose vertices v belong to the set V and whose edges $e = (u, v); u, v \in V$ belong to the set E. $N = |V|$ is the number of vertices in G whereas, $M = |E|$ is the number of edges. We will denote the number of vertices and edges with $|V|$ and $|E|$ or N and M.

G can be:

- **directed**: if the pair (u, v) is oriented.
- **weighted**: if each edge e has associated a scalar value p_e.
- **multi-graph**: there can be multiple edges between two vertices.
- **simple**: unweighted, undirected graph containing neither graph loops nor multiple edges.

Unless explicitly stated, we consider only simple graphs.

A path between two vertices $u, v \in V$ is a sequence of edges from u to v. A closed path in which some vertices are repeated is called **cycle**. G is said to be **connected** if there exist a path between any two vertices in V. A **tree** is a connected graph with no cycle. 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**. Given any two nodes u_1, u_2 in G, the **distance**, is measured as the shortest path that connects them. The **diameter** is the maximum distance in G.

The **degree** of a node is defined as the number of edges incident to the vertex. The total degree of G then is $\sum_{v \in V} deg(v) = 2|E|$. The degree of a vertex is an important metric, many structural properties of graphs can be deduced from the knowledge of the degree of all the nodes.

1.1.2 Erdös-Rény random graphs

The earliest probabilistic generative model for graphs was the random graph model introduced by Erdös and Rény in 1960 [32] and Gilbert in 1959 [35].

Consider a set of vertices $V = 1, ..., n$ and let an edge between any two nodes, i and j, be formed with probability p, where $0 < p < 1$. The formation of edges is independent. The corresponding graph is $G(n, p)$. This is a binomial model of link formation, which gives rise to a manageable set of calculations regarding the resulting structure. The average number of edges on the graph as a whole is $\frac{1}{2}n(n-1)p$, whereas the average degree of a vertex is:

$$z = \frac{n(n-1)p}{n} = (n-1)p \simeq np,$$

where the approximation holds for large values of n.

One interesting feature, originally demonstrated by Erdös and Rény, is that, when the value of z varies, from 0 to a positive integer, the model shows a phase transition, exactly in $z = 1$. For $z < 1$, there are few edges in the graph and most vertices are disconnected from each other. However, when z is approaching the value of 1, there is one largest component that contains a finite fraction F of the total number of vertices. The size of F scales linearly with the size of G and it is called giant component. The size of the other components in the graph remains constant as the graph size increases. It can be proved that, for a large value of n, the size of the remaining components is of order $log(n)$.

We can calculate some statistics that describe the graph. For instance, we can find the degree distribution fairly easily. The degree distribution of a random graph describes the probability that a given node has degree d. The probability that a given node i has exactly k edges is:

$$P(d = k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

For large values of n and small values of p the binomial expression can be approximated by the Poisson distribution. Both binomial and Poisson distributions are strongly peaked about the mean d, and have a large tail that decays rapidly as $1/k$.

The degree distribution provides a number of information about the structure of the graph. For instance, a Poisson distribution implies that most vertices have a degree close to the average value (see figure 1.1).

The main drawback of the random graph model is that it produces graphs that fail to match real-world networks.

1.2 Real world graph

Many studies show that the features of real world (large) networks cannot be described with the classical random graph model.

The most important difference is that, real world networks exhibit a power law distribution of the degree: $P(d = k) = Ck^{-\gamma},\ k \to \infty$.

The power law distribution has been found in various real datasets like, for instance:

- World Wide Web: considering pages as vertices and links as edges [17].
- Internet: at the level of so-called "autonomous systems" [2] [34].
- Citation network: in which the nodes are papers and citations are links [65].
- Protein-protein interaction networks [74].

In figure 1.1 are plotted, for comparison, the Poisson distribution and a power law with $\gamma = 3$.

In order to mimic the power law distribution, Barabasi and Albert [17] proposed the preferential attachment model. Instead of adding edges with uniform probability, edges are connected to vertices with a probability proportional to their popularity. Suppose to add a vertex to the graph, then edges are added one at time. The probability that the new vertex will be attached to vertex i is given by: $\pi_i = k_i / \sum_j k_j$.

Erdös and Rény graphs have small diameter, but have few triangles. Real-world graphs, like those from social networks, contain many (if A and B are friends and A and C are friends, then it is fairly likely that B and C

[2] An autonomous system is a group of computers within which data flow is handled autonomously, while data flow among groups is conveyed over the public Internet

are also friends). To construct a network with small diameter and positive density of triangles, Watts and Strogatz [76] started from a ring lattice with n vertices and k edges per vertex, then rewired each edge with a probability p, connecting one end to a vertex chosen at random. The resulting graph is connected by definition and has short diameter, it is in fact called small-world graph.

The power law distribution is often called scale-free distribution and, networks that obey that distribution, are called scale-free networks.

The skewed distribution associated with real-world graphs implies that some special vertices, called hubs, have an huge number of neighbors whereas most of them have few (see fig 1.1). As we shall discuss below, this property has negative side effects on the parallel implementation of graph algorithms since it can lead to a severe imbalance of workload among the tasks (see section 1.3.2). On the other end, the small diameter property, has positive consequences. For instance, the number of iterations of a parallel BFS is proportional to the diameter of the graph (see section 1.3.2).

It is worth noting that most of real-world networks can be represented by means of a sparse graph that is, the number of edges is much smaller than the maximum number of possible edges: $|E| << |V|^2$.

1.2.1 Real-world graph generators

In parallel with the study of the structural properties of real-world graphs, there has been an effort to find practical mechanisms to generate graphs that have the desired properties. As stated in [47] to have realistic generators is important for at least two reasons. First they allow to simulate a graph for testing theoretical hypothesis and can be used to simulate different scenarios. Second they give some insights on the networks properties thus helping the

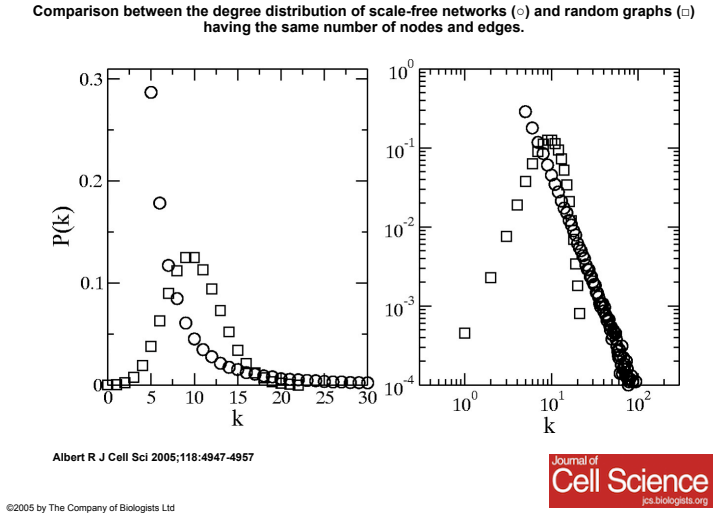


Figure 1.1. Comparison between the degree distribution of scale-free networks (○) and random graphs (□) having the same number of nodes and edges. For clarity the same two distributions are plotted both on a linear (left) and logarithmic (right) scale. The bell-shaped degree distribution of random graphs peaks at the average degree and decreases fast for both smaller and larger degrees, indicating that these graphs are statistically homogeneous. By contrast, the degree distribution of the scale-free network follows the power law $P(k) = Ak^{-3}$, which appears as a straight line on a logarithmic plot. The continuously decreasing degree distribution indicates that low-degree nodes have the highest frequencies. However, there is a broad degree range with non-zero abundance of very highly connected nodes (hubs) as well. Note that the nodes in a scale-free network do not fall into two separable classes corresponding to low-degree nodes and hubs, but every degree between these two limits appears with a frequency given by $P(k)$.

development of theoretical models. One of the most successful generator is the R-MAT generator [25]. It is based on a Recursive Matrix approach, in which the adjacency matrix that represents the graph is recursively subdivided and then populated, following certain probabilistic rules. The whole matrix is divided in four partitions, each partition has associated a probability: a, b, c, d. Then, each partition is again divided in four and this

procedure is repeated until the simple cell is reached.

This model has gained much popularity, for instance, the novel Graph 500 benchmark chose this generator to create the input graph that has to be analyzed.

By changing the value of the four parameters, a, b, c, d the properties of the produced graph vary accordingly. For $a = b = c = d = 0.25$ the model reproduces the standard, Erdös-Rényi, random graph. On the other end, the values a = 0.57, b = 0.19, c = 0.19, and d = 0.05, provides a steep degree distribution power-law graph. This produces a maximum degree of approximately $200.000 = 2^{17.64}$, with 2^{25} vertices and 2^{28} edges [58].

Our work, and most of the studies that we will review, rely on an R-MAT generator to produce large graphs with the desired properties.

1.3 Analysis of large graphs

Specific properties of large graphs are described by means of a set of new metrics. For instance, the *betweenness centrality* measures the centrality of a node in a network. It is equal to the number of shortest paths from all vertices to all others that pass through that node.

As already stated, the size of the most interesting graphs requires the use of parallel computing systems. However, graph algorithms are a typical example of applications for which it is not simple to have a sizeable advantage by using parallel computing architectures. To gain a better understanding of the issues due to implementation of algorithms with irregular memory access patterns, like those in use for studying graphs, many recent studies focus on (apparently) simple problems. Several communities have proposed computational challenges having as subject graphs. For instance, the 9 DI-

MACS challenge [1] aims at finding shortest path in graphs, the Graph 500 [2] benchmark uses BFS as its core, SSCA#2 benchmark is composed of four kernels operating on large scale directed multi-graphs [5]. Hereafter we focus on BFS.

Breadth first search is a simple graph algorithm that is widely used as a building block for more complex algorithms. For instance it can be used to find the connected components in a graph or to compute the shortest path between two vertices. It is representative of a class of algorithms for which is hard to obtain a significant speed up from parallelization.

In the following sections, we discuss the serial BFS and the level synchronous BFS algorithm. The latter is of considerable importance in the implementation of parallel algorithms.

1.3.1 Graph representation and serial BFS

There are two common ways to represent a graph $G = (V, E)$: with either a set of adjacency lists or an adjacency matrix. The adjacency list representation is preferable when the graph is **sparse** i.e. $|E| << |V|^2$ because is more compact. The adjacency matrix should be preferred when the graph is **dense** i.e. $|E| \sim |V|^2$. The adjacency list representation of a graph is implemented via a set of $|V|$ lists, one for each vertex of the graph. For each $u \in V$ the adjacency list of u contains all the vertices v such that there is an edge (u, v) in E (see figure 1.2).

Given the graph $G = (V, E)$ and a source (or root) vertex s, the breadth first search explores each edge of G to discover all vertices that are reachable from s. BFS is widely studied because it is part of more complex algorithms. Here we summarize only those aspects that will be relevant for the rest of the present work, for further information see [27].

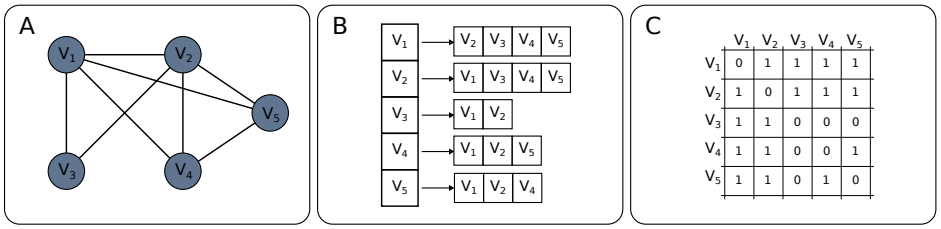


Figure 1.2. In panel A is represented an undirected graph G(V,E)=G(5,16). In panel B is provided the adjacency list representation of G. Panel C is the adjacency matrix of G.

Algorithm (1) presents an implementation of the BFS, where the graph G is undirected and represented as an adjacency list (fig 1.3 A, B).

Algorithm 1 Serial BFS

```
1: d[u] ← −1, ∀ u ∈ V
2: p[u] ← −1, ∀ u ∈ V
3: d[s] = 0
4: p[s] = s
5: enqueue(Q, s)
6: while Q ≠ ∅ do
7:    u ← dequeue(Q)
8:    for each v ∈ Adj[u] do
9:       if p[u] == −1 then
10:         p[v] = u
11:         d[v] = d[u] + 1
12:         enqueue(Q, v)
13:      end if
14:   end for
15: end while
```

The algorithm starts by visiting the source vertex s; its distance $d[s]$ and its predecessor $p[s]$ (line 3-4) are set, then the vertex is enqueued in Q. At each iteration, the first vertex of the queue is dequeued, all its neighbors are inspected and, if the value of their distance and/or predecessor are not set (line 9), that is they have never been seen, they are added to the queue. The

algorithm ends when all reachable vertices have been visited and the queue is empty.

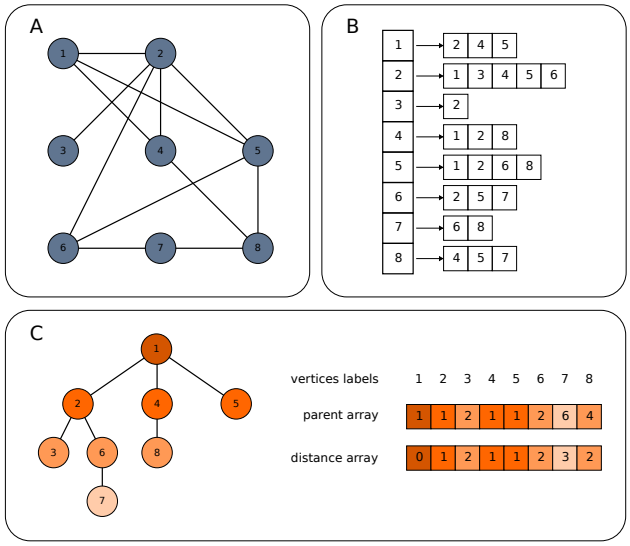


Figure 1.3. In panel A is represented an undirected graph. In panel B is provided the adjacency list representation of the graph. Panel C shows the result of one BFS visit started from vertex 1. On the left is depicted the BFS-tree rooted at 1. On the right, the parent and the distance arrays. The colors highlight different levels of the BFS.

The BFS procedure "expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier" [3], so that, all vertices at distance k are discovered before any vertices at distance $k+1$. At the end of the procedure, the distance array contains the shortest path from s to any reachable vertex in the graph [27]. The array of predecessors or parent contains a BFS-tree rooted at s: each index of the array is a vertex in the graph and the corresponding value is the predecessor in the BFS-tree. It is worth noting that BFS explores only the connected component that contains the source vertex s.

[3] From [27]

Table 1.1. The table is referred to the graph in figure (1.3). The BFS is started at vertex 1 and new vertices visited are added to next level, according to algorithm (1).

BFS LEVEL	CQ	NLFS	NQ
0	1	2,4,5	2,4,5
1	2,4,5	1,3,4,5,6,1,2,8,1,2,6,8	3,6,8
2	3,6,8	2,2,5,7,4,5,7	7
3	7	6,8	

Figure (1.3 C) depicts the result of the algorithm (1) applied to the graph shown in fig. (1.3). Table 1.1 shows the levels of the BFS performed on the graph represented in figure (1.3).

The time complexity of the BFS is given by the time for the queue operation that is $O(V)$, plus the time for scanning each adjacency list, that is $O(E)$. The initialization takes $O(V)$ and thus the total time, $O(V + E)$, is linear in the size of the adjacency list.

1.3.2 Level synchronous BFS

Breadth first search, as many graph algorithms, is memory-bound and memory access patterns are fine-grained and irregular. These features cause poor performances on shared memory systems, that are cache-based [13, 7]. On a distributed memory system the running time of the algorithm is dominated by the communication part [79, 23, 73] and is difficult to outperform the sequential version. Most of the parallel implementations, on both type of architectures, are based on the level synchronous BFS algorithm.

In the level synchronous BFS, the current queue, CQ, is seen as a current level set of vertices. For each vertex in the current level, all its neighbors must be visited. In parallel systems, this operation, is always carried out in parallel. The set of all neighbors composes the *Next Level Frontier Set* (NLFS). From the NLFS only new vertices are selected to build the queue for the next level (figure 1.4).

The BFS visit is divided into levels with a distance from the root that increases at each subsequent level. For a graph with diameter D, the number of levels visited by the algorithm will be at least D/2 and at most D, depending on where the search is initiated.

This method is referred in literature as **Level Synchronous BFS** because, to ensure the correctness of the computation in a parallel implementation, a synchronization is required at the end of each level.

As shown in figure 1.4, the total number of elements in the CQ is $|CQ| = k$. The total number of elements in the NLFS is $\sum_{i=0}^{k} d_{u_k}$, the sum of all degrees of vertices in the queue.

Real-world graphs have skewed degree distributions (see section 1.2). While many of the vertices have a small number of neighbors, hubs can have thousands. The degrees of the vertices may differ from each other, by

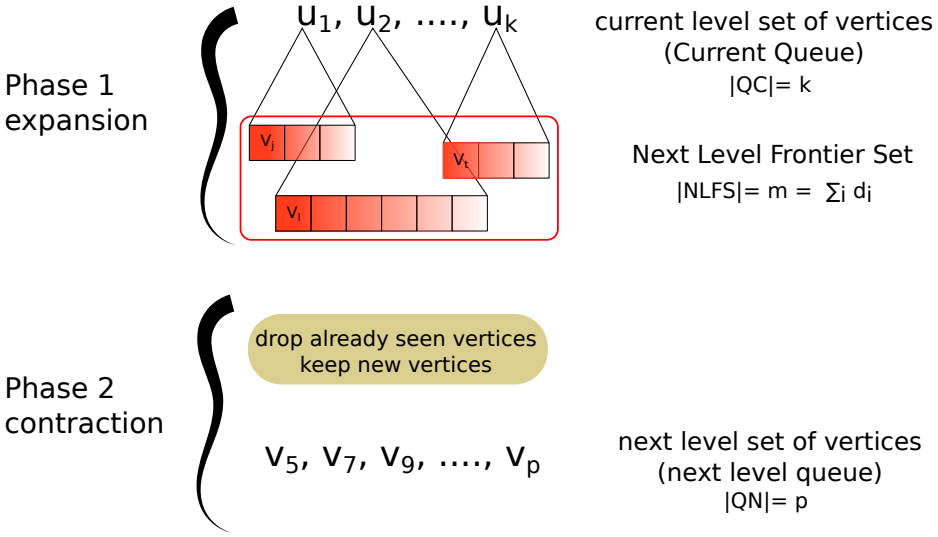


Figure 1.4. BFS phases: In the first phase, all neighbors of each vertex in the current queue are inserted in the NLFS. In the second phase vertices already seen in the NLFS are removed. The remaining vertices are inserted in the next level queue. The total number of elements in the NLFS is equal to the sum of the degrees of all elements in the queue.

several orders of magnitude. Consequently the NLFS can be greater than the queue by several order of magnitude. Moreover, it will expand and contract very quickly. Thus, the number of BFS levels is always small compared to the number of elements in the graph.

If there are k tasks, one for each element of the queue, during the expansion phase, each task has to visit the neighbors of its element. The workload among tasks is clearly unbalanced.

It is important to realize that, the size of the NLFS, is much greater than the size of the other structures used in the algorithm. Thus, for most implementations, the number of memory accesses during the BFS is in the order of the NLFS size.

Now suppose that the graph is stored as an adjacency lists structure,

and is loaded in the main memory of the system. At the beginning of each iteration of the algorithm, each task reads a vertex from the CQ. Then it starts to visit the neighbors of its vertex. Task 1 starts to visit the adjacency list of vertex u_1, Task 2 visits the adjacency list of u_2 and so on. The adjacency lists are not always in contiguous memory locations. The more distant they are, the more expensive it will be to visit them, in terms of memory accesses.

The vertices in the queue, change at each iteration, as well as the vertices in the NLFS. Thus, memory accesses, are irregular and not predictable. It should also be clear, by now, that they strongly depend on the structure of the graph. For real-world graphs, the situation is exacerbated by the enormous difference in the size of adjacency lists.

In Chapter 3 we will describe the issues in a parallel implementation on a shared memory system. Distributed systems have different issues, related to the communication among tasks, that will be described in Chapter 4.

Chapter 2

GPUs and CUDA overview

In the last 5 years, driven by a steadily growing request for real-time, high-definition 3D graphics, Graphic Processor Units or GPUs have evolved becoming highly parallel, multi-threaded, processors with huge computing power and very high memory bandwidth, as illustrated by Figure 2.1.

The reason behind the discrepancy in floating-point processing capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computations - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 2.2.

In general, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high

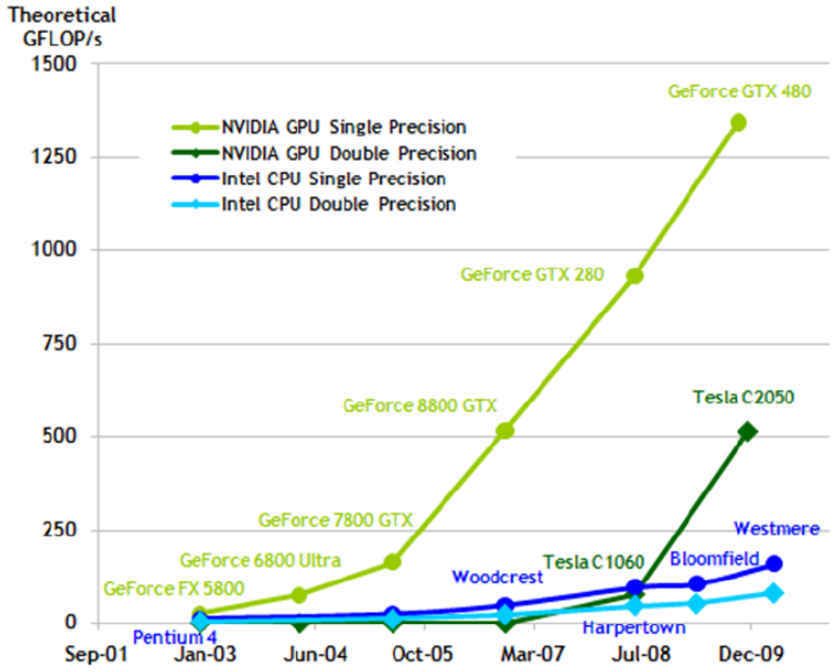


Figure 2.1. Floating-Point operations per second for a CPU and a GPU.

arithmetic intensity, the memory access latency can be hidden with calculations instead of resorting to large data caches. Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. As a matter of fact, many algorithms, beside image rendering, may be accelerated by data-parallel processing, from general signal processing or physics simulations to computational finance or computational biology.

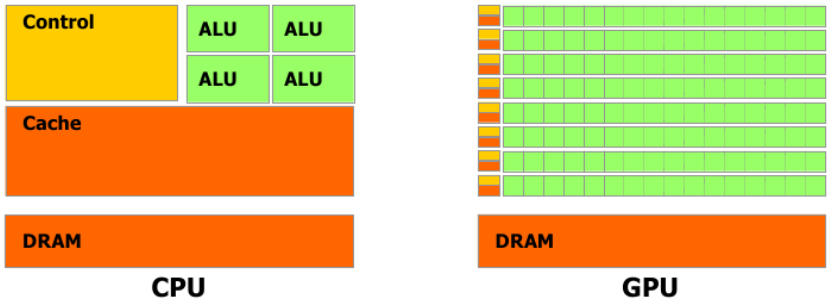


Figure 2.2. In a GPU more transistor are devoted to data processing.

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing architecture (with a new parallel programming model and instruction set architecture) that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language.

2.1 The CUDA programming model

The advent of multi-core CPUs and many-core GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to many-core GPUs with widely varying numbers of cores. The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core are three key abstractions, a hierarchy of thread groups, shared memories, and barrier syn-

chronization, that are simply exposed to the programmer as a minimal set of language extensions. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of processor cores as illustrated by Figure 2.3, and only the runtime system needs to know the physical processor count.

This scalable programming model allows the CUDA architecture to span a wide market range by simply scaling the number of processors and memory partitions.

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to single execution like regular C functions. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through a built-in *threadIdx* variable.

For convenience, *threadIdx* is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. The index of a

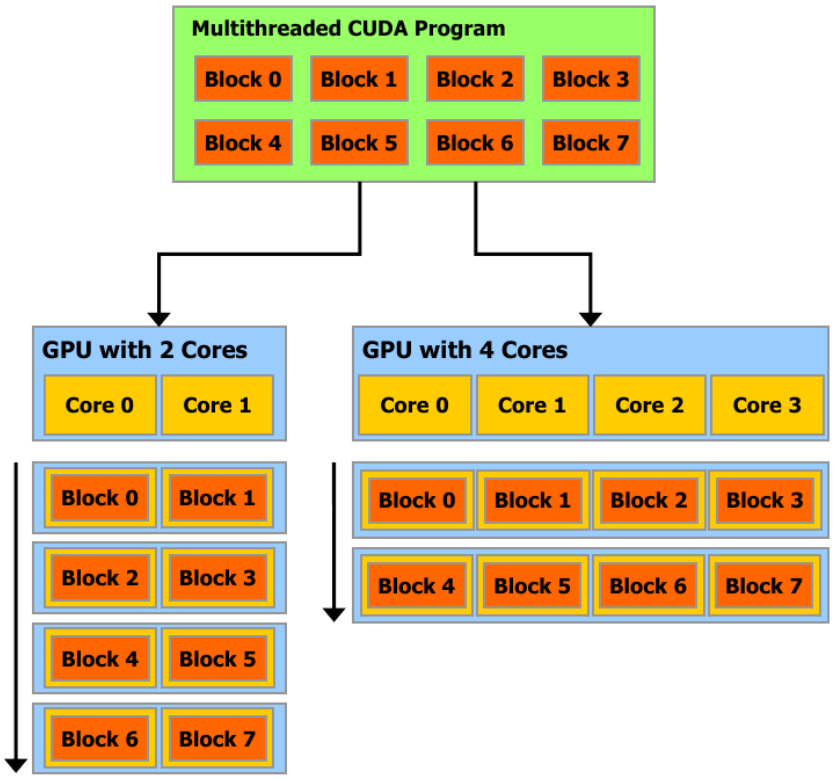


Figure 2.3. A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

thread and its thread ID relate to each other in a straightforward way: for a one-dimensional block, they are the same; for a two-dimensional block of size (Dx, Dy), the thread ID of a thread of index (x, y) is $(x + yDx)$; for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is $(x + yDx + zDxDy)$.

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same multiprocessor and must share the limited memory resources available on that core. On current GPUs, a thread

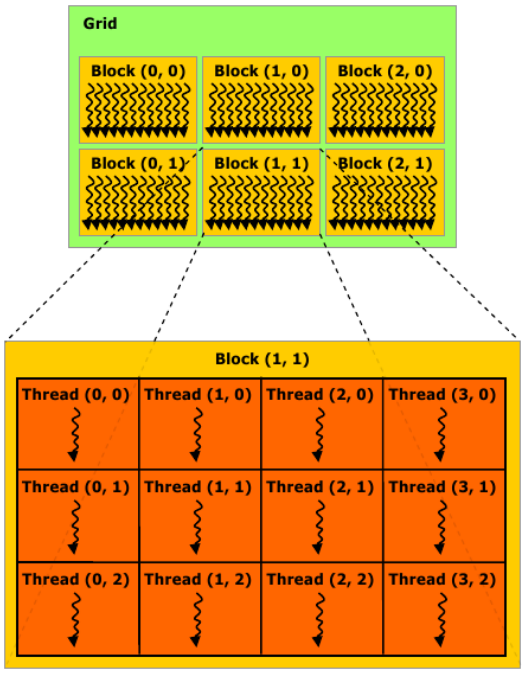


Figure 2.4. Grid of thread blocks.

block may contain up to 1536 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Blocks are organized into a one-dimensional, two-dimensional or tri-dimensional grid of thread blocks as illustrated by Figure 2.4. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

The number of threads per block and the number of blocks per grid are specified in the kernel call. Each block within the grid can be identified by a multi-dimensional index accessible within the kernel through the built-in *blockIdx* variable. The dimension of the thread block is accessible within the kernel through the built-in *blockDim* variable.

Thread blocks are required to execute independently: it must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores as illustrated by Figure 2.3, enabling programmers to write code that scales with the number of cores. Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling a specific intrinsic function that acts as a barrier at which all threads in the block must wait before any is allowed to proceed. For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache).

2.2 SIMT Architecture

GPU is a multiprocessor designed to execute hundreds of threads concurrently. To manage such a large number of threads, it employs a unique architecture called SIMT (Single-Instruction, Multiple-Thread). The multiprocessor manages threads in groups of 32, called warps. Individual threads composing a warp start together but have their own instruction address counter and register state and are therefore free to branch and execute independently. When one or more thread blocks are assigned to the multiprocessor, it divides them into warps and each warp is then scheduled for execution. A warp executes one common instruction at a time, so, to obtain the highest concurrency and the best performances, all 32 threads should have the same execution path. If threads diverge to follow different conditional branches, the warp serially executes each branch. However branch divergence occurs

only at the warp level. Different warps always execute instruction independently.

2.3 Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 2.5. Each thread has private local memory. Each thread block has shared memory visible to all threads in the block and with the same lifetime as the block. All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages (see [63]). Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats (see [63]). The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

As illustrated by Figure 2.6, the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime (see [63]). This includes device memory allocation and deallocation as well as data transfer between host and device memory.

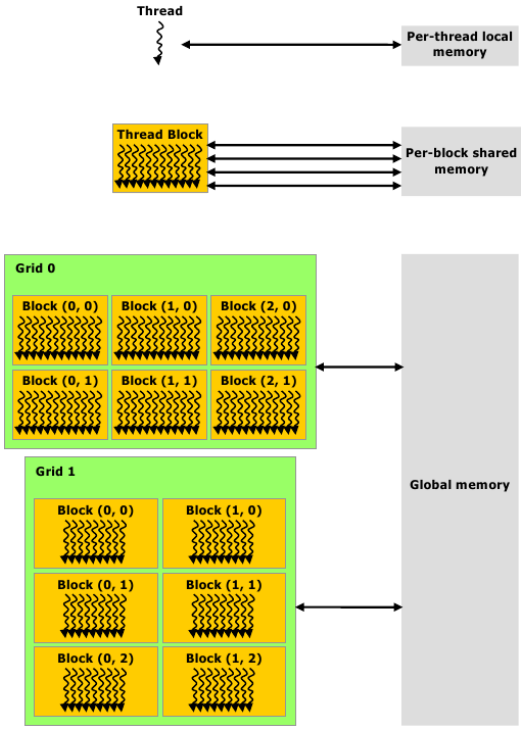


Figure 2.5. GPU memory hierarchy.

2.4 CUDA streams

CUDA supports concurrency within an application through *streams*. A stream is a sequence of commands that are executed in order. Different streams, on the other hand, may execute their commands out of order with respect to each other or concurrently. The amount of execution overlap between two streams depends on the order in which the commands are issued to each stream and whether or not the GPU supports overlap of data transfer and kernel execution. Further information about *streams* can be found in the CUDA documentation [63].

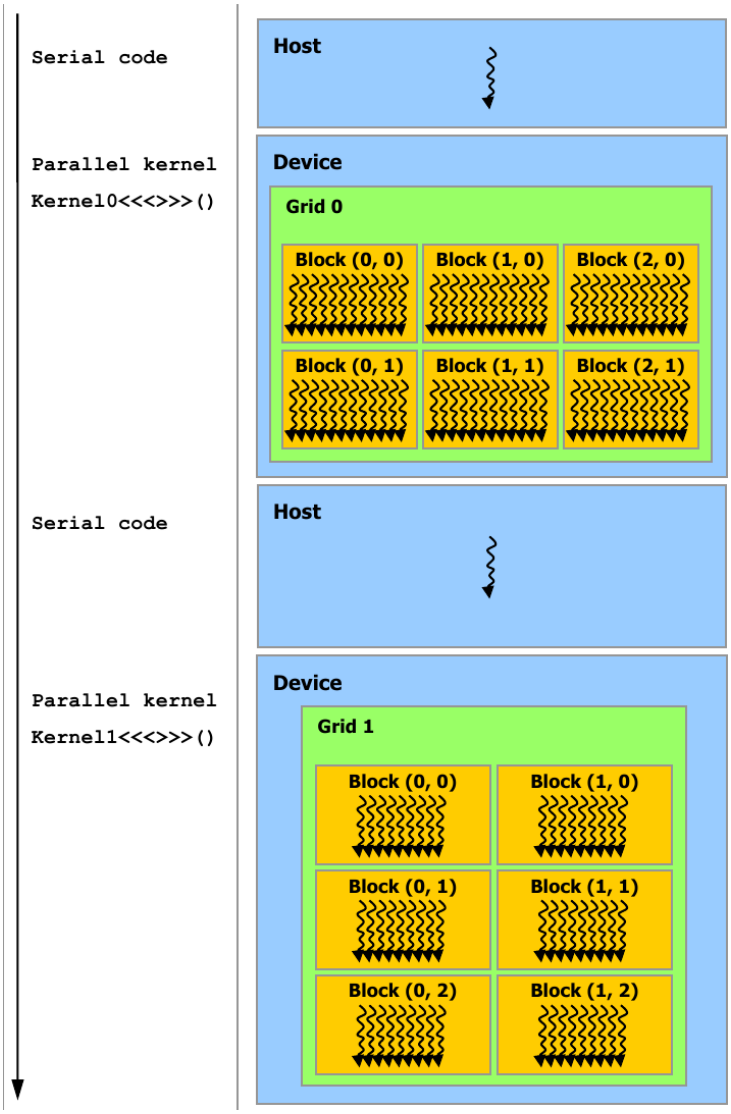


Figure 2.6. Heterogeneous programming: serial code executes on the host whereas parallel code executes on the device.

2.5 Clusters of GPUs

When the size of a problem is too large for the memory of a single GPU (that currently is limited to a few GBytes) or there is the need to reduce the time

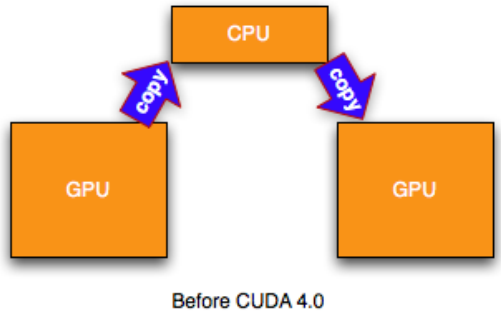


Figure 2.7. Data exchange between GPUs before CUDA 4.0

to solution, it is possible to resort to *clusters* of GPUs. In general, GPUs can not exchange data directly. In CUDA, up to version 4.0, the CPU had to be always involved in the communication as shown in Figure 2.7. For every data exchange between two GPUs, it was necessary to:

- upload data to the CPU (a *device to host* memory copy operation)
- if the GPUs were controlled by different CPUs, then exchange data among CPUs. The Message Passing Interface (MPI) could be used for that purpose to guarantee portability and scalability when GPUs are plugged into systems interconnected by networks (*e.g.,* Infiniband).
- download data to the GPU (*host to device* memory copy)

thus realizing an MPI+CUDA hybrid programming scheme. The need to explicitly copy data between device and host memories, before and after any MPI transfer call represents an issue from both efficiency and simplicity of programming viewpoint.

Actually, CUDA 4.0 introduced the possibility to carry out memory copy operations directly between two different GPUs. If such mechanism, named

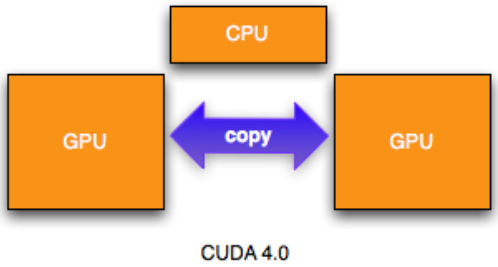


Figure 2.8. Direct memory copy between GPUs enabled by CUDA 4.0

in CUDA as *peer-to-peer*, is enabled, then the copy operation no longer needs to be staged through the CPU (see Figure 2.8) and is therefore faster.

However, only recent NVIDIA GPUs support the *peer-to-peer* mechanism. Moreover the source and the target GPU must be connected to the same PCI-e root complex so the general issue remains unsolved.

Recently, two widely used MPI implementations, OpenMPI [4] and MVAPICH2 [3], started to offer the possibility to specify GPU memory pointers in MPI functions, relieving the programmer from the management of data transfers between GPU and CPU memories.

This feature represents an essential part of research efforts, taking place at both hardware and software levels, aimed towards the definition of a general mechanism for direct communication among GPUs, (at least without explicit involvement of CPUs). On the hardware-side, research focuses on the development of interconnection technologies able to transfer data from GPU memory straight to the communication link. One solution supporting this approach is the APEnet infrastructure (see section 4.4.1).

Chapter 3

Parallel BFS on shared memory systems

3.1 Overview of shared memory systems

There is a great variety of shared memory parallel systems, very different from each other. Generally, they have in common the ability for all processors to access all memory, as a global address space. Changes in a memory location due to a processor are visible to all other processors. However, each processor has its own cache memory. A cache memory is a smaller and faster memory, which stores copies of the data from the most recently used main memory locations. Whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors (cache coherence).

From the programming point of view, the global address space provides a user-friendly perspective to memory. Data sharing among tasks is both fast and *almost* uniform, due to the proximity of memory to CPUs. The

main drawback is that synchronization, required to ensure correct access to global memory, is a responsibility of the programmer. Moreover, to improve performances, programmers must resort to cache optimization techniques [13, 7].

The problem of accessing shared data can be generalized by considering shared resources. A mechanism for ensuring that only one process accesses a particular resource at a time is to establish sections of code involving the resource as so-called critical sections and arrange that only one such critical section is executed at a time. Several mechanisms, like the mutual exclusion, can be applied to ensure that the result of the operation will be correct. This is a classic problem covered in many textbooks [78]. In a shared memory system, parallel operations are executed concurrently by different threads. To ensure correctness in the critical sections of the code, programmers must implement synchronization barriers and thread-safe functions. Thread safe functions can be invoked from multiple threads simultaneously and always produce correct results. Typically, these functions exploit several types of atomic operations [1].

As discussed in chapter 2, NVIDIA GPUs have their own programming model. At the block level, the programmer can use barriers to enforce synchronizations among threads. Critical sections can be implemented via a set of dedicated functions that in CUDA are called atomic-functions.

However, in CUDA, does not exist a cheap mechanism to synchronize threads that belong to different blocks. The programming model encourages the development of programs that are a a sequence of distinct kernels. The

[1] In concurrent programming, an operation (or set of operations) is atomic, linearizable, indivisible or un-interruptible if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes. Additionally, atomic operations commonly have a succeed-or-fail definition, they either successfully change the state of the system, or have no apparent effect.

kernel itself acts like a synchronization barrier among all the blocks (there are several algorithms that benefit from this programming paradigm, as an example the prefix-sum, see [71]).

In the next section we introduce the pseudocode for the parallel BFS algorithm. Then we review recent works on both CPU and GPU architectures.

3.2 Overview of parallel algorithms for BFS

"Parallel BFS is similar to the sequential version, which starts with a source vertex s and visits levels of the graph one after the other using a queue to keep track of vertices that have not yet been visited. The main difference is that each level is going to be visited in parallel" (source: Blelloch [22]).

Algorithms (2) and (3) describe two versions of parallel BFS. The main difference between the two is that the former uses a queue whereas the latter doesn't. Algorithm 2 is a simple extension of the serial version where the **for** loop on line 6 is carried out in parallel (the parallelization of loop 7 is straightforward only on some architectures and, for the sake of simplicity, we will discuss it later). As for the corresponding serial version, it performs a linear amount of work, i.e., the time complexity is $O(N + M)$.

The pseudo-code 2 does not show that, to ensure correctness in a shared memory system, the parallel enqueue operation and the update of the parent array, must be implemented in a careful way. As a matter of fact, the update of the parent array gives rise to a benign race condition, but the enqueue operation requires special care to ensure correctness and to achieve good performances.

To clarify the issue, suppose to be in a shared memory system where different parallel operations are executed by different threads. Each thread

Algorithm 2 Level synchronous parallel BFS, with queues
CQ: Current level Queue
NQ: Next level Queue

```
1: p[u] = −1, ∀ u ∈ V
2: CQ, NQ ← ∅
3: p[s] = s
4: enqueue(CQ, s)
5: while CQ ≠ ∅ do
6:    for all u in CQ in parallel do
7:       for all v ∈ Adj[u] (in parallel) do
8:          if p[v] == −1 then
9:             p[v] = u
10:            enqueue(NQ, v)
11:         end if
12:      end for
13:   end for
14:   CQ ← NQ
15:   NQ ← ∅
16: end while
```

Algorithm 3 Level synchronous parallel BFS, without queues
C: Current level set, C has $|V|$ elements
N: Next level set, N has $|V|$ elements
V is the set of all vertices of the graph G(V,E).

```
1: p[u] ← −1, ∀ u ∈ V
2: C, N ← ∅
3: p[s] = s
4: C[s] = s
5: while C ≠ ∅ do
6:    for all u in C in parallel do
7:       if C[u] != 0 then
8:          for all v ∈ Adj[u] do
9:             if p[v] == −1 then
10:               p[v] = u
11:               N[v] = v
12:            end if
13:         end for
14:      end if
15:   end for
16:   C ← N
17:   N ← ∅
18: end while
```

is in charge for one vertex in the queue. In figure 3.1 is depicted the third iteration of algorithm 2 on the graph from section 1.3.1. The iteration starts with vertices $2, 4, 5$ in the CQ. Vertex 8 is discovered simultaneously by the two threads in charge of vertices 4 and 5. These two threads then enter the loop on line 5. Both will write $p[8]$ in an unpredictable order. However, this is a benign race condition, any thread writes, the resulting BFS-tree will be valid: the operation is idempotent. Once that the parent array is updated, threads enter the critical section, i.e., they have to update the queue (line 10). The enqueue operation must be realized with a safe thread-parallel function that, atomically, increases the queue counter, and then inserts the new element.

During the same iteration, also the vertices 1 and 3 are discovered by two distinct threads (see figure 3.1). However, the vertex 1 will not be added to the queue, because of the **if** instruction on line 8. (It is noteworthy that, in algorithm 2 and 3, the parent array is used to record the state of vertices, i.e. if they are visited or not, most implementations use a separate array to keep track of visited vertices).

In table 3.1 are shown the arrays used in the BFS algorithm (without copy removal) with their corresponding elements, for each iteration of the BFS. By comparison with table 1.1, it is apparent that the parallel version introduces redundant work. If multiple copies are allowed in NQ, the array may expand exponentially [22] and the resulting performances degrade. Addressing this problem is not easy and may require to redesign the algorithm.

Algorithm 3 solves this problem at the cost of performing a greater, asymptotically, amount of work.

In algorithm 2, at the beginning of each iteration, $|V|$ threads are created and each thread is in charge for one element of the array C. The algorithm

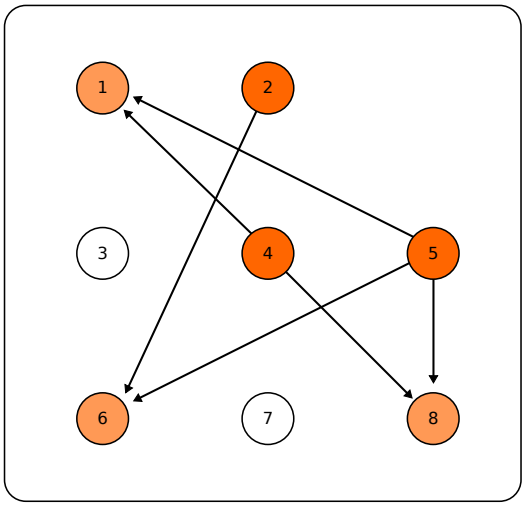


Figure 3.1. In figure is shown the third level of the BFS started at vertex 1. In parallel BFS the same vertex can be reached by two or more edges. As an example, vertices 4 and 5 find vertex 8 from two different edges, whereas vertices 2 and 5 visit vertex 6. This can lead to store and process redundant information during the run of the BFS.

Table 3.1. The table refers to the graph in figure (1.3). The BFS is started at vertex 1 and new visited vertices are added to next level, according to algorithm (2). The algorithm doesn't prune the NLFS from multiple copies of the same vertices (see text for details).

BFS LEVEL	CQ	NLFS	NQ
0	1	2,4,5	2,4,5
1	2,4,5	1,3,4,5,6,1,2,8,1,2,6,8	1,3,6,1,8,1,6,8
2	1,3,6,1,8,1,6,8	2,4,5,2,2,5,7,2,4,5,4,5,7,2,4,5	7,7
3	7,7	6,8,6,8	

starts, and each thread reads its element in the array C. Those threads, whose elements are not zero (lines 6-7), visit the neighbors of their vertices and set the parent when necessary (lines 8-13). Then, C and N are swapped. The algorithm stops when the array C is empty.

In this algorithm there is no need for a queue, which means that there are

no critical sessions. All the races among threads are benign. Moreover, the array C has a fixed size and doesn't have multiple copies of the same vertices. This strategy has his own shortcomings. At the end of each BFS level it is necessary to insert a synchronization barrier [2]. Even worse, $|V|$ threads are created and launched at the beginning of each iteration, regardless the number of the elements in the queue (or the number that have actually to be inspected during the iteration). In the worst case, the algorithm performs a $O(N^2 + M)$ amount of work.

3.3 Parallel BFS on multi-cores CPU

In the last few years, several studies of high performance computing, have tackled the problem of traversing a large graph with real-world properties (see Section 1.2).

Most often, the graph is generated by using special generators, like the R-MAT generator, that we have introduced in Section 1.2.1). We will refer to synthetic real-world graphs as R-MAT graphs. A new metric to evaluate performances has been introduced by the Graph 500 benchmark and has been adopted by most of recent works. This metric measures the number of Traversed Edges Per Second (TEPS) during the BFS visit.

Different solutions have been proposed to mitigate the effect of irregular memory access patterns, synchronization overhead and parallel insertion in the BFS queue.

In 2006, Bader and Madduri [13] designed a parallel BFS for the Cray

[2] Hong *et al.* noticed however that synchronization have a small impact if the input graph have real-world properties. First the number of levels is limited. Second, the computational-intensive parts are restricted to only a small fraction of levels.

MTA-2 architecture [3]. Their implementation is based on algorithm 2. Loops on line 5 and 9 are parallelized by suitable compiler directives. Their implementation uses optimized functions that atomically update the value of the distance array (that is the output of their BFS) and insert elements in the queue. They reported almost the same performance on R-MAT and random graphs. For an R-MAT graph with 200 millions vertices and 1 billions edges the code reached 0.5 GTEPS, by using 40 processors [7].

In 2010, Agarwal *et al.* [7] developed a multi-core multi-socket parallel BFS for Intel Nehalem platforms. They implemented a level synchronous BFS like algorithm 2. The first optimization they introduced is the use of a global bitmask to mark visited vertices. This greatly reduces the working size of the inspected set of vertices. They reported an improvement of the processing rate by a factor of four (number of reads per unit time). The graph and the bitmap were partitioned thorough the CPU-sockets so that only local vertices were updated locally, information about non-local vertices were exchanged. They noticed that those communications affect the performance and developed a lightweight communication mechanism among groups of cores residing on different sockets. With a 64 threads enabled Nehalem EX, they reported 1 GTEPS for a R-MAT graph with 128 million vertices and 4 billion edges.

An interesting algorithm has been proposed in 2011, by Beamer et al. [20] for the Graph 500 benchmark. They introduced a bottom-up approach in the BFS visit. Basically, they noticed that, during each level of the BFS there are a great number of wasted attempts to become a parent of a neighbor. By using a bitmap to mark visited vertices reduces those attempts but there is still redundant work, because each vertex on the frontier tries to become

[3]Shared memory system

the parent of its neighbors. Instead of doing this, in their algorithm, each unvisited vertex attempts to find any parent among its neighbors (a neighbor is a parent if it is part of the frontier). In this approach, each child writes by itself its parent and there is no need of atomic operations. Their results are really remarkable, with a quad-socket 40-core Intel Xeon E7-8870 they report 5.1 GTEPS for a graph with 256 million vertices and 4B billion undirected edges. They ranked 19 on the Graph 500 list, in November 2011.

Other solutions have been proposed in [36, 68, 38]. Optimizations are always directed to improve bandwidth utilization and cache performances.

3.4 Parallel BFS on GPU

Many studies, in recent years, demonstrated that applications having a regular access pattern in memory and a high arithmetic intensity (ratio between number of arithmetic and memory access operations) can be successfully ported to GPU with a significant speed up in the execution time. Algorithms with irregular memory access patterns, however, have been proved to be more challenging and exhibit less spectacular improvements.

The main drawback, with algorithms that have an irregular "flow" is that data structures vary at running time. During the BFS visit, at each iteration, the size of the queue and of the NLFS vary greatly.

In the CUDA programming model [4], the number of threads and blocks is statically assigned before the kernel starts and cannot be modified at run time. Thus, the programmer typically implements a static mapping between threads and data elements. With a static mapping, there is the risk of having either too few threads, thus serializing most of the work, or too many threads,

[4]CUDA 5.0, that has been recently released, supports dynamic thread allocation.

thus incurring in an unnecessary overhead for the generation and the release of resources.

For instance, a naive way to map threads to data is to assign a thread (defined on a 1D *grid* in the CUDA sense) to each vertex in the queue so that the loop in line 6 of algorithm 2 will be executed in parallel. With this assignment, the loop in line 7 is serialized, thus limiting the number of active threads to the number of elements in the CQ.

Several works, having as target a single GPU, have adopted a different strategy, based on algorithm 3. The static assignment of tasks to vertices trivially maps to the data parallel GPU model. The work flow of each thread is independent from other threads. Those implementations suffer two main problems. First, the overhead associated with the launch of a constant number of threads, even if they are not necessary. Second, the number of threads is actually too small to visit the NLFS in parallel. In the most computationally-intensive levels of the BFS, the work is almost serialized.

To bypass the problem of the static mapping, the resources actually required, can be computed at run time. For instance, once the CQ is built, it is possible to calculate the total number of elements in the NLFS. This implies that, in principle, we can run a kernel that computes the number of threads and their offsets. Each offset corresponds to the index of the element to which the thread will be assigned and will be used to correctly map threads to data. Then, a second kernel, can actually performs the work (it can be the status look-up of vertices in the NLFS). It turns out that, in some situations, it is convenient to adopt such parallelization-strategy, instead of using the static assignment. Unfortunately, the implementation of such strategy, is not straightforward.

Summing up, the main issue, on a single device, is finding the right map-

ping of data to threads so that the full power of the GPU is exploited. The irregular flow of the algorithm, limits the number of concurrent working threads. Compared to the CPU, another possible issue is the amount of global memory that is limited to 6 GBytes for the latest NVIDIA FERMI GPUs (whereas commodity CPUs can have hundreds of GBytes). This limits the size of the graph that can be visited.

In 2007, Harish *et al.*, [37] implemented a parallel BFS based on algorithm 3, by using a CUDA enabled GPU (Nvidia GTX 280 with 1028 MByte of memory). To mitigate the overhead of having $|V|$ threads running at each BFS iteration, they implemented an optimization, based on vertex-list compaction, that reduces the number of active threads. The synchronization between two subsequent levels of the BFS is implemented by splitting the problem in two kernels. The first kernel uses two arrays that hold, respectively, the old and the new frontier, in order to prevent read-after-write inconsistencies. The second kernel swaps the frontiers and update the visited array. Due to the limited size of the device memory, the size of the graph they can visit is small, compared to the ones that can be visited with multicore CPUs. They reported an execution time of 0.5 sec for a R-MAT graph with 10 million vertices and 120 million edges.

In 2009, Deng *et al.,* [29] implemented the graph traversal by means of the Sparse-Matrix Vector product (SMVP). They developed their own implementation of the SMVP on CUDA. Their target are Electronic Design Automation (EDA) applications, thus, the size and the kind of the graphs they visited cannot be directly compared to those we have seen so far.

In 2010, Luo *et al.,* [50] developed a new algorithm to perform a BFS on a CUDA GPU (Nvidia GTX280). Their algorithm is queue-based, like algorithm 2. However, to avoid the critical section of the parallel queue

insertion, they introduced a hierarchical queue structure.

The idea is that, once the lower-level queues have been created, then the location of each element in the higher-level queue are also known and it is possible to copy the elements to the higher-level queue in parallel. To completely avoid collisions in the building of the hierarchical queue, they implemented the first queue-level at the warp level (see section 2.2). It is important to notice that to guarantee correctness the low level insertion in the queue is achieved via an atomic operation. They reported effective speed-up with respect to the implementation in [37] for all the input graphs.

For instance for a scale-free graph with average degree ~ 6 and $10M$ vertices they reported a running time is 0.483s. However, for this kind of graph, they need to pre-process the graph and convert it to a near-regular graph, by splitting the big-degree nodes (the same idea has been previously used in [55]).

In order to achieve better performances, in 2010, Hong et al. [41] introduced a warp-centric programming model. Instead of assigning different tasks to each thread, they allocated a chunk of tasks to each CUDA warp. During neighbors expansion, the SIMD lanes of the warp, are used to visit the adjacency lists of vertices assigned to the warp. They tackled directly the mapping of data to threads, thus obtaining better results at the cost of greater programming difficulties.

In a subsequent work [42] they proposed a hybrid CPU/GPU method that takes advantage of the GPU only for the most computationally-expensive levels of the BFS. They implemented algorithm 3, on the CPU and used the optimization proposed in [7]. For the CPU (Nehalem Xeon X5550) they reported nearly 0.8 GTEPS for an RMAT graph with 32 million vertices and 1 billion edges. The CPU+GPU version shows some improvements with

respect to the CPU only version but cannot be tested on graph of such size. They reported 0.9 GTEPS for 32 million vertices and 240 millions edges on a Fermi Tesla 2050.

In 2011, Merrill *et al.,* [30] noticed that GPU architecture is not well suited for problems that require dynamic and irregular data movement within shared data structures. Their work is focused on parallelization strategies that permit to map in an effective way threads to data, given their dynamic allocation requirements. They suggest that, an efficient prefix sum operation allows for a reorganization in which a sparse and uneven work becomes an uniform and dense one. Moreover, they individuate as the basic element of computation the CTA (Cooperative Thread Array), *i.e.,* a block in the CUDA programming model. Instead of assigning data to each thread, they assigned chunks of data to the CTA.

Their work is the first that incorporates fine-grained (not global) parallel adjacency list expansion at the CTA level. This means that large neighbors lists are cooperatively strip-mined at the full width of the CTA. Another important feature is the local duplicate detection which eliminates most of the race conditions and redundant work. They reported a detailed analysis of the expansion and contraction mechanism of the BFS over subsequent levels pointing out that, the removal of duplicates, can reduce the number of vertices in the NLFS by one order of magnitude. To achieve better performances and also reduce the memory occupancy, they implemented a global bitmask array to keep track of visited vertices. With a NVIDIA TESLA 2050, they report 1.8 GTEPS for an R-MAT graph wit 2 million vertices and 32 million edges. Their GPU implementation of the BFS is the fastest currently available (with really remarkable performances) but the size of the supported graphs remains bounded by the GPU memory size.

In the same paper the authors developed a multi-GPU version of the code. The multi-GPU implementation relies on the VMA technology that supports up to four devices, with a unified memory address space. By using four GPUs, they report a result of 8 GTEPS, however, this result, refers to the visit of a graph with an average degree equal to 256, a pretty high value. For an input graph with an average degree equal to 16, the code does not exceed 3 GTEPS. This value can be considered as a marginal improvement with respect to the speed-up that the same authors report comparing their results with those achievable on a single CPU (as an example with the result in [7]). As a matter of fact, the reported strong scaling, discussed in [30] is not spectacular: "We observe 1.5x, 2.1x, and 2.5x speedups when traversing a R-MAT graph with 2 million vertices and 128 million edges using two, three, and four GPUs, respectively" [5]. Our distributed implementation have a similar speed-up but using an Infiniband interconnection. In Chapter 4 (see section 4.3.5) we provide a more detailed comparison and discussion.

Authors	Graph Type	Num. of Vertices	ef	GTEPS	Num Processors	Arch. Type	Output
Agarwal [7]	R-MAT	2^{21}	16	0.6	2 sockets	Nehalem EP	parent
Agarwal [7]	R-MAT	2^{21}	16	0.65	4 sockets	Nehalem EX	parent
Hong [42]	R-MAT	2^{25}	8	0.4	2 sockets	Nehalem X5550	distance
Hong [42]	R-MAT	2^{25}	8	0.64	1	Tesla C2050	distance
Hong [42]	R-MAT	2^{25}	8	0.68	1	CPU+GPU	distance
Hong [42]	R-MAT	2^{21}	8	0.6	1	CPU+GPU	distance
Merrill [30]	R-MAT	2^{21}	16	1.8	1	Tesla C2050	distance
Merrill [30]	R-MAT	2^{21}	16	3.2	4	Tesla C2050	distance
Merrill [30]	R-MAT	2^{24}	16	3.0	4	Tesla C2050	distance

Table 3.2. Comparison of different implementations of BFS on shared memory systems. The column ef is the average degree so that, the number of edges is ef times the number of vertices. While all the input graphs are R-MAT, the exact values of the coefficients are available only for [30]. The output of the algorithm is also important when comparing performances: the computation of the distance array is faster with respect to the computation of the parent array. We tried to compare similar instances of input whenever possible.

[5] from [54]

Authors	Graph Type	Num. of Vertices	ef	GTEPS	Num Processors	Arch. Type	Output
Bader [13]	R-MAT	2^{27}	5	0.5	40	Cray MTA-2	distance
Agarwal [7]	R-MAT	2^{20}	16	1.1	2 sockets	Nehalem EP	parent
Agarwal [7]	R-MAT	2^{22}	64	1.3	4 sockets	Nehalem EX	parent
Hong [42]	R-MAT	2^{25}	16	0.9	1	CPU+GPU	distance
Hong [42]	R-MAT	2^{25}	64	0.93	2 sockets	Nehalem X5550	distance
Merrill [30]	R-MAT	2^{21}	64	8.3	4	Tesla C2050	distance
Merrill [30]	R-MAT	2^{20}	256	3.5	1	Tesla C2050	distance
Beamer [20]	R-MAT	2^{28}	16	5.1	4 sockets/40 cores	Westmer-EX	parent

Table 3.3. Best performances reported by different authors for BFS on shared memory systems.

In tables 3.2 and 3.3 we report the results of some of the works described so far. The column *ef* is the average degree of the input graph. The number of edges is *ef* times the number of vertices (the term *ef* stands for *edgefactor*, see section 4.2.1). The first table is a comparison of similar results (similar size of the input graph and similar average degree) whereas the second table shows the best performances achieved by the various implementations discussed. Results obtained by [20] and [30] are noteworthy. Unfortunately, both implementations relies on optimization techniques that cannot be used in a distributed implementation. We will discuss this topic in more details in Chapter 4, after we have introduced the basic concepts of the distributed version of the BFS algorithm.

Chapter 4

Parallel BFS on distributed memory systems

Very large graphs do not fit the memory of a single system. To study them it is necessary to resort to a distributed memory architecture. Generally, a distributed system is a cluster of computing nodes interconnected via a wired network. Each node is a system in itself that can be equipped with a single- or a multi-core CPU. To carry out computations, the nodes of the cluster, must exchange data each other. Graph algorithms have low arithmetic intensity, that is, during the execution, the time spent in computation is a small fraction of the whole. On a single processor, most of the time is spent in read and write operations from/to memory. In a distributed environment, data can be in a remote memory, thus, most of the execution time is spent sending and receiving data over the communication network. It is noteworthy that, latencies involved in communication are very high, compared to those introduced by the access to data in memory. Moreover, the communication patterns involved in graph algorithms, are irregular. Both the size of the

messages and the set of senders/receivers vary during the execution.

It is not surprising therefore, that, as reported by many authors, [79, 51, 23], the bottleneck of a distributed BFS, is the communications among nodes.

The optimization of the communication among tasks is crucial for an efficient BFS algorithm on a distributed architecture. In our case, we need to take into account also the specific features of the computing node, that is a GPU. Unfortunately, most of the optimizations described in section 3.3 and 3.4 are not applicable. In a distributed cluster of GPUs, it is not possible to use an algorithm based on the pseudo-code in 3. For that parallelization-strategy, the current and the next level frontier must be an array of exactly $|V|$ elements. Then, a trivial static mapping, makes use of a thread for each vertex in the graph. However, our goal is to visit a graph whose size is such that the number of vertices $|V|$ is too high to store a global array of size $|V|$ in the memory of a single node. Vertices are scattered among nodes and each node only holds a subset of the whole graph (it is apparent that vertices distribution requires some care. The final number of edges assigned to each task must be balanced.)

Harish [37] and Hong [42] used the static mapping. Hong, Agarwal, Merrill and Beamer [42, 7, 30, 20] used a global bitmask array to mark visited vertices. The bitmask highly reduces the size of the global array, however, that solution is not scalable. The maximum size of the graph would be limited by the maximum size of the array that fits the device memory. For instance, the Graph 500 benchmark, provides a maximum size of 2^{40} vertices. Even using a bit for each element requires at least 128 GBytes of memory that is much more than the size currently supported by a GPU.

In addition, all the shared memory optimizations speed-up the visit of

local vertices. In the distributed problem, however, the time spent to execute this operation is only a small fraction of the total running time which is dominated by the part of the algorithm that copes with non-local vertices.

Algorithm 4 shows the pseudocode of a distributed BFS. The root vertex is randomly selected and then, the BFS search starts locally on the task in charge of the root and propagates to other tasks as the NLFS expands through the graph. Tasks with vertices in the NLFS perform a local frontier advancement, exchange information about other vertices with the corresponding owner tasks and update parents, if needed.

With respect to the shared memory versions, the distributed version presents a new computational part, the building of the array to send (line 22), the communication part (line 25-26) and the filtering of received vertices (line 27-32).

To reduce the communication burden, it is possible to implement different strategies. In the following section we review some of them from recent studies. It is noteworthy, that, as far as we know, there are no other implementations of BFS on a distributed GPU cluster.

Algorithm 4 distributed memory BFS

CQ is the current level queue.
NQ is the next level queue.

Require: s (starting vertex)
1: $CQ \leftarrow \emptyset$
2: $NQ \leftarrow \emptyset$
3: $d[u] \leftarrow -1, \forall\ u \in V$
4: $p[u] \leftarrow -1, \forall\ u \in V$
5: **if** s **is local then**
6: $d[s] = 0$
7: $p[s] = s$
8: $enqueue(CQ, s)$
9: **end if**
10: $totlen \leftarrow 1$
11: **while** $totlen > 0$ **do**
12: $u \leftarrow dequeue(CQ)$
13: $sendarry \leftarrow []$
14: $recvarry \leftarrow []$
15: **for each** $v \in Adj[u]$ **do**
16: **if** v **is local then**
17: **if** $p[v] == -1$ **then**
18: $p[v] = u$
19: $enqueue(NQ, v)$
20: **end if**
21: **else**
22: $sendarry.append((u, v))$
23: **end if**
24: **end for**
25: $SEND(sendarry)$
26: $RECV(recvarry)$
27: **for each** (z, w) **in** recvarray **do**
28: **if** $p[w] == -1$ **then**
29: $p[w] = z$
30: $enqueue(NQ, w)$
31: **end if**
32: **end for**
33: $CQ \leftarrow NQ$
34: $NQ \leftarrow \emptyset$
35: $totlen = allreduce(size(CQ))$
36: **end while**

4.1 Related Works

In 2006, Yoo et al. presented a distributed BFS algorithm for the IBM BlueGene/L, a distributed system with 32,768 nodes, hosted at the Lawrence Livermore National Laboratories [79]. Their work was focused on reducing communication overheads by means of a two dimensional partitioning of the graph. This partition greatly reduces the number of processors involved in collective communications. They have also developed ad-hoc collective communication functions for the 3D torus network of BlueGene/L. The code was able to traverse a graph with 3 billion vertices and 30 billion edges, by using thousands processors. They reported a minimum of 80 million edges per second (MEPS), for graphs with low average degrees and a maximum of 700 MEPS, with high average degrees. However, their implementation, assumes an input graph with regular degree distribution, the scalability they obtained may not be achievable with a graph with a skewed distribution.

In 2011 Buluç *et al.,* [23] proposed a parallel BFS on a CPU-based cluster (Hopper, AMD platform, 40000 cores). They implemented a two dimensional decomposition, directly on the sparse matrix that represents the graph in Sparse Matrix Vector Multiplication (SpMV) form. They reported 17.8 GTEPS for an undirected graph with 4.3 billion vertices and 68.7 billion edges.

In 2012, Ueno *et al.,* [73], developed an optimized version of the Graph 500 benchmark. By using 1366 nodes and 16,392 CPUs, they visited a graph with 2^{36} vertices and 2^{40} edges and obtained the impressive result of 103 GTEPS. The base algorithm is a level synchronous BFS represented as SPMV with a 2D decomposition. Further optimizations include the parallelization of the send/recv operations and an improvement of cache utilization via the sorting

of the visited bitmask array by decreasing the vertex degree. Their paper also reviews the reference implementation and propose optimized methods for the construction and validation phases of the benchmark. Even with those impressive performances they reported communications as the bottleneck of the implementation.

In 2012, Lv *et al.,* [52], describe an MPI implementation of the Graph 500 benchmark in which the key idea is to keep events as asynchronous as possible. They have separated communications from computation. Communication is assigned to a master thread and computation to many traversal threads. The algorithm is implemented by using MPI + Pthreads (POSIX threads) on a standard Linux environment. The current Nehalem platforms, allow a maximum of ten concurrent memory requests. This feature can be used on memory-bound algorithms, like BFS, by using a massive number of threads, greater than the number of cores.

On a multi-core cluster of Xeon X5650, by using 2048 threads and 32 MPI processes, they visited a graph with 2^{30} vertices at the rate of 1.45 GTEPS.

In a later implementation, Lv *et al.* [51], tried to reduce communication by working on the data structure that represents the graph. They implemented the global NLFS as a bitmap array and compressed it, to reduce the size of messages. To further improve the compression ratio they implemented a directory to sieve the bitmap and make it even sparser for compression. By using 512 nodes Xeon X5650, the code traversed a graph with 2^{33} vertices, achieving 12 GTEPS. As for the other implementations we have seen so far, the communication among nodes remains the most time-consuming part and accounts for $\sim 70\%$ of the total running time.

We wish to highlight that all of these studies, with the exception of [79] have been carried out in the last two years. Because they are contemporary

to our study, we could not use the solutions they have introduced. On the other end, it was clear from the beginning of our study, that, in order to develop a scalable code, we had to address the issue of the communication.

4.2 BFS on a multi-GPUs architecture

As a first step to gain a better understanding of the problem, we developed a straightforward implementation of the distributed BFS problem. Our work follows the Graph 500 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 an R-MAT 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.

In the following sections we compare our GPU implementation to the reference CPU implementation provided by the Graph 500 benchmark [1]. The reference code is a multi-CPU implementation of a distributed BFS. The communication among nodes is implemented by means of MPI. Communication and computation are overlapped by using fixed size buffers for the messages. The size of the buffers is tuned so that each task has enough vertices to process locally while the next chunk of non-local vertices are exchanged.

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;

[1] We used version 1.2 because more recent versions either fail to run or are too slow.

ii) each tree edge connects vertices whose BFS levels differ by exactly one; *iii)* every edge in the input list has either vertices with levels that differ by, at most, one, or both vertices out of the BFS tree; *iv)* the BFS tree spans an entire connected component's vertices, and *v)* a node and its parent are joined by an edge of the original graph.

4.2.1 Graph generation

We generate a synthetic graph according to the Graph 500 guidelines by using the RMAT generator [25, 47]. To characterize the size of the graph, the benchmark uses two parameters: $SCALE$ and *edgefactor*. The number of vertices in the graph is given by 2^{SCALE} whereas the number of edges is $edgefactor \times 2^{SCALE}$. The value of $SCALE$ ranges from 26 to 42 whereas the *edgefactor* is fixed to 16. 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.

Since the graph must be undirected, we double the number of edges (self loops are not replicated). We generate $N = 2^{SCALE}$ vertices and $M = 16 \times N$ edges. Each edge joins two vertices, so the total number of elements is 32×2^{SCALE}. On a single Nvidia GPU that currently may have up to 6 GBytes of *global* (*i.e.,* main) memory, the maximum $SCALE$ can be 24. To carry out a BFS, additional data structures are needed so that maximum $SCALE$ can not be reached. Moreover the Graph 500 specs require that, once created, the data structure can not be modified. In the end, the maximum $SCALE$ of the Graph 500 benchmark that we are able to run on a single device is 21.

We resorted to the distributed generator provided by the Graph 500 group as part of their reference code. It is a distributed CPU-MPI implementation of an RMAT generator (here and in the following we refer to the *simple*

reference MPI implementation, version 1.2 [2]).

Edges are assigned to tasks via a simple rule: edge $(U_i, V_j) \in P_k$ if $U_i\%\#P == k$, where $\#P$ is the number of tasks and $\%$ is the modulus operator.

4.2.2 Distributed data structure

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 is simple and has reduced memory requirements. The CSR data structure is composed by two arrays, an array of offsets (*Offset Array*) and an array (*Adjacency Lists*) that contains the adjacency list of all the vertices in the graph (see figure (4.1) panel B).

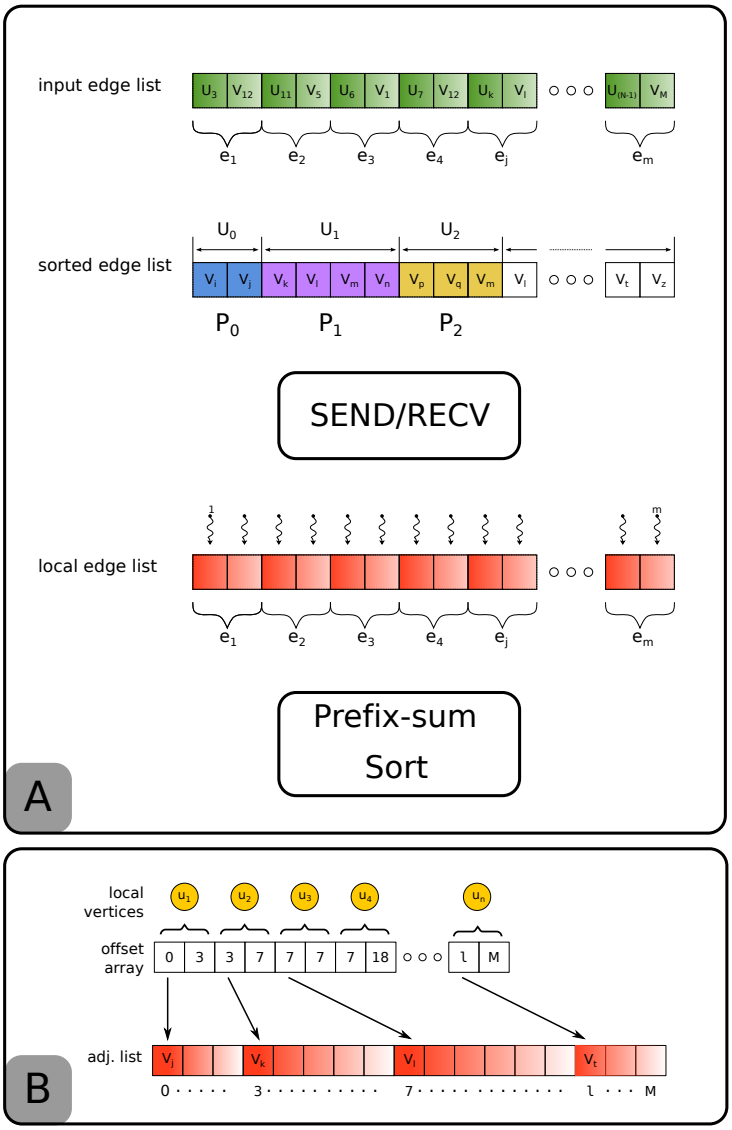


Figure 4.1. Panel A depicts the data structure generation procedure. Initially, the *input edge list* is sorted according to the first vertex U, partitioned among processors and non-local data are sent to the corresponding processors. On the receiving side, each processor collects the data in a *local edge list*. This list is then sorted (as in an previous step) and a prefix-sum operation is used to build the the CSR data structure represented in Panel B.
Panel B shows the Compressed Sparse Row data structure. 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. For convenience we store also the last index in the *Offset Array*.

The algorithm that builds the data structure from the input edge list (generated as discussed in the previous section) is represented in figure 4.1. The use of the prefix-sum operation, to compute offsets and to dynamically map threads to data, permits to achieve a great level of parallelism and good performances. Although we do not present the details of the generation of the data structure, we report, for this part, a speed-up of about two order of magnitude with respect to the MPI-CPU reference code provided by the Graph 500 group (see figure 4.2).

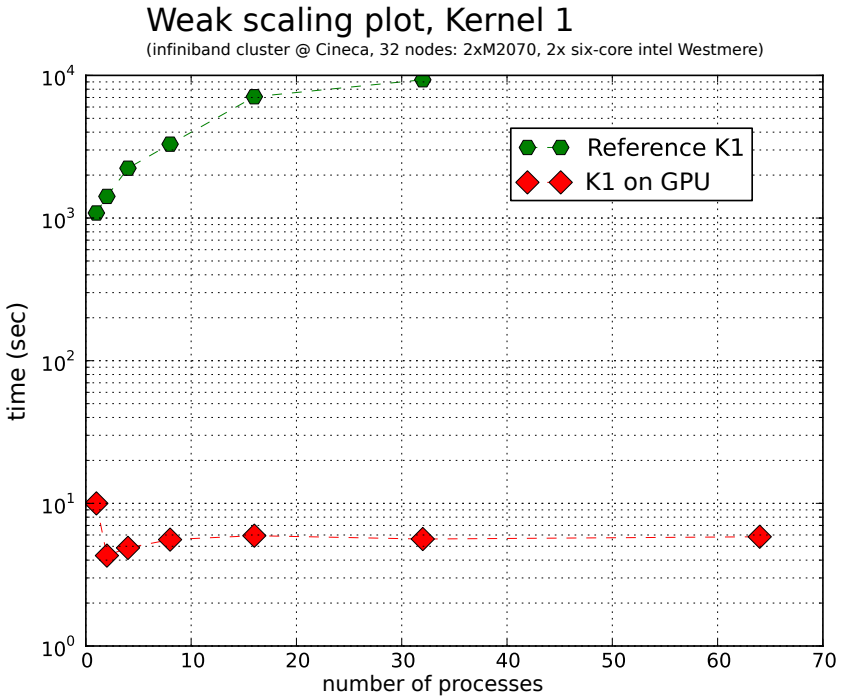


Figure 4.2. Weak scaling plot of the time required to build the CSR from the generated edge list (Kernel 1 of the Graph 500 benchmark). Our multi-GPU implementation outperforms dramatically the multi-CPU implementation of the reference code provided by the Graph 500 group.

4.2.3 Straightforward implementation

A straightforward way to implement a distributed BFS, on a multi-GPUs cluster, is to use a queue-based method with atomic operations. In the present work, the output of the BFS is the parent array or array of predecessors. Finding the parent array is more expensive than computing the array of distances, since information about predecessors must be stored and exchanged at each BFS level.

The current level queue and the next level queue are maintained as two separate arrays. Two additional arrays are required for sending and receiving vertices. Those arrays are very large, their size [2] limits the overall size of the graph that can be held locally. At each BFS level all vertices in the next level set are visited in memory, without the need of extra arrays.

Each vertex in the queue is assigned to one CUDA thread. Each thread visits the neighbors of its vertex and, first of all, verifies if they are local or not. If neighbors are not local, they are sent to the respective owners along with their parents. For local neighbors, the parent array is checked, to see if they have been already visited. Vertices that have never be visited, are added to the next level queue. To maintain consistency of the queue, the enqueue mechanism relies on atomic operations. Each task sends and receives edges; for each received edge, it checks and enqueues the first vertex of the edge, if necessary (figure 4.3 A).

Despite of its simplicity, this algorithm has many issues: first the workload is not balanced among the threads. As shown in figure (4.3) B, thread 1 visits an adjacency list with L_1 elements whereas thread 2 visits an adjacency list with L_2 elements. In "real world" graphs, the two adjacency lists L_1 and L_2

[2] We found that, for an undirected graph with M edges, each of the two arrays must have a minimum of $5 \times M$ entries.

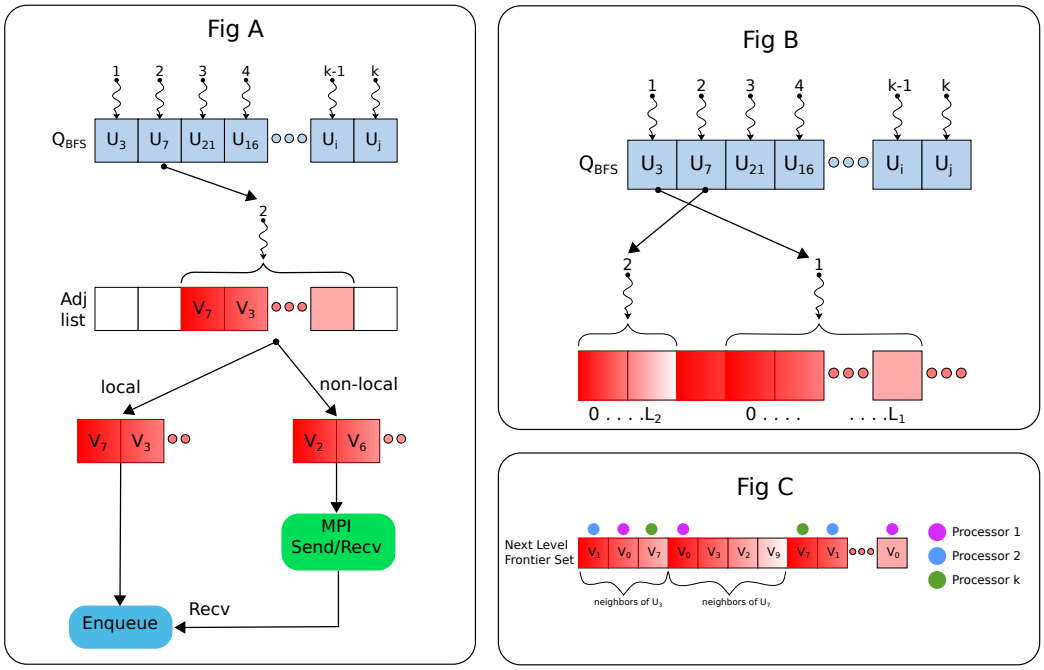


Figure 4.3. Panel A): Straightforward BFS: algorithm execution flow. Each vertex in the queue is associated with one CUDA thread. Each thread visits the adjacency list of its vertex (see text for explanation). Panel B): Straightforward BFS: Issues. Threads workloads are unbalanced. t_1 visits L_1 elements whereas t_2 visits L_2 elements. Panel C): The NLFS, the array of neighbors, contains multiple copies of the same vertex

may differ by orders of magnitude. Moreover, memory access patterns are typically irregular and threads that belong to the same warp may need to access memory regions that are non-contiguous and/or far away each other. Finally, the number of memory accesses depends on the number of elements in the NLFS, which can be much greater than the number of threads.

Besides that, there is a communication issue. During phase 1 (figure (1.4)) the queue is expanded to the NLFS, a set that is built from the adjacency lists of each vertex in queue. In the CSR data structure, like in any simple representation of a graph, the adjacency lists contain multiple copies of the

same vertices (see figure 4.3 C). As a consequence, the NLFS may contain several copies of the same vertices. Those copies are sent directly to their owners, thus producing a useless communication overhead. Another potential issue is that the algorithm relies on atomic operations that are notoriously expensive on GPUs.

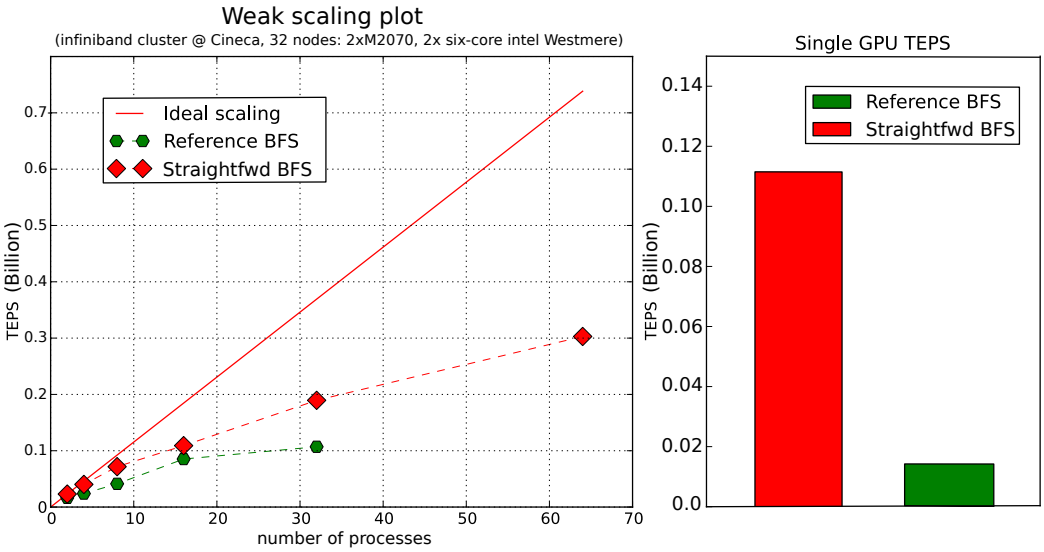


Figure 4.4. Straightforward multi-GPU implementation versus multi-CPUs reference implementation. Left panel: On y-axis the TEPS, on x-axis the number of tasks. The multi GPU code (in red) shows some improvements. On the right: single GPU implementation of the straightforward algorithm.

Results

Figure (4.4) is a weak scaling plot of the Traversed Edges per Second (TEPS) during the BFS visit obtained with a straightforward multi-GPU implementation. As a comparison we report also the results of the reference code provided by the Graph 500 group. The plot shows that there are some improvements by using GPUs but they are not spectacular. The right panel

shows the result of a single GPU implementation of the straightforward algorithm. We highlight that, to have the same performances with the distributed version, we need 16 GPUs. This gives a measure of the amount of extra work required by the distributed implementation. The main reason is the inability of processing non local edges so we have first to distinguish and then exchange them. Basically in the distributed algorithm the running time is dominated by the part dealing with non-local-edges.

4.3 Optimized BFS on a multi-GPU platform

4.3.1 Motivation

The straightforward implementation, discussed previously, helped us to identify three main issues: the unbalanced workload among threads, the use of atomic operations and the communication of duplicated data. The first two refer to the parallelization on a single GPU whereas the last one has been reported in several papers that deal with a distributed implementation of the BFS (discussed in section 4.1).

The workload imbalance is a direct consequence of the trivial mapping employed in the straightforward algorithm that assigns threads to vertices in the BFS queue. The problem is exacerbated for the graphs we consider. For such graphs the number of elements in the queue and the number of elements that have to be visited at each BFS level (namely the elements of the NLFS) can differ by orders of magnitude. As far as we know, there are no general solutions to address balancing problems of this kind. In [30] it is described a sophisticated approach that reduces the load imbalance by mapping threads to data at the CTA level (i.e. the CUDA block level, see

section 3.4). However, the optimization relies on the usage of a bitmap.

We tackled the problem directly with the aim of fully exploiting the GPU parallelism. We assume that a good solution would be having as many active threads as the number of elements in the NLFS so that each thread is in charge of only one vertex and the whole NLFS can be processed in parallel. As shown in figure (4.5) mapping threads to NLFS elements is not trivial. In the following sections we will describe the details of our novel technique to map threads to data that achieves a perfect load balancing by employing a prefix-sum operation and a binary search function. Our mapping allows for building, in parallel, a contiguous array that represents the NLFS. Once the array is available, several techniques can be applied to reduce the communication overhead. As pointed out in section 4.2.3, the NLFS may contain several copies of the same vertices. Those copies are sent directly to their owners, thus producing a useless communication overhead. Multiple copies can be removed by simply perform a pruning operation. We implemented it by means of a combination of Sort and Unique operations that remove all the duplicates from the NLFS array. This strategy has two major advantages: first it reduces the number of exchanged elements and consequently the number of processed vertices. Moreover, it reduces the number of atomic operations required to enqueue local vertices. Actually, by performing the pruning operation on the whole NLFS we remove also multiple copies of local vertices and thus we reduce the number of elements that need to be processed during the local enqueue phase.

4.3.2 Algorithm overview

The algorithm is queue-based and returns the parent array. As in the straightforward approach, we use two arrays to store the current and the

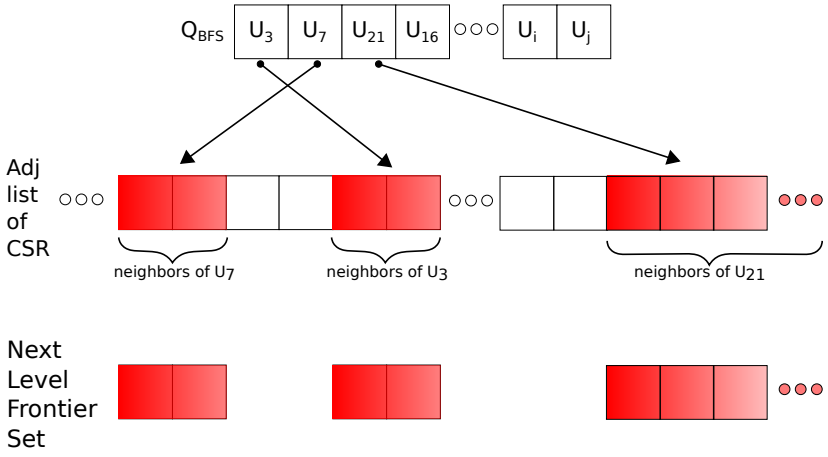


Figure 4.5. The entry of the NLFS are not contiguous in the *Adjacency List* of the CSR data structure. The operation of mapping threads to the NLFS elements is not trivial.

next level queue, plus two arrays to store edges that need to be sent and received. Starting from the queue, we build an array of offsets and compute m, the total number of elements in the NLFS. Then we start m threads. Each thread computes the CSR index of the NLFS element that the thread will handle. We read the NLFS in parallel and prune it from multiple copies of the same vertices. Then we exchange vertices with other tasks, visit new vertices and update the parent array. The algorithm stops when all the queues are empty. In each iteration of the BFS the algorithm performs the following steps:

A) **Building an array of offsets and computing m the total number of elements in the NLFS**. For each element in the current level queue, we start one thread. We build the array Q_{degree}, by substituting each vertex with its degree. Then, we perform a prefix-sum operation on Q_{degree} to build the *NewOffset* array. The last element of *NewOffset*

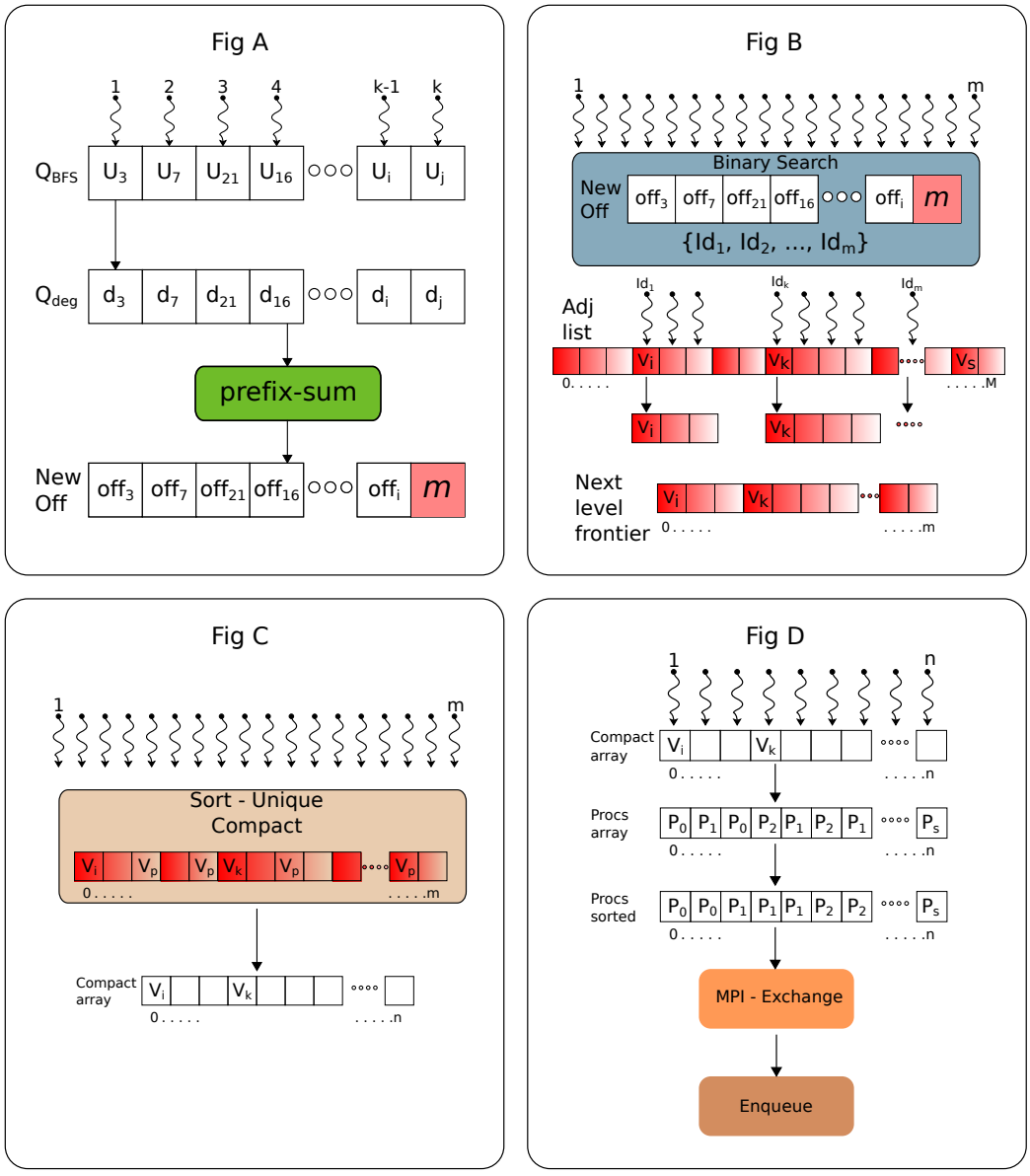


Figure 4.6. Panel A, B, C, D represent the steps performed by the algorithm (see text for details).

is the total number of elements, that we call m, in the NLFS. We also build another array, Q_{offset}, by substituting each vertex with its starting

offset in the CSR data structure. The Q_{offset} array is necessary to carry out the next step (figure 4.6 A).

B) **Mapping threads to entries of NLFS: building a contiguous array for the NLFS.** In this step we use m threads. Each thread performs a binary search on the *NewOffset* array and, by using the old offset stored in Q_{offset}, computes the index of its entry in the *Adjacency List* array of the CSR, as follows:

```
i = binsearch(NewOffset, thread_id, nelements);
t_off = thread_id - NewOffset[i];
index = Qoffset[i] + t_off;
```

Where *thread_id* is the global thread identification index and *nelements* is the number of elements in the *NewOffset* array. The *binsearch* function returns the index of the entry in the *NewOffset* array, whose value is lower or equal, to $thread_{id}$.

Then, each thread reads from the *Adjacency List* the element corresponding to the index and writes it in a new array: the *Next Level Frontier* array (that has m elements). In the end, we have a contiguous array of all neighbors for the given queue (figure 4.6 B).

C) **Pruning of the *Next Level Frontier* array.** We use m threads to carry out a sort of the *Next Level Frontier* array. When the array is ordered it is easy to compact it to n unique elements. It is important to realize that the ratio between the number of elements after and before the pruning operation can be very small (see figure 4.7). This part of the code is quite demanding in terms of memory. Since we want

to build the parent array, we have to carry the information about the parent of each vertex in the *Next Level Frontier* array, that is we have to sort the vertices keeping the payload of the parents (figure 4.6 C).

D) **Exchange of vertices with other tasks and update of the parent array.** We store all edges to be sent in the array *EdgesToSend* and sort it with respect to the owner of the first vertex of each edge. Each task i sends to each other task j the elements of its array *EdgesToSend* that belong to it (that is, whose owner is task j) while keeping its own part. Then the task waits until it receives all the edges it owns from other tasks and collect them in *EdgesRecv*. This array, along with the local part of the *EdgesToSend* array, obviously contains only local vertices. In the end, local vertices that have never been visited, are added to the next level queue. We highlight that the number of elements to be sent and received differs from task to task and among different BFS levels. To manage this situation, MPI collective primitives are used, at each BFS level, to know the actual number of vertices to be received.

Simple classic models (like PRAM) can hardly provide realistic bounds for the performances on modern parallel architectures. However, to gain some insight on the complexity of the presented algorithm we evaluated the complexity of the four phases described above, as if they were executed by a single task. The most expensive operation of the first two steps is the binary search that performs $O(|M|log(|V|))$ operations in the worst case. In step C the sorting operation, implemented via a radix-sort has a worst case complexity $\simeq O(64 * M)$ since each element is encoded as a 64-bit integer. The last step D has complexity $\simeq O(M + N)$. Since $log(|V|)$ is always less than 64, the overall complexity is bounded by $O(M + N)$.

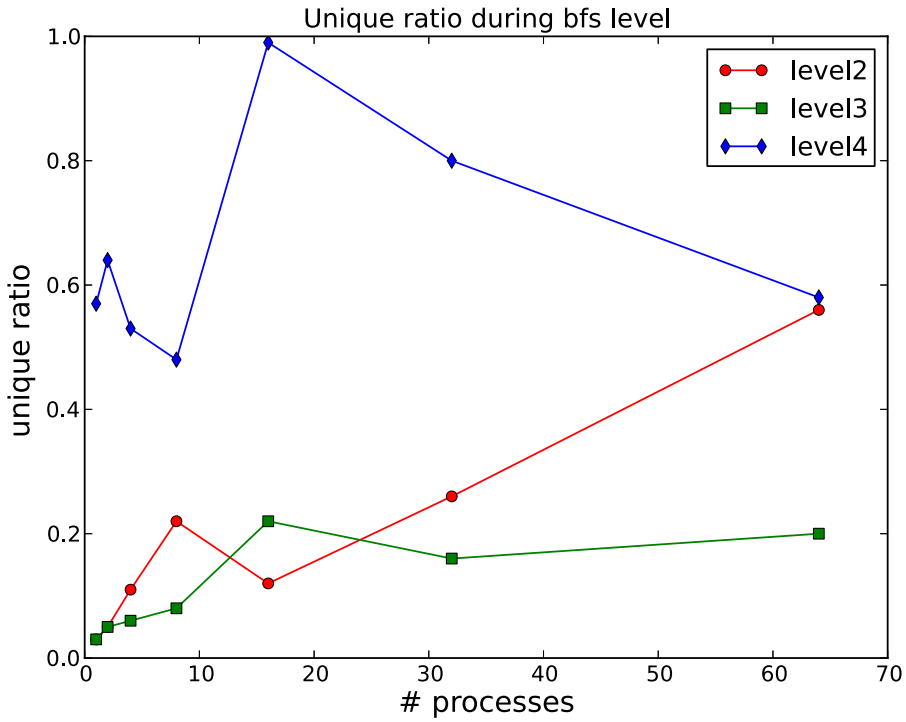


Figure 4.7. Unique ratio for the most expensive BFS levels. The unique ratio is the ratio of elements after and before the pruning operation. A small value of the ratio corresponds to a more effective pruning operation. When the number of tasks increases, the ratio increases accordingly because the number of local copies is lower. However, even with 64 tasks the third level has a unique ratio ~ 0.2, which means that 80% of the elements are removed.

4.3.3 Results

In figure (4.8) we compare the sort-unique version with the straightforward implementation and the reference multi-CPU implementation. The sort-unique BFS is up to 5 times faster than the straightforward algorithm. With 64 GPUs we can traverse more than 1 billion edges per second.

In panel C of figure (4.9) we report a time breakdown of the algorithm for 32 tasks. It is apparent that the sum of all CUDA kernels takes most of the

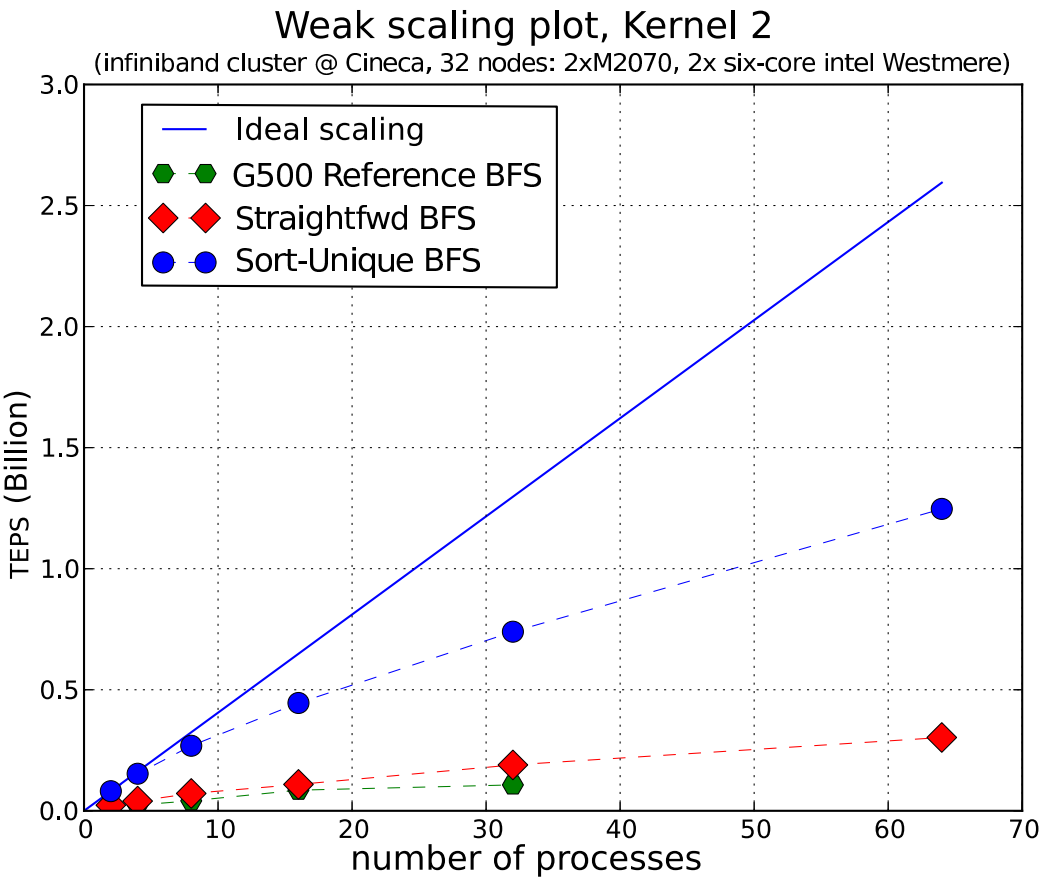


Figure 4.8. Comparison of the different BFS algorithms. The sort-unique algorithm is up to 5 times faster than the straightforward algorithm.

time. Panel B of figure (4.9) shows that computations and communications among tasks are well balanced (we report here the results for 16 tasks because the plot is more clear but the situation does not change for higher numbers of tasks). Panel A of figure (4.9) shows the computation and communication time of one task (*i.e.,* task 0) in a run with 32 GPUs. The pruning procedure (sort and unique) is not only the most consuming part, it actually dominates the running time. The point to point communication is the second most expensive part of the algorithm, whereas the binary search is not as expensive

as one could imagine. Our data to threads mapping has been demonstrated to be very effective and it may be used also in other situations, where there are unbalanced workloads and irregular memory accesses.

BFS LEVEL	1	2	3	4	5	6	7
CLQ	0	386	412287	576282	6046	18	0
NLF	0	2783060	60151972	2201992	6137	18	0
LV	0	52128	364684	39888	166	18	0
RV	772	4480536	22932116	2513692	12336	28	0
NLQ	0	386438+25849	544807+31475	5963+83	18	0+0	0
QR	0	0	0.01	0.26	0.99	1.0	0
UR	0	0.6	0.19	0.58	1.00	1.0	0

CLQ	=	Current Level Queue		
NLF	=	Next Level Frontier		
LV	=	Local Vertices		
RV	=	Received Vertices		
NLQ	=	Next Level Queue =	Enqueued Received +	Enqueued Local
QR	=	Queue Ratio		
UR	=	Unique Ratio		

Table 4.1. Number of elements of the main arrays used in the algorithm. 64 tasks, $SCALE = 2^{27}$

SCALE	N	kernels time	mpi time	NLFS	NLFS-after-SU
21	1	0.68	0.0	37651259	1043789
22	2	0.85	0.1	37906934	1678486
23	4	0.85	0.4	37739872	2688755
24	8	0.85	0.5	58416610	4502903
25	16	0.9	0.6	45334918	5519616
26	32	0.95	0.7	58863642	8703456
27	64	1.01	0.9	42174869	9316248

Table 4.2. The first column reports the size of the graph: $|V| = 2^{SCALE}$; the second column is the number of GPUs; the third and the fourth columns are respectively the sum over all BFS levels of the execution time spent in computation (CUDA kernels) and in communication (MPI primitives); the last two columns are the number of elements in the NLFS before and after the sort-unique operation. The number of elements in the NLFS refers only to the third level of the BFS (which is the most time consuming).

Table 4.1 shows the number of elements of some working arrays at each

iteration of the BFS for a run with 64 tasks. At level 3 both computation and communication become very expensive because the number of elements reaches its peak value. Table 4.2 shows the timings of the computational and communication parts of the algorithm. They refer, respectively, to the sum over all BFS levels of the execution time of all CUDA kernels and all MPI communications. It is apparent that the time spent in computation is almost constant when the number of GPUs increases, whereas the time spent in communication increases. The reason is that the size of the subgraph assigned to each GPU is approximately constant ($|V| \simeq 2^{21}$ where the equal holds for the run with 1 GPU only). Then, to increase the size of the whole graph more GPUs are added. It is clear that the computation on each GPU remains almost constant. On the other end by increasing the number of GPUs the number of exchanged messages increases accordingly. The small growth in the computational time is due to an increase in the number of element received that have to be enqueued.

Figure 4.10 shows the performance of our algorithm when the average degree (edgefactor) of the input graph is increased. Clearly the pruning operation is more effective when the value of the edgefactor is higher and the performance increases accordingly. Figure 4.11 shows the weak scaling plot of the code for a Random (Erdös and Rény) input graph. Random graphs have a degree following a Poisson distribution and a more regular structure. The algorithm shows good scaling properties also on this kind of graph.

4.3.4 Graph 500 benchmark

We ran our code on the "Todi" Cray XK6 cluster of the Swiss Center for Scientific Computing (CSCS), equipped with NVIDIA Tesla X2090 connected by Infiniband QDR and submitted our results to the Graph 500 benchmark.

With 128 GPUs we visited a graph with $SCALE = 2^{28}$ and reached 3 billions TEPS. We entered the Graph 500 ranking at position number 20 (November 2011 ranking).

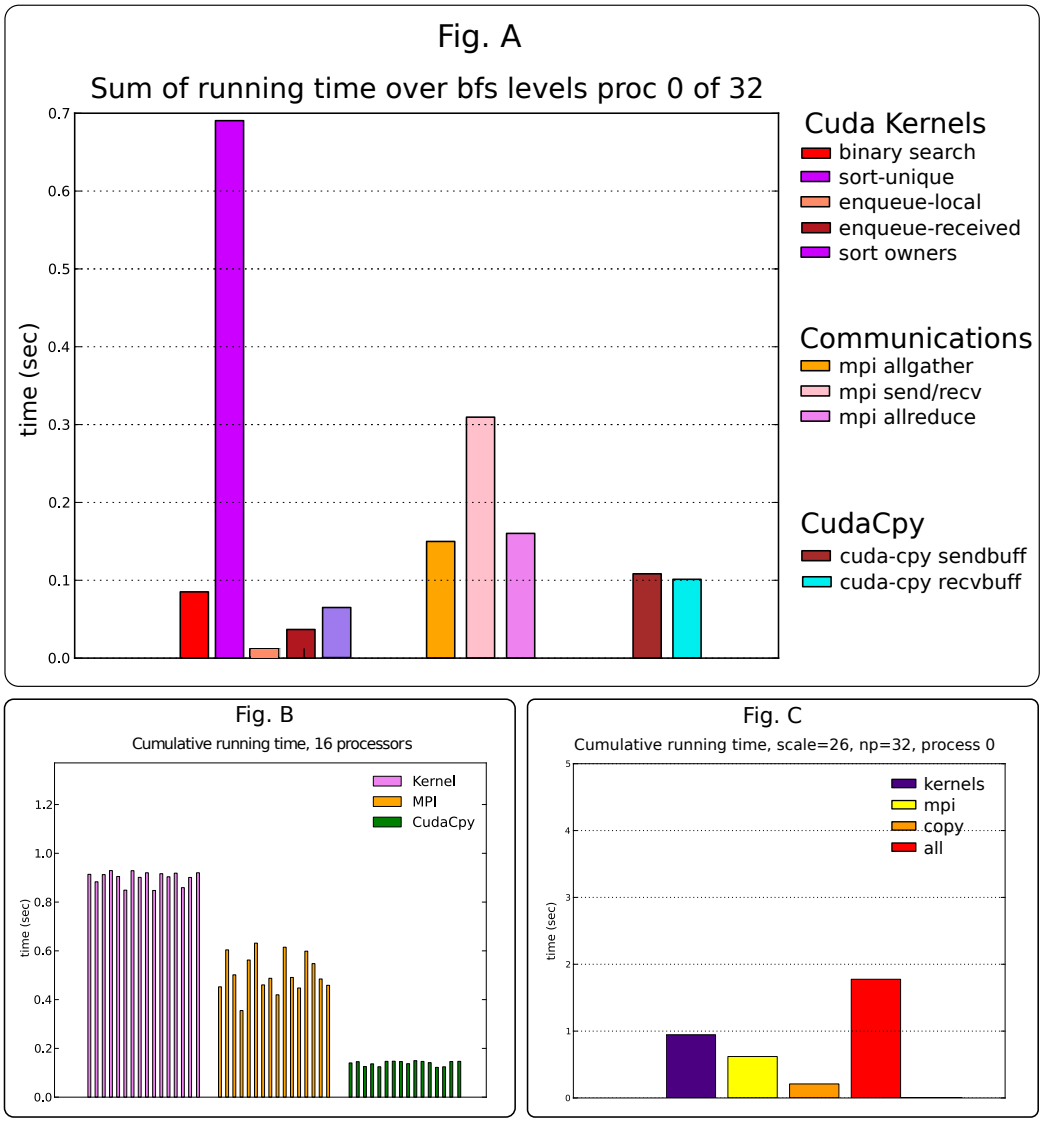


Figure 4.9. Panel A: Running time of various parts of the algorithm, task 0 of 32. The pruning procedure dominates the running time, then it comes the point to point communication part. There are only few levels of the BFS in which the number of elements is very large. Those are the most expensive computational levels. Panel B: Sum of running time of all kernels and all communications over BFS levels. Each bar is a different task. This plot shows that computation and communication among tasks are reasonably balanced. Panel C: Sum of running times of different parts of the algorithm. CUDA kernels execution is the most time consuming part.

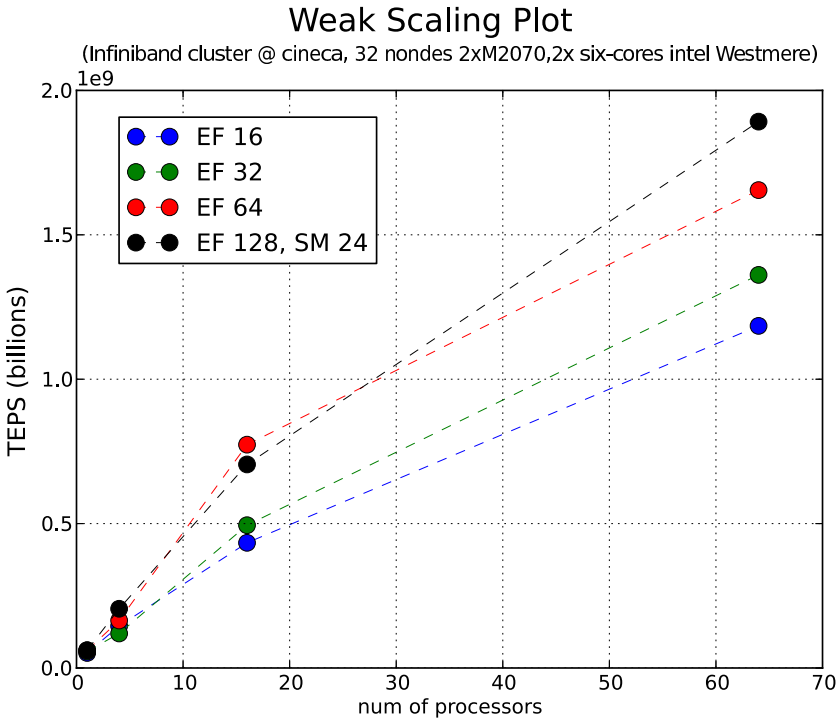


Figure 4.10. The performance of the code is evaluated by varying the average degree (EF) of the input graph. The SCALE of the problem on each GPU is equal to 2^{19} so that with 64 GPUs the total SCALE is 2^{25}. It is apparent that the code performs better for higher value of EF. For EF=128 the maximum scale (SM) reachable with 64 GPU is 2^{24}.

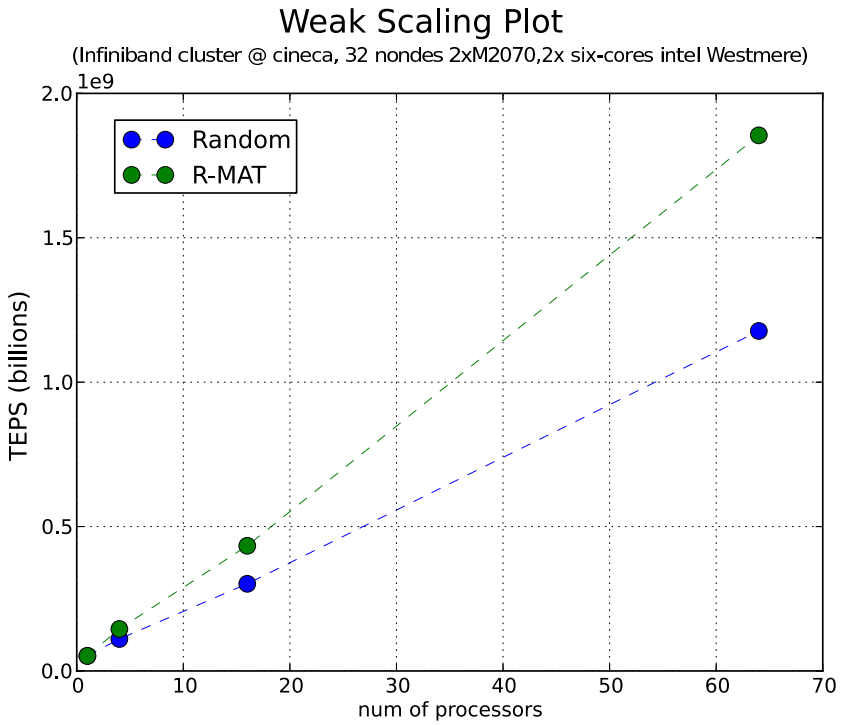


Figure 4.11. Weak scaling plot for a Random (Erdös and Rény) input graph. The SCALE on each GPU is 2^{20}. For comparison we show also the data of an R-MAT graph with the same size.

4.3.5 Comparison among implementations on different architectures

In Tables 3.2 and 3.3 of section 3.4 we reported results obtained on shared memory systems by various authors. In Tables 4.4 and 4.5 we summarize the results of some distributed memory implementations.

As pointed out by Bader *et al.* [13] and by looking at results, it is apparent that shared memory systems perform better than distributed memory systems for (relatively) small size graphs. To achieve more than 3 GTEPS most of the distributed implementations needs hundreds of tasks. Distributed implementations focus on huge size graphs that can be visited only by means of distributed architectures.

As far as we know our implementation is the first on a cluster of GPUs. For this reason we first compare it to the work of Merrill *et al.* [30] and then to some of the implementations on cluster of CPUs.

The work in [30], resorts to a duplicate removal procedure but with a completely different approach, by using a heuristic that removes a high percentage of duplicates at CTA level. In contrast, our algorithm eliminates every duplicate in the Next Level Frontier Set (at a global level).

The last row in Table 3.2 shows the result obtained in [30] with four GPUs when visiting a graph with 16 million vertices having an average degree equal to 16. We recall that the multi-GPUs implementation relies on the VMA technology that supports up to four devices, with a unified memory address space. In comparison our code needs 8 GPUs to visit a graph with 16 million vertices and we achieve a performance of 0.3 GTEPS. However, besides the apparent difference between the two platforms, it is also important to observe the differences in the characteristics of the visited graphs and,

even more important, the differences between the specifications followed by the two implementations. First, the 4 GPUs used in [30] have a unified memory address space with a reduced latency, compared with a standard network interconnection like Infiniband. Moreover, Merrill *et al.,* didn't follow the G500 specifications that impose severe limitations, first of all, the requirement that each vertex of the input graph must be represented as a 64-bit integer. This requirement has a considerable impact when designing an algorithm for a system, like the GPU, where the global memory is a limited resource (see section 4.2.1). As a final remark, in table 4.3 we report the strong scaling of both codes. It is apparent that we have a scaling very similar to that reported by [30] but using an Infiniband interconnection (and visiting a graph with a smaller average degree).

N procs	N Vertices	ef	GTEPS	Sort-Unique Speed-up	Merrill [30] Speed-up
1	2^{21}	16	0.049	1	1
2	2^{21}	16	0.078	1.6	1.5
4	2^{21}	16	0.10	2.1	2.5
8	2^{21}	16	0.126	2.6	
16	2^{21}	16	0.161	3.2	
32	2^{21}	16	0.376	7.6	

Table 4.3. Strong scaling result of our implementation. For comparison we report the speed up presented in [30] when traversing a R-MAT graph with 2 million vertices and 128 million edges (2^{21} vertices and ef= 64) using two and four GPUs.

The main difference between our work and those discussed in section 3.4 and 3.3, is that the size of the graph that can be visited on shared memory systems is limited by the amount of global memory of the system. From [30], it is apparent that the largest graph that can be handled by using 4 GPUs, has 2^{25} vertices and an average degree up to 32. In a distributed algorithm, the number of vertices that can be visited is limited only by the number

of available nodes. By using 128 GPUs, we can traverse a graph with 2^{28} vertices and $32 * 2^{28}$ edges.

Implementations of distributed BFS on clusters of CPU have been tested up to thousands of nodes and, not surprisingly, achieve performances higher than those we can obtain with slightly more than one hundred GPUs. Moreover, the large amount of main memory, available on each node of a CPU cluster and the reduced number of concurrent tasks, allows for the use of different strategies to improve performances on graph algorithms.

In Tables 4.4 and 4.5 we summarize the results of several distributed implementations. In the first table we compare similar results (small number of nodes with a relatively small graph), the second table shows the best performances achieved by various implementations.

By looking at Table 4.4, it is clear how our work can be more easily compared with those of Lv *et al.,* [52, 51], because both follow the Graph 500 specifications and use a relatively small number of nodes. In table 4.6 we compare our performances with those reported in [51, 52]. With 128 GPU we perform better than [52] and our code shows better scaling properties. Compared with [51], our performance is lower but, in that work, Lv *et al.,* used input graphs with a significantly larger size. We also perform better than the Nehalem implementation in [23] and like 102 nodes of the Cray XT6. It is clear that all the CPU implementations visit graphs with a larger size that, almost always, corresponds to a higher result in terms of TEPS.

It is worth to note how, all the distributed implementations on CPU clusters, resort to a bitmap (introduced by [7] see section 3.3) to speed-up memory operations. Unfortunately, that approach, can be hardly used on a GPU because of the limited size of the global memory and the cost of atomic

Authors	Graph Type	Num. of Vertices	ef	GTEPS	Num Processors (CPU/GPU)	Arch. Type	Output
Yoo [79]	Random	Peak	10	0.08	256	IBM BlueGene/L	?
Buluç (1D) [23]	R-MAT	2^{29}	16	2	128 (512 cores)	Cray XT4	distance
Buluç (2D) [23]	R-MAT	2^{29}	16	1	128 (512 cores)	Cray XT4	distance
Buluç (1D) [23]	R-MAT	2^{29}	16	9	1024 (4096 cores)	Cray XT4	distance
Buluç (2D) [23]	R-MAT	2^{29}	16	5	1024 (4096 cores)	Cray XT4	distance
Buluç (2D) [23]	R-MAT	2^{30}	16	3	102 (1224 cores)	Cray XT6	distance
Buluç (2D) [23]	R-MAT	2^{30}	16	8	834 (10008 cores)	Cray XT6	distance
Buluç (2D) [23]	R-MAT	2^{22}	16	0.266	32 (128)	Intel Nehalem	distance
Buluç (2D) [23]	R-MAT	2^{22}	16	0.35	64 (256)	Intel Nehalem	distance
Buluç (2D) [23]	R-MAT	2^{24}	16	0.567	32 (128)	Intel Nehalem	distance
Buluç (2D) [23]	R-MAT	2^{24}	16	0.603	64 (256)	Intel Nehalem	distance
Lv (Multicore)[52]	R-MAT	2^{29}	16	0.9	32 (192 cores)	Xeon X5650	parent
Lv (Multicore)[52]	R-MAT	2^{28}	16	1.2	64 (384 cores)	Xeon X5650	parent
Lv (Multicore)[52]	R-MAT	2^{30}	16	1.2	64 (384 cores)	Xeon X5650	parent
Lv (Compression)[51]	R-MAT	2^{29}	16	2.2	64	Xeon X5650	parent
Lv (Compression)[51]	R-MAT	2^{31}	16	6.2	256	Xeon X5650	parent
Lv (Compression)[51]	R-MAT	2^{32}	16	12	1024	Xeon X5650	parent
Ueno (2D) [72]	R-MAT	2^{34}	16	4	128 (768 cores)	Xeon X5670 (Tsubame)	parent
Ueno (2D) [72]	R-MAT	2^{35}	16	7.2	256 (1536 cores)	Xeon X5670 (Tsubame)	parent
Sort-Unique	R-MAT	2^{26}	16	0.74	32	Tesla 2070	parent
Sort-Unique	R-MAT	2^{27}	16	1.24	64	Tesla 2070	parent
Sort-Unique	R-MAT	2^{27}	16	1.8	64	Tesla 2090	parent
Sort-Unique	R-MAT	2^{28}	16	3.0	128	Tesla 2090	parent

Table 4.4. Comparison of different implementations of distributed BFS. We report results obtained with a similar number of CPU/GPU and a comparable size of the input graph. The implementation in [23] follows many of the Graph 500 specifications, [52, 51] and our (Sort-Unique) implementation follows strictly the Graph 500 rules. We highlight that the number of processors is the total number of CPU/GPU so a node with 2 CPU or 2 GPU counts as 2 processors. (The results for [79] are reported in [7]). Some of the reported values are approximations that we have extrapolated from the plots included in the papers.

operations that would be required to maintain the bitmap coherency.

Most works also implement 2D decomposition on the sparse matrix that represents the graph in a SpMV approach [23, 73]. This approach reduces the number of tasks involved in the communication. As shown in [23] the 2D approach is computationally expensive and is not always convenient. However, it becomes very effective when the scale of the problem is large enough and the communication involves thousands of processors.

Muntes et al. [56] devised a new partitioning scheme that increases the probability of finding neighbors in the same node. This technique seems to be very effective even when the number of nodes is small. A major drawback is that the new partitioning may generate disconnected subgraphs in a

Authors	Graph Type	Num. of Vertices	ef	GTEPS	Num Processors (CPU/GPU)	Arch. Type	Output
Yoo [79]	Random	Peak	200	0.73	256	IBM BlueGene/L	?
Buluç [23]	R-MAT	2^{32}	16	17.8	3300 (40000 cores)	Cray XT6	distance
Lv [51]	R-MAT	2^{32}	16	12.1	1024 (6144 cores)	Xeon X5650	parent
Ueno [72]	R-MAT	2^{36}	16	22.8	1024 (6144 cores)	Xeon X5670 (Tsubame 2.0)	parent
Ueno [73]	R-MAT	2^{36}	16	103	1366 (16362 cores)	Xeon X5670 (Tsubame 2.0)	parent

Table 4.5. Comparison of different implementations of distributed BFS. Here we show the maximum performances reached by each implementation we analyzed.

Sort-Unique			Lv [52]			Lv [51]			Buluç [23]		
N procs	N Vertices	GTEPS	N procs	N Vertices	GTEPS	N procs	N Vertices	GTEPS	N procs	N Vertices	GTEPS
4	2^{23}	0.15	4	2^{24}	0.3				128	2^{24}	0.567
16	2^{25}	0.45	16	2^{26}	0.5				256	2^{24}	0.603
64	2^{27}	1.3	64	2^{28}	1.2	64	2^{29}	2.2	128	2^{29}	2
128	2^{28}	3.0	256	2^{30}	2.2	256	2^{31}	6.2	102	2^{30}	3
			1024	2^{32}	5.6	1024	2^{33}	12	834	2^{30}	8.5

Table 4.6. Direct comparison of our results with three different distributed implementations.

partition.

4.4 APEnet interconnection

In this section, we present an extension of our work by using a different *custom* interconnection technology (APEnet+) that allows for *direct* data exchange among GPUs (with no intervention of the hosting CPU as in the Infiniband case, see Section 2.5). Although APEnet+ is still in a development and testing stage, the reported adaptations of the original algorithm required by the *GPUdirect* technology can be of interest since the results, albeit preliminary, show a clear advantage with respect to Infiniband.

4.4.1 APEnet

APEnet is a 3D Torus interconnection technology proposed in its first version [9] back in 2004 and which is now being developed in its second generation version, called APEnet+ [10]. It has a direct network design which com-

bines the two traditional components, the Network Interface (NI) and the Router (RTR). The Router implements a dimension ordered static routing algorithm and directly controls an 8 ports switch with 6 ports connecting the external torus link blocks (X^+, X^-, Y^+, Y^-, Z^+, Z^-) and 2 local packet injection/extraction ports. The APEnet+ Network Interface comprises the PCIe X8 Gen2 link to the host system, for a maximum data transfer rate of 4+4 GB/s, the packet injection logic (TX) with a 32KB transmission buffer, and the RX RDMA logic which converts the destination virtual memory address in a scatter list to physical (bus) memory. The virtual-to-physical address mapping is currently implemented in software on a microcontroller co-located in the FPGA (Altera NIOS2) that equips the board.

APEnet+ HW architecture is designed around a simple Remote Direct Memory Access (RDMA) programming model. The model has been extended with the ability to read and write the GPU global device memory, directly over the PCIe bus, by exploiting the Nvidia GPUdirect Peer-to-Peer (P2P) HW protocol. The P2P HW protocol is natively supported by both current Nvidia Fermi and new Kepler class GPUs, and is used to implement the inter-GPU memory access features available since the release of CUDA 4.0 for GPU plugged to the same PCI-e switch complex.

In APEnet+ the GPU related P2P features are exposed by minimally extending the APEnet+ programming model: GPU memory buffers can be posted as APEnet+ RX buffers and remotely accessed by using their virtual memory start address. GPU memory buffers can also be used as TX data buffers, and in that case they are automatically mapped into the APEnet+ virtual-to-physical translation table.

When a GPU memory area is used as either a target or source buffer in a transmission, the mapping information of that area are firstly retrieved from

Table 4.7. Traversed Edges Per Second, Strong Scaling, $|V| = 2^{20}$

NP	INFINIBAND	APENET
1	6.25389e+07	6.24038e+07
2	7.8924e+07	1.01101e+08
4	8.20081e+07	1.26543e+08

Table 4.8. Traversed Edges Per Second, Weak Scaling, $|V| = 2^{SCALE}$

NP	SCALE	INFINIBAND	APENET
1	19	5.60594e+07	5.9808e+07
2	20	7.8924e+07	1.01101e+08
4	21	1.08637e+08	1.46482e+08

the GPU device driver and subsequently used to manipulate the GPU in such a way that the buffer can be accessed directly on the PCIe bus.

APEnet+ is being actively improved in both HW and SW, so performance is expected to increase in the next few months. Nonetheless, *direct* GPU memory alignment access constraints, i.e. address and burst length multiple of 32 bytes, are probably going to stay for the near feature until more sophisticated HW blocks are introduced in APEnet+.

4.4.2 Implementation on APEnet

We recall that, usually, the communication among GPUs requires a passage through the hosting CPU [21]. Since the APEnet hardware allows for a direct communication between two GPUs, we modified, accordingly, all the point-to-point communications to use the RDMA features of APEnet. However, to that purpose, we had, as a preliminary step, to align data in order to meet APEnet hardware requirements.

As already stated, we use, as a performance metrics, the number of Traversed Edges Per Seconds (TEPS), so that higher numbers correspond to better performances. Our preliminary results are summarized in table 4.7 and table 4.8. Table 4.7 shows the strong scaling (the size of the graph is fixed) obtained for a graph having 2^{20} vertices and compares the results obtained by using the same GPUs connected by either Infiniband or APEnet. It is apparent that APEnet performs better than Infiniband with an advantage that increases when more GPUs are in use (due to limited availability of APEnet cards we could not perform tests with more than 4 nodes at the present time, but new cards should be available in a short time). Although all CUDA kernels and the rest of the code are identical in the MPI-Infiniband and APEnet version of the code, we wanted to double-check that difference in performances is actually due to the communication part. To that purpose we carried out a detailed measure of the time required by the different part of the code and report the resulting breakdown in Figure 4.12. It is apparent that the communication time is significantly lower with APEnet. Moreover, in the Infiniband version, part of the time is also spent in cudaMemcpy operations to move data back and forth between GPU and CPU. Those memory copy operations are not present in the APEnet version since GPUs exchange data directly. The sum of these two effects explains the difference in the BFS execution time between Infiniband and APEnet and is consistent with the reported number of TEPS. Finally, on table 4.8 we report the results for an experiment with a graph size that increases with the number of GPUs (weak scaling). In figure 4.13 are shown the size of the messages that are sent and received during the execution of a BFS with four processes for a graph with 2^{20} vertices. During the third and the fourth step of the BFS most of the vertices in the graph are visited and the number of elements that have to be

sent is maximized.

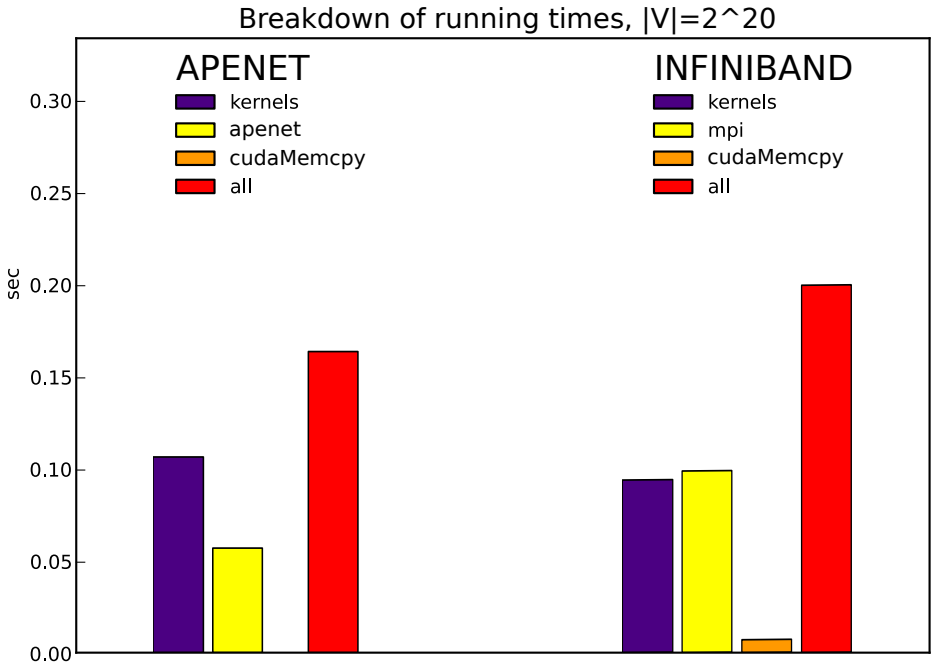
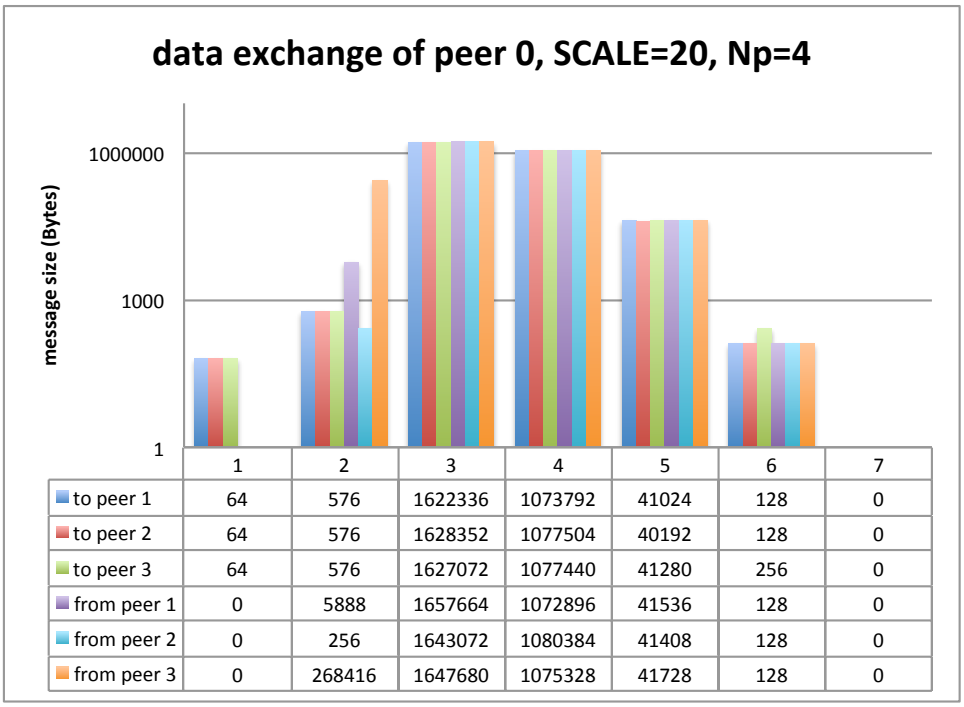


Figure 4.12. Breakdown of the execution time on one out of four tasks for both APEnet and Infiniband.



	1	2	3	4	5	6	7
to peer 1	64	576	1622336	1073792	41024	128	0
to peer 2	64	576	1628352	1077504	40192	128	0
to peer 3	64	576	1627072	1077440	41280	256	0
from peer 1	0	5888	1657664	1072896	41536	128	0
from peer 2	0	256	1643072	1080384	41408	128	0
from peer 3	0	268416	1647680	1075328	41728	128	0

Figure 4.13. Size of messages at each BFS iteration for processor 0 of 4. The $SCALE$ of the input graph is 20. At level 3 the number of visited vertices increases dramatically and the size of messages reaches its peak value.

Chapter 5

Conclusions and future works

5.1 Summary

We studied the problem of developing an efficient BFS algorithm to explore large graphs having billions of nodes and edges. The size of the problem requires a parallel computing architecture. We proposed a new algorithm that performs a distributed BFS and the corresponding implementation on multi-GPUs clusters. As far as we know, this is the first attempt to implement a distributed graph algorithm on that platform.

Our study shows how most straightforward BFS implementations present significant computation and communication overheads. The main reason is that, at each iteration, the number of processed edges is greater than the number actually needed to determine the parent or the distance array (the standard output of the BFS): there is always redundant information at each step. Reducing as much as possible this redundancy is essential in order to improve performances by minimizing the communication overhead.

To this purpose, our algorithm performs, at each BFS level, a pruning

procedure on the set of nodes that will be visited (NLFS). This step reduces both the amount of work required to enqueue new vertices and the size of messages exchanged among different tasks.

To implement this pruning procedure efficiently is not trivial: none of the earlier works on GPU tackled that problem directly. The main issue being how to employ a sufficient large number of threads and balance their workload, to fully exploit the GPU computing power. To that purpose, we developed a new mapping of data elements to CUDA threads that uses a binary search function at its core. This mapping permits to process the entire Next Level Frontier Set by mapping each element of the set to one CUDA thread (perfect load-balancing) so the available parallelism is exploited at its best. This mapping allows for an efficient filling of a global array that, for each BFS level, contains all the neighbors of the vertices in the queue as required by the pruning procedure (based on sort and unique operations) of the array. This mapping is a substantial contribution of our work: it is quite simple and general and can be used in different contexts. We wish to highlight that it is this operation (and not the sorting) that makes possible to exploit at its best the computing power of the GPU.

To speed up the sort and unique operations we rely on very efficient implementations (like the radix sort) available in the CUDA Thrust library.

We have shown that our algorithm has good scaling properties and, with 128 GPUs, it can traverse 3 billion edges per second (3 GTEPS for an input graph with 2^{28} vertices). By comparing our results with those obtained on different architectures we have shown that our implementation is better or comparable to state-of-the-art implementations.

Among the operations that are performed during the BFS, the pruning of the NLFS is the most expensive in terms of execution time. Moreover, the

overall computational time is greater then the time spent in communications. 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. For instance, with 4 GPUs the ratio is 2.125 whereas by using 64 GPUs the value is 1.12. The result can be explained as follows. In order to process the largest possible graph, the memory of each GPU is fully used and thus the subgraph assigned to each processor has a maximum (fixed) size. When the graph size increases we use more GPUs and the number of messages exchanged among nodes increases accordingly. To maintain a good scalability using thousands GPUs we need to further improve the communication mechanism that is, in the present implementation, quite simple. To this purpose, many studies employed a 2D partitioning of the graph to reduce the number of processors involved in communication. Such partitioning could be, in principle, implemented in our code and it will be the subject of a future work.

The current version of our algorithm has been implemented as a set of CUDA kernels and the CPUs are used as a communication co-processors of the GPUs. This choice has been made, first of all, to gain a better understanding of the effectiveness of GPUs on these specific problems and secondly to demonstrate the possibility to implement a GPU version of the Graph 500 benchmark. The benchmark is composed by two parts: the first is responsible of the construction of the data structure used to represent the graph whereas the second one performs the BFS visit. To be compliant with the benchmark, we have also developed the data structure generation for GPU. In order to maximize the number of concurrent operations, we applied the same idea we have used to implement the BFS. We employed operations like prefix-sum and sort to rearrange data and process it with the maximum

number of available threads. On this part, we reported a speed-up of two order of magnitude compared to the reference multi-CPUs implementation provided by the Graph 500 committee.

Finally, we presented our results on a different *custom* interconnection technology (APEnet+) that allows for *direct* data exchange among GPUs. In general, GPUs that reside on different nodes in a cluster, cannot exchange data directly. Data must be transferred through the hosting CPU before and after any MPI call. This aspect represents an issue for both the efficiency of the code and the simplicity of programming. Although APEnet+ is still in a development and testing stage, the reported adaptations of the original algorithm required by the *GPUdirect* technology are of interest since the results, albeit preliminary, show a clear advantage with respect to a state-of-art interconnection technology like Infiniband QDR. The direct GPU communication employed by the APEnet+ architecture allows for more efficient communications. This results in a TEPS increase of $\sim 35\%$ compared to an identical execution in which APEnet+ is substituted by the Infiniband connection.

The definition of a standard to perform direct communications among GPUs is currently a hot topic in research and APEnet+ represents the first real working solution.

5.2 Future work

Effective overlap of computation and communication is a well known technique for latency hiding and can yield significant performance gains for applications. In section 2.4, we described the CUDA streams, a feature that enables the overlap of communication with computation on GPU.

In the next future we expect to exploit CUDA streams in our code. At

present, the most time-expensive operation, the sort-unique, dominates the running time of our implementation and cannot be overlapped with communication. Actually, our algorithm uses the sort-unique to reduce the size of messages exchanged, thus this operation must be performed before starting the point-to-point communication. On the other end, the time required by the remaining computations is small compared to the time required by communication. For this reason, the use of streams in our code can be advantageous only if we overlap data transfers with the sort-unique operation.

The key idea is to partition the NLFS into equal sized blocks and apply the sort-unique operation in a pipelined fashion on consecutive blocks so that, the processing of the i^{th} block can be overlapped with the exchange of data pertaining to the $(i-1)^{th}$ block. Two different CUDA streams can be used to process respectively the i^{th} and the $(i-1)^{th}$ block.

Actually, the aforementioned technique should be applied to the queue instead of the NLFS. The main reason is that, by dividing the queue, we indirectly decrease the number of elements in the NLFS and consequently, the size of the memory reserved to store the NLFS array can be reduced.

This strategy has two major advantages compared to the original algorithm. It clearly overlaps computation and communication and, in principle, allows for the visit of a larger graph. Actually, the maximum size of the graph that the algorithm can visit is limited by the size of the array used to store the NLFS, that is the largest array in our implementation.

This approach, however, is not free from drawbacks. The most important is that by iterating the sort-unique operation on NLFS subsets built from the queue-blocks, the resulting pruning will be only partial. It is not possible to guarantee that all the duplicates will be removed.

5.3 Space-time trade off

Along with the implementation of streams discussed above, we are devising a different strategy to avoid redundant information.

The key idea is to replicate the data structure that represents the graph. In the replicated structure, each vertex will be assigned a new label so that the labels form a contiguous set of integers, from 0 up to the total number of elements in the data structure, say m.

If all vertices, both those owned by the processor and those that are in the adjacency lists, are represented by a contiguous set of integers, it is possible to use a bitmask of size $O(m)$ to keep track of visited vertices over all BFS levels. By recording the state of all vertices stored on the processor we can filter the NLFS from duplicates by a simple look-up in the bitmask, thus removing the sort-unique operation.

This approach recalls the well known situation of space-time trade off. However, the size of the data structure is less than the size of the array that we use to store the NLFS in the original algorithm. Thus, we should be able to gain both in space and in time.

Bibliography

[1] 9 DIMACS challenge (http://www.dis.uniroma1.it/challenge9/index.shtml).

[2] Graph 500 benchmark (www.graph500.org).

[3] Mvapich2 (http://mvapich.cse.ohio-state.edu/overview/mvapich2/).

[4] Openmpi (http://www.open-mpi.org/).

[5] SSCA#2 synthetic benchmark (http://www.graphanalysis.org/benchmark/).

[6] Top 500 lists (www.top500.org).

[7] 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.

[8] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Modern Phys.*, 74(1):47–97, 2002.

[9] R. Ammendola, M. Guagnelli, G. Mazza, F. Palombi, R. Petronzio, D. Rossetti, A. Salamon, and P. Vicini. Apenet: Lqcd clusters a la ape. *Nuclear Physics B - Proceedings Supplements*, 140(0):826 – 828, 2005.

LATTICE 2004 - Proceedings of the XXIInd International Symposium on Lattice Field Theory.

[10] Roberto Ammendola, Andrea Biagioni, Ottorino Frezza, Francesca Lo Cicero, Alessandro Lonardo, Pier Paolucci, Roberto Petronzio, Davide Rossetti, Andrea Salamon, Gaetano Salina, Francesco Simula, Nazario Tantalo, Laura Tosoratto, and Piero Vicini. Apenet+: a 3d toroidal network enabling petaflops scale lattice qcd simulations on commodity clusters. 12 2010.

[11] D.A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, W. Mann, and Theresa Meuse. Hpcs scalable synthetic compact applications #2 graph analysis. 2007.

[12] David A. Bader and Kamesh Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. *HIPC 2005*, 3769:465–476, 2005.

[13] 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.

[14] 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.

[15] Albert-Laszlo Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means.* Plume, reissue edition, 2003.

[16] Albert-László Barabási. Scale-free networks: a decade and beyond. *Science*, 325(5939):412–413, 2009.

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

[18] Albert-László Barabási, Zoltán N. Oltvai, and Stefan Wuchty. Characteristics of biological networks. In *Complex networks*, volume 650 of *Lecture Notes in Phys.*, pages 443–457. Springer, Berlin, 2004.

[19] J. Barnat, P. Bauch, L. Brim, and M. Ceska. Computing strongly connected components in parallel on cuda. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 544 –555, may 2011.

[20] Scott Beamer, Krste Asanovi, and David A. Patterson. Searching for parent instead of fighting over children: A breadth-first search implementation for graph500. Technical report, Electrical Engineering and Computer Sciences University of California at Berkley, 2011.

[21] M. Bernaschi, M. Bisson, M. Fatica, and E. Phillips. An introduction to multi-gpu programming for physicists. *The European Physical Journal - Special Topics*, 210(1):17–31, 2012.

[22] Guy Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.

[23] 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.

[24] Aydn Buluc. *Linear Algebraic Primitives for Parallel Computing on Large Graphs*. PhD thesis, 2010.

[25] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. *Computer Science Department. Paper 541. (http://repository.cmu.edu/compsci/541)*, 2004.

[26] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008.

[27] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2nd edition, 2001.

[28] Frank Dehne and Kumanan Yogaratnam. Exploring the limits of gpus with parallel graph algorithms. *CoRR*, abs/1002.4482, 2010.

[29] Yangdong S. Deng, Bo D. Wang, and Shuai Mu. Taming irregular EDA applications on GPUs. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 539–546, New York, NY, USA, 2009. ACM.

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

[31] Rick Durrett. *Random Graph Dynamics (Cambridge Series in Statistical and Probabilistic Mathematics)*. Cambridge University Press, New York, NY, USA, 2006.

[32] P. Erdős and A Rényi. *On the evolution of random graph*, pages 17–61. Publication of the Mathematical Institute of the Hungarian Academy of Sciences, 1960.

[33] David A. Bader et al. Stinger: Spatio-temporal interaction networks and graphs STING extensible representation. Technical report, 2009.

[34] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29(4):251–262, August 1999.

[35] Gilbert. Random graph. *The Annals of Mathematical Statistics The Annals of Mathematical Statistics The Annals of Mathematical Statistics*, 30(4):1141–1144, December 1959.

[36] Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. *SIGPLAN Not.*, 40:423–437, October 2005.

[37] 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.

[38] K.A. Hawick, A. Leist, and D.P. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Computing*, 36(12):655 – 678, 2010.

[39] David R. Helman and Joseph JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Selected papers from the In-*

ternational Workshop on Algorithm Engineering and Experimentation, ALENEX '99, pages 37–56, London, UK, 1999. Springer-Verlag.

[40] Jared Hoberock and Nathan Bell. Thrust cuda library (http://thrust.github.com/).

[41] S. Hong, S.K. Kim, T. Oguntebi, 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.

[42] 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.

[43] H. Jeong, B. Tombor, R. Albert, Z.N. Oltvai, and A.L. Barabasi. The large-scale organization of metabolic networks. *Nature*, 407(6804):651–654, 2000.

[44] Swapnil D. Joshi and V. S. Inamdar. Performance improvement in large graph algorithms on gpu using cuda: An overview. *International Journal of Computer Applications*, 10(10):10–14, November 2010. Published By Foundation of Computer Science.

[45] 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.

[46] Jure Leskovec. *Dynamics of Large Networks*. PhD thesis.

[47] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, March 2010.

[48] Jure Leskovec and Eric Horvitz. Planetary-scale views on an instant-messaging network. 2008.

[49] Laszlo Lovasz. Very large graphs. *arXiv:0902.0132*.

[50] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[51] 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.

[52] Huiwei Lv, Guangming Tan, Mingyu Chen, and Ninghui Sun. Understanding parallelism in graph traversal on multi-core clusters. *Computer Science - Research and Development*, pages 1–9, 2012.

[53] Mohammad Mahdian and Ying Xu. Stochastic kronecker graphs. In *Algorithms and Models for the Web-Graph*, volume 4863 of *Lecture Notes in Computer Science*, pages 179–186. Springer Berlin / Heidelberg, 2007.

[54] Duane Merrill. *Allocation Oriented Algorithm Design with Application to GPU Computing*. PhD thesis, School of Engineering and Applied Science at the University of Virginia, 2011.

[55] Ulrich Meyer. External memory bfs on undirected graphs with bounded degree. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, pages 87–88, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[56] Victor Muntés-Mulero, Norbert Martínez-Bazán, Josep-Lluís Larriba-Pey, Esther Pacitti, and Patrick Valduriez. Graph partitioning strategies for efficient bfs in shared-nothing parallel systems. In *Proceedings of the 2010 international conference on Web-age information management*, WAIM'10, pages 13–24, Berlin, Heidelberg, 2010. Springer-Verlag.

[57] Richard C. Murphy and Peter M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Transactions on Computers*, 56(7):937–945, 2007.

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

[59] Mark Newman. Random graphs as models of networks. *arXiv:cond-mat/0202208*.

[60] M.E. Newman and D.J. Watts. Scaling and percolation in the small-world network model. *Phys Rev E Stat Phys Plasmas Fluids Relat Interdiscip Topics*, 60(6 Pt B):7332–7342, 1999.

[61] M.E.J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[62] M.E.J. Newman, S.H. Strogatz, and D.J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64(2):026118, 2001.

[63] Nvidia. *NVIDIA CUDA C programming guide 4.0.*

[64] G Palla, L Lovasz, and T Vicsek. Proceedings of the national academy of sciences; multifractal network generator. 107(17):7640–7645, 2010.

[65] S. Redner. How popular is your paper? an empirical study of the citation distribution. *The European Physical Journal B - Condensed Matter and Complex Systems*, 4:131–134, 1998. 10.1007/s100510050359.

[66] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67(6):305 – 309, 1998.

[67] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. *Parallel and Distributed Processing Symposium, International*, 0:1–10, 2009.

[68] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Efficient breadth-first search on the cell/be processor. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1381–1395, October 2008.

[69] N. Schwartz, R. Cohen, D. ben Avraham, A.-L. Barabási, and S. Havlin. Percolation in directed scale-free networks. *Phys. Rev. E (3)*, 66(1):015104, 4, 2002.

[70] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens.

[71] Sengupta Shubhabrata, Mark Harris, Zhang Yao, and Owens John D. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[72] K. Ueno and T. Suzumura. 2d partitioning based graph search for the graph500 benchmark. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1925 –1931, may 2012.

[73] 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.

[74] Andreas Wagner. The yeast protein interaction network evolves rapidly and contains few redundant duplicate genes. *Molecular Biology and Evolution*, 18(7):1283–1292, 2001.

[75] Duncan Watts. The "new" science of networks. *Annual Review of Sociology*, 30(1):243–270, 2004.

[76] Duncan Watts and Steven Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):440–442, 1998.

[77] Paul M. Weichsel. The kronecker product of graphs. *Proceedings of the American Mathematical Society*, 13(1):pp. 47–52, 1962.

[78] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. 1999.

[79] 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 *In SC 05: Proceed-*

ings of the 2005 ACM/IEEE conference on Supercomputing, page 25. IEEE Computer Society, 2005.