Task-based Parallel Breadth-First Search in Heterogeneous Environments

Lluís-Miquel Munguía*† David A. Bader† Eduard Ayguade‡
*Barcelona School of Informatics, Universitat Politècnica de Catalunya, Barcelona, Spain
†College of Computing, Georgia Institute of Technology, Atlanta GA 30332
‡ Barcelona Supercomputing Center (BSC), Spain

lluis.munguia@gatech.edu

***Abstract*—Breadth-first search (BFS) is an essential graph traversal strategy widely used in many computing applications. Because of its irregular data access patterns, BFS has become a non-trivial problem hard to parallelize efficiently. In this paper, we introduce a parallelization strategy that allows the load balancing of computation resources as well as the execution of graph traversals in hybrid environments composed of CPUs and GPUs. To achieve that goal, we use a fine-grained task-based parallelization scheme and the OmpSs programming model. We obtain processing rates up to 2.8 billion traversed edges per second with a single GPU and a multi-core processor. Our study shows high processing rates are achievable with hybrid environments despite the GPU communication latency and memory coherence.**

I. Introduction

In recent years, graph algorithm design has gained an important role in science, as many emerging large-scale scientific applications now require working with large graphs in distributed networks. Breadth-First Search (BFS) is of particular relevance because it is widely used as a basis for multiple fields. Other common graph applications also use Breadth-First Search as a fundamental part of their algorithm. Some of the relevant problems include flow network analysis, the shortest-path problem, and other graph traversals, such as the A* algorithm.

Due to their irregular nature, large graph traversals have become a problem that is hard to parallelize efficiently. Such irregularity, where the memory access pattern is not known a priori, and the necessity of large data structures, have made I/O efficiency one of the greatest hurdles to optimal performance. In addition to such inconveniences, traditional performance optimization strategies usually applied to memory-bound applications are no longer effective because of a low data reuse and lack of memory locality.

Disregarding memory optimization strategies, previous graph parallelization efforts have been oriented toward masking the I/O problems with high doses of aggressive parallelism and multi-threading. Cray XMT, IBM Cell/BE, and NVIDIA GPUs are architectures that exploit such advantage and prioritize bandwidth over latency. Work on the mentioned platforms has shown great performance improvements in overcoming the high latencies incurred during graph explorations. The general purpose GPU (GPGPU) architectures have the added value of being an affordable solution while maintaining high throughput and low power consumption levels.

While any of the previously mentioned platforms offers massive parallel processing power, its performance while traversing a graph will ultimately depend on its connectivity properties, the architecture, and the memory subsystem. GPGPU architectures yield unmatched performance if sufficient parallelism is available and the graph fits on the GPU's memory. But they fail to yield the same performance otherwise, due to large overheads and the impossibility of overlapping the communication latencies with effective computation.

In this paper, we propose several hybrid strategies for tackling Breadth-First Search graph traversals that aim to balance such GPU shortcomings with the assistance of multi-core CPUs. Through the use of heterogeneous programming platforms, the deficiencies caused by the limited available parallelism can be overcome and better potential performance can be achieved. We introduce a task-based parallelization strategy that provides several performance benefits and propose two distinct hybrid environment implementations. We then proceed to do a performance evaluation of both implementations, reaching processing rates up to 2.8 billion traversed edges per second with a single GPU and a multi-core processor under a comprehensive set of graph sizes which include graphs that do not fit in the standard GPU main memory.

II. Relevant work

Since the early days of computer development, a distributed approach has been widely accepted as the best methodology to tackle big Breadth-First Search graph traversals. Although prior works comprise BFS implementations on numerous kinds of platform systems, most

978-1-4673-2371-0/12/$31.00 ©2012 IEEE

of them share a common underlying PRAM strategy.

Bader et al. [2] took a fine-grained parallel approach and succeeded in substantially speeding up irregular graph traversals. By avoiding load-balancing queues and relying on the massive multi-threaded architecture of the Cray XMT, memory latencies were overlapped with the computation of thousands of hardware threads. This strategy proved to be effective, as the random locality nature of the BFS prevents the use of memory access optimizations.

GPGPU architectures are based on a philosophy similar to that of the Cray massive multi-threaded systems. Subsequent work on GPU-based BFS algorithms was pioneered by Harish et al. [3], [4] , Merrill et al. [6] and Luo et al. [8] . They suggest the use of GPUs as the next milestone for BFS algorithms.

Harish et al. dismissed the use of queues to store the vertices to be traversed (also known as vertex frontier) due to the huge overhead caused by atomic and synchronous operations on the GPU. Instead, their approach checks every vertex to see if it belongs to the current frontier. Because of this design decision, the time complexity of the algorithm becomes much higher, as all vertices must be checked at every level: Let L be the total number of levels, then the time needed to exhaustively check frontier vertices will be $O(VL)$, making a total of $O(VL + E)$ when taking into account the time to explore every edge. In the worst case scenario, when $L = O(V)$, the complexity would turn into $O(V^2)$; which is less work-efficient than the $O(V)$ CPU implementation. Hussein et al. [5] also described a vertex-oriented BFS implementation using a similar quadratic strategy. Their approach performed BFS solely on the GPU in order to avoid communication overheads when transferring graph data. In most cases, both algorithms will perform more work than the standard sequential algorithm. Hong et al. [9] introduced a vectorized approach to the same vertex-oriented strategy by assigning vertices to an entire group of threads. Their strategy allowed such group of threads to cooperatively explore adjacencies, thus mitigating the load unbalance caused by vertex exploration.

Nevertheless, other parallelization strategies for GPU architectures were also studied. The work of Deng et al. [7] introduced a second approach to GPU applications in BFS algorithms. In this case, matrix-based data representations were used. Each frontier could be transformed into a matrix-vector multiplication. Even though it was a novel approach, the worst case was still slower than the sequential BFS version.

Deeming prior approaches ultimately inefficient, Luo et al. [8] used queues in order to keep workload complexity at the same level as the serial BFS algorithm. Their approach is based on the use of efficient queue structures to store new frontiers and hierarchical kernel arrangements to reduce synchronization overhead, allowing his best implementation to run significantly faster than previous quadratic implementations. Merrill et al. [6] also favored this kind of structure in their latest proposal. Their work comprises a deep and thorough study of several strategies for implementing BFS on GPUs and multi-GPUs. To date, their implementation is the fastest on this kind of architecture.

To our knowledge the work by Hong et al. [10] is the only prior attempt to integrate CPUs and GPUs in a hybrid environment by choosing dynamically between three different implementations for each level of the BFS algorithm.

Previous literature has shed some light on how to design scalable algorithms for data-intensive applications such as BFS. Such studies stress the huge advantage of GPU computing over more traditional approaches as a cost-effective parallel solution.

Other prior work has also focused on edge-oriented partitioning strategies. Yoo et al. [1] proposed a 2D edge partitioning algorithm and implemented a scalable distributed algorithm for the BlueGene supercomputer. On Cray machines, Buluc et al. [11] presented also a two-dimensional sparse matrix partitioning-based approach that was aimed at reducing communication overheads between processing nodes in a distributed memory environment.

Multicore general purpose CPU techniques have been also widely used and studied for graph algorithms. Agarwal et al. [12] presented a thorough study that used Intel Nehalem processors showing performance comparable to that obtained with massive multi-threaded machines such as the Cray XMT and the BlueGene/L. Xia et al. [13] also achieved good results by using a topologically adaptive parallel BFS algorithm on Intel and AMD processors.

In this paper, we decouple the BFS problem from the underlying hardware instead of examining it from the perspective of a single specific architecture. This new approach provides several performance benefits and new ways to look at the problem, which are explained further in the sections that follow.

III. Background

A. The CUDA programming model

CUDA is a proprietary NVIDIA standard for general-purpose parallel programming across NVIDIA-powered GPUs. CUDA provides a low-level hardware abstraction and a programming framework that allows the programmer to control massive multi-threaded GPUs. A CUDA program will contain two kinds of code:

1) Code for the host CPU
2) Highly parallel code for the SPMD GPUs; also known as CUDA kernels.

A normal execution of a CUDA kernel starts by executing host CPU code. The execution is moved to the GPU once a CUDA kernel is invoked, where the programmer can take advantage of the massive parallel architecture. The CUDA runtime creates thousands of threads collectively grouped in blocks and grids. A kernel may be launched synchronously or asynchronously.

Threads within a grid are organized in a two-level hierarchy. A grid is divided into several blocks, while a block is composed of a group of threads. The two levels will have different synchronization properties, as threads within a block can be synchronized together, while it is not feasible to synchronize blocks at the grid level.

B. BFS problem characterization

Given an undirected graph $G(V, E)$ and a source vertex s_0, a breadth-first search will traverse G starting from s_0 exploring all the vertices at a given distance d from s_0 before traversing vertices at a further range. As a result, a spanning tree rooted to s_0 with its connected component from the root can be obtained. The work and time complexity of the search will be $O(V + E)$, since every vertex and edge must be explored given the worst case.

```
Input: Graph G(V,E), source node s0
Output: Distances from s0 Dist[1..|V|]
1   ∀v ∈ V do:
2       dist[v] = ∞
3   dist[s0] := 0
4   Q := ∅
5   Q.enqueue(s0)
6   while Q ≠ ∅ do
7       i := queue.dequeue()
8       for each neighbor v of i do
9           if dist[v] = ∞ then
10              dist[v] := dist[i] + 1
11              Q.enqueue(v)
12          endif
13      endfor
14  endwhile
```

Algorithm 1: Serial BFS algorithm

Algorithm 1 presents a serial algorithm that uses queues to maintain the order of the traversed vertices. It starts by introducing the source vertex s_0 in the empty queue. Until this queue is depleted, the first vertex in it will be extracted and its adjacencies explored. Each of the unvisited adjacencies will be enqueued again.

The data structure used in this paper to describe the undirected and unweighed graph is a Compact Sparse Representation (CSR). CSRs include an array A that contains all the adjacency lists that define a graph. In addition, an extra array N with $|V| + 1$ positions is needed to determine which adjacencies belong to which vertex. Simply put, the elements stored in N are pointers to A. Such pointers indicate the beginning and the end of the adjacencies starting at the node. As an example, a vertex n will have its adjacencies starting at $N[n]$ and finishing at $N[n+1]$. Hence the need for $|V|+1$ positions to describe the limits for $|V|$ vertices. As a matter of fact, $N[|V| + 1] = |E|$.

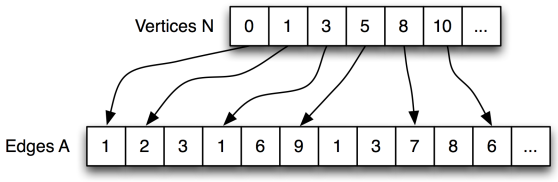

Fig. 1. CSR graph representation

IV. Parallelization strategy

Our approach will follow a 1D level-synchronous parallelization strategy, which is a straightforward parallelization of the loop in line 6 of Algorithm 1. Ideally, the n vertices of the graph will be distributed between the different p processors uniformly.

Given a graph $G(V, E)$, its vertices and outgoing edges can be separated disjunctively, thus allowing no data input overlap between the different processing nodes. Following the CSR representation used, vertices and edges can be grouped together in the way depicted in Figure 2.

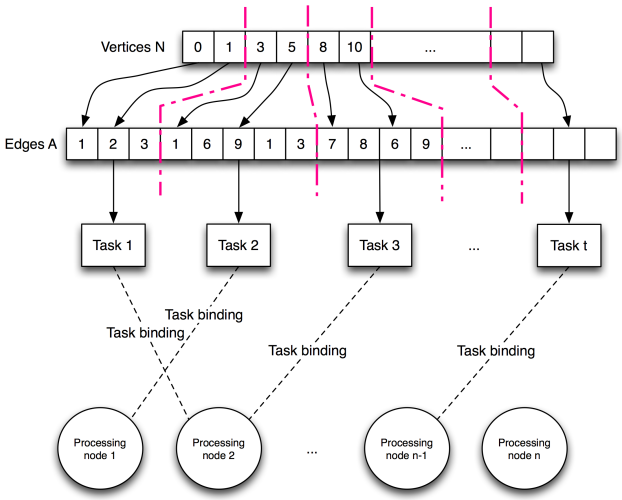

Fig. 2. Graph input division

We define such subdivision as a task. A graph can potentially be divided into an arbitrary number of tasks, independently of the number of parallel processing units available. In our parallelization strategy, each task is also the owner of the distance statuses of the vertices they are assigned. A task becomes a logical division of the graph space. During the execution of each BFS level iteration, all logical tasks are assigned to a processing node and their active vertex frontiers are to be traversed. This binding is also illustrated in Figure 2, in addition

to the graph partitioning. Essentially, the task notion is not different from the subdivision strategies proposed in previous works. However, it is important to note that the number of tasks is arbitrary and could be independent from the number of processors. This subtle difference allows several performance benefits explored further in this paper.

```
Input: Graph G(V,E), source node s_0
Output: Distances from s_0 Dist[1..|V|]
1  ∀v ∈ V do:
2      dist[v] = ∞
3  dist[s_0] := 0
4  VF:= ∅
5  NVF:= ∅
6  taskQueue := taskOwner(s_0)
7  VF.enqueue(s_0, taskQueue)
8  currentFrontier := 0
9  while VF≠ ∅ do
10     for each i in VF parallel do
11         i := queue.dequeue()
12         dist[i] := currentFrontier
13         for each neighbor v of i do
14             if VB[v] = 0 then
15                 VB[v] := 1
16                 currentTask := taskOwner(v)
17                 NVF.enqueue(v, currentTask)
18             endif
19         endfor
20     endParallelfor
21 Synchronization_barrier()
22 currentFrontier := currentFrontier + 1
23 VF:=NVF
24 NVF:= ∅
25 endwhile
```

Algorithm 2: Parallel task BFS algorithm

Algorithm 2 shows a high level description of a task-parallel Breadth-First Search. Unlike the serial version, Algorithm 2 introduces a number of new structures aimed toward maintaining coherency. Because it follows a 1D level-synchronous parallelization strategy, a vertex frontier is needed. A vertex frontier is the subset of vertices that are discovered for the first time in a level iteration i and are to be expanded in the following iteration $i+1$. Every BFS level execution will consume a vertex frontier VF. At the same time, a new vertex frontier NVF will be built and provided for the next iteration. Three methods are given to interact with VF and NVF: $taskOwner$, $enqueue$ and $dequeue$. The first provides the ID of the Task owning the vertex while the other two interact with the vertex frontiers, allowing the introduction and extraction of vertices. In addition, a visited bitmap VB is maintained in order to determine which vertices have already been visited. A synchronization barrier is required to guarantee the level synchronism between the different levels, which is expressed with the variable $currentFrontier$.

Globally, a task must be interpreted as a unit of processing with its disjunctive input and shared output. At a given iteration, each task will explore its active

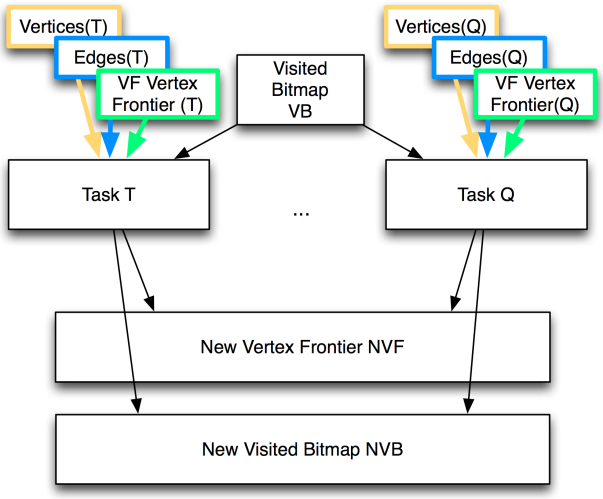

Fig. 3. Task input division

vertex frontier VF. A given task may discover nodes that belong to other tasks. Therefore, the output must be synchronized under shared data structures such as queues. For the sake of clarity, NVF is presented in Algorithm 2 as a unified structure and shared between tasks. Regardless of the implementation methodology, it is important to note its function as a point of communication between tasks. Figure 3 displays the data flow that takes place during the processing of a graph traversal.

The task notion offers a high degree of flexibility. For example, task divisions could be much more numerous than processors. Given such a case, tasks could be assigned dynamically to a processor unit to be executed. This approach leads to several favorable properties.

1) Load Balance: If the task granularity is small enough, tasks could be assigned dynamically to the processing units, thus splitting the workload equally between processor nodes.
2) Spatial locality: Tasks are contiguous subdivisions of the graph. Therefore, two nodes processed consecutively in the same task may have a higher chance of residing closely in memory.
3) Possibility to process tasks on hybrid environments: Since tasks are logical divisions, they could be processed in heterogeneous environments and synchronized thereafter.

V. Strategies for tackling the BFS problem in heterogeneous environments

Using the task division, we propose two graph traversal strategies that take advantage of heterogeneous platforms. CPUs and GPUs offer different execution properties when performing a graph traversal. Depending on the graph topology and the size of the Vertex Frontier for each iteration, one platform will be more suitable and will yield better results than the other. As an example,

a GPU might offer a better performance if enough parallelism is available. However, it will fail to outperform a multi-core CPU in big-diameter graphs with low connectivity. Figure 4 demonstrates this contrast by comparing the levels of performance of the CPU and GPU implementations. It shows the processing times of one CPU/GPU thread traversing a single task with different Vertex Frontier sizes. For this specific graph, the GPU task implementation already outperforms the CPU counterpart when the size of the Vertex Frontier is more than 7000 vertices.

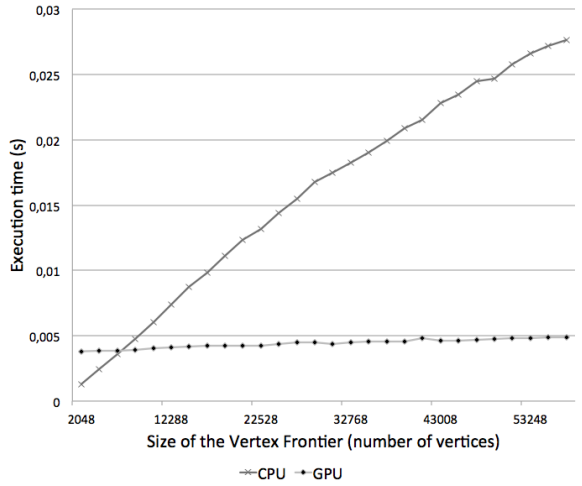

Fig. 4. Single thread task performance comparison under a randomly generated graph with 1M vertices and 16M edges

In this section, we describe several strategies for taking advantage of both CPU and GPU components in a hybrid environment.

A. Workload-aware hybrid BFS

We propose a first scheme that consists of an adaptive algorithm. This algorithm adapts the execution flow to the graph topology by redirecting it to the most appropriate hardware. As an example of graph topology, we show in Figure 5 the evolution of the vertex discovery in a traversal of a randomly generated graph. It can be observed from the chart that the number of discovered vertices varies greatly from level to level. We obtain a performance advantage from this by changing from CPU to GPU or vice-versa depending on the size of the New Vertex Frontier NVF.

In order to determine when to transfer a task between GPU and CPU processing nodes we define a workload limit λ. If, for a given task, the number of vertices to be traversed contained in the Vertex Frontier VF is higher than λ, the task will take advantage of the massive parallelism offered by the GPU. Otherwise, the task will be assigned to a CPU processing node in order to avoid

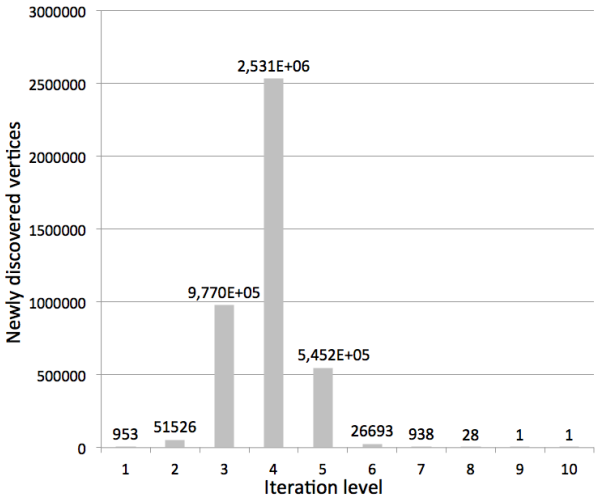

Fig. 5. Traversal evolution for a randomly generated graph with 4M vertices and 32M edges

the unnecessary latency penalties and overhead inherent to the GPU.

Figure 6 illustrates the state machine associated with the adaptive algorithm. Executing the graph traversal in a hybrid environment introduces the problem of memory consistency, as it will require maintaining two disjunct memory spaces when switching the execution flow from CPU to GPU or vice versa. Several optimizations can be used in order to minimize the transfer information and latency, such as heavy caching and graph data replication. For each task, the following information is needed.

1) Graph portion if not already copied: $O(|V|/t)$ for the vertex array and potentially $O(|E|)$ if the task holds the entire edge array.
2) Distance statuses of the vertices owned by the task: $O(|V|/t)$

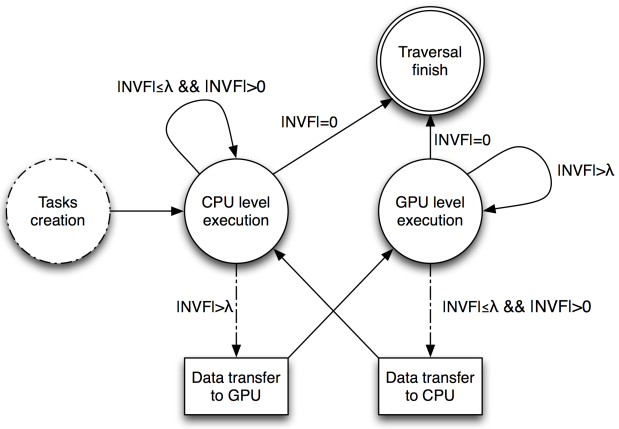

Fig. 6. Workload-aware algorithm state machine

3) Active Vertex Frontier VF: potentially $O(|V|/t)$ if all nodes are explored at once.
4) Visited vertex Bitmap VB portion owned by the task: $O(|V|/t)$.

The λ factor adds yet another parameter for the sake of flexibility. At its extremes, the algorithm can be set to be executed only in the CPU ($\lambda = |V|$) or in the GPU ($\lambda = 0$); leaving a huge range of possibilities in the middle. However, the appropriate λ parameter depends highly on the graph characteristics. Further details of the adaptive algorithm will be shown in the Implementation section.

B. Fixed-partitioned-space hybrid BFS

The previous scheme offered high hardware adaptation to the graph topology requirements. However, it may also carry a high potential transfer and synchronization overhead because of the heterogeneous memory space, and it only takes advantage one of the two components at a time.

A second strategy can be aimed at using both CPU and GPU simultaneously in a synchronized fashion at the expense of also reducing the adaptation flexibility. Instead of dynamically binding tasks to environments, they can be assigned to two static disjunct sets of tasks instead. Let $\{G\}$ be a subset of the task set $\{T\}$ bound to be executed in the GPU, we can define $\{C\} = \{T\} \backslash \{G\}$ as the subset bound to be executed in the CPU. Such subsets will remain unchanged throughout the traversal execution.

The fixed partitioned space hybrid BFS scheme consists of two phases:

1) An execution phase: Both CPUs and GPU traverse the assigned tasks in a parallel fashion. $\{C\}$ and $\{G\}$ are formed by tasks that conserve the parallelism and can be executed independently. As a result of the level traversal, two disjunct New Vertex Frontiers (NVF_C and NVF_G) and two new Visited vertex Bitmaps (VB_C and VB_G) are created.
2) A communication phase: The part of the NVF_C and the NVF_G belonging to tasks bound for execution in the other environment must be transferred. In addition, the Visited Bitmaps generated (VB_C and VB_G) must also be unified and synchronized.

With the fixed partition scheme, only the New Vertex Frontier and the Visited Bitmap need to be exchanged. No synchronization is required for the vertex distance status, as each task is responsible for maintaining the distances of the owned vertices. Figure 7 shows the proposed execution flow and synchronization scheme.

VI. Implementation

Our implementation relies on the OmpSs productivity framework to handle the task subdivision and execution.

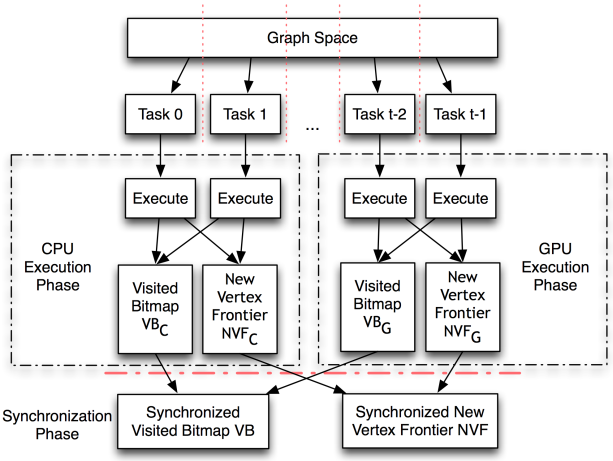

Fig. 7. Fixed-partitioned-space hybrid BFS execution schema

OmpSs is a programming model introduced by Ayguade et al.[14] that stresses the use of effective workload scheduling strategies as a key element for achieving better overall performance in distributed algorithms. Among other features, OmpSs integrates the task notion we have introduced in this paper, as well as task executions in heterogeneous environments like CPUs, GPGPUs and Cell processors.

In this section, we will describe a specific task implementation for both GPU and CPU environments. Since a task is a logical division that should be able to be executed in any component, any task implementation should be compliant with the inputs and outputs described in the previous sections.

A. GPU task implementation

As stated in many prior works, the strategy for implementing BFS on the GPU requires a different point of view from implementations on the CPU. Developing a new algorithm exclusively for the GPU is not the purpose of this paper. In this section, we provide an implementation that combines several elements discussed in previous studies [6]. However, under the task specification, any implementation can be introduced as long as it meets the input/output conditions.

OmpSs features a runtime that handles CUDA kernels expressed in tasks. Despite the fact that tasks cannot be launched concurrently in a GPU, they can be pipelined or executed in multiple GPUs seamlessly if the dependencies expressed in the task annotations allow it.

Algorithm 3 describes the GPU BFS task implementation using OmpSs terminology. OmpSs introduces annotations to delimit task inputs and outputs. These annotations are used afterwards by the runtime to dynamically build a dependence task graph and schedule tasks in a data flow way. Edges are traversed cooperatively

between all the threads in the same warp in order to avoid load imbalance between threads. Until all edges are traversed, threads vie for the control of the warp (lines 9 to 15). When new vertices are discovered, they are accumulated in a vertex mask VM. Once all the tasks bound for that specific GPU are finished, the Vertex Mask contains the newly discovered vertices. Then, it is compacted and converted into the New Vertex Frontier NVF by means of a scan and prefix sum. This step is needed in order to meet the same output condition as the CPU counterpart. Effectively, the Vertex Mask bitmap is converted into the new vertex frontier for the next level iteration.

```
Input: Task Graph partition T(V,E), Active
     Vertex Frontier VF, current frontier cf
Output: Shared Vertex Mask bitmap VM
Input – Output: Shared Visited Bitmap VB,
     Provisional Partial distance vector
     Dist[1..|V|]


#pragma omp target device ( cuda )
#pragma omp task input ( T(V,E), VF, cf) output
     (dist) inout(VM,VB)
1    volatile shared owner[NUMWARPS][3]
2    tid := getThreadId()
3    i :=VF[tid]
4    range :=V[i]
5    range_end :=V[i + 1]
6    if dist[i] = ∞ then
7        dist[i] := cf
8    endif
9    while any(range_end > range) do
10       if range_end > range then
11           owner[warp_id][0] := lane_id
12       endif
13       if owner[warp_id][0] = lane_id then
14           owner[warp_id][1] := range
15           owner[warp_id][2] := range_end
17           range := range_end
16       endif
17       pointer := owner[warp_id][1] + lane_id
18       end := owner[warp_id][2]
19       while pointer < end do
20           v := E[pointer]
21           if VB[v] = 0 then
22               VB[v] := 1
23               VM[v] := 1
24           endif
25       endwhile
26   endwhile
```

Algorithm 3: GPU Task BFS

By using a Shared Vertex Mask, the GPU task implementation allows the accumulation of the newly discovered Vertex Frontier without the overhead caused by the access to shared queues by multiple GPU threads. Figure 8 depicts the execution flow of several tasks assigned to the GPU.

B. CPU task implementation

The proposed task implementation tailored for the CPU strongly resembles the generic serial BFS algorithm. However, it will have a task subdivision as an input instead of an entire graph.

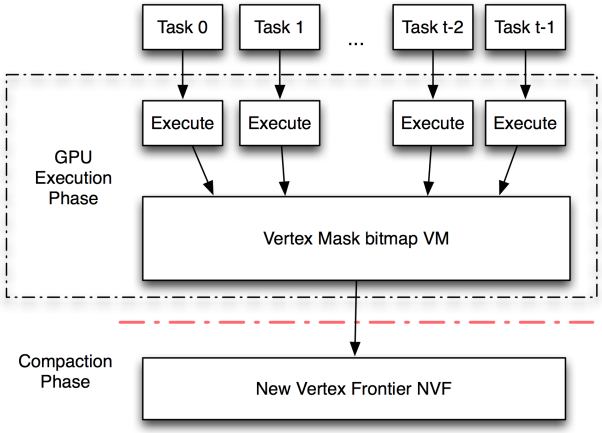

Fig. 8. GPU task implementation

In contrast to the proposed GPU implementation, a given CPU task will be of a serial nature, as the parallelism resides in the independent execution of several tasks on a coarser level.

```
Input: Task Graph partition T(V,E), Active
     Vertex Frontier VF, current frontier cf
Output: Shared Next Vertex Frontier NVF
Input – Output: Shared Visited Bitmap VB,
     Provisional Partial distance vector
     dist[1..|V|]


#pragma omp target device ( smp )
#pragma omp task input ( T(V,E), VF, cf )
     output (dist) shared(NVF,VB)
1  while VF≠ ∅ do
2      i :=VF.dequeue()
3      if dist[i] = ∞ then dist[i] := cf
4      for each neighbor v of i do
5          if VB[v] = 0 then
6              VB[v] := 1
7              currentTask := taskOwner(v)
8              NVF.enqueue(v, currentTask)
9          endif
10     endfor
11 endwhile
```

Algorithm 4: CPU Task BFS

Algorithm 4 shows a description of the CPU BFS task implementation. It takes many elements from the parallel for loop's body from Algorithm 2. As a peculiarity, Algorithm 4 uses a task partition as input, as well as the Active Vertex Frontier VF owned by that task and the current frontier cf. cf determines the actual distance from the source vertex s_0.

VII. Experimental results

We examine our implementations and their performance using a set of graphs generated with the Graph500

Benchmark [15] generator. Graph500 is a set of benchmarks and performance metrics that provide useful information on the suitability of supercomputing systems for data intensive applications. The generator is based on a parallel Kronecker graph generator and is meant to comply with the requirements of the Graph500 Search benchmark. These requirements are aimed at producing reproducible results for a given graph size and seed, regardless of the level or type of parallelism in use. The evaluated graphs cover sizes from 1 million to 8 million vertices and their arity range from 8 to 128. We use a node from the Barcelona Supercomputing Center's MinoTauro cluster to test our implementations. Each node is based on a dual socket system using Intel Xeon E5649 processors. Each provides 6 cores running at 2,53 GHz and 12MB of shared cache memory. In addition, one of the two NVIDIA Tesla M2090 GPUs with 512 CUDA Cores and 6GB of GDDR5 Memory is also used. We use the measure of traversed edges processed per second (TEPS) to gauge the performance of the proposed implementations. The TEPS magnitude can be defined as

$$TEPS = \frac{Edges\,traversed}{Time\,elapsed} \quad (1)$$

Because of its implementation, the Graph500 generator does not output uniform and completely connected graphs. For the experiments, we choose a source vertex s_0 that spans in a relatively comprehensive tree, covering over 90% of the total number of edges in all graphs.

As an introductory note, Figure 9 illustrates the benefits of the task subdivision strategy by showing the evolution of the performance of a graph traversal while increasing the number of tasks. This figure is based on a CPU-only implementation when traversing a graph from the set with 2 million vertices and an arity of 32 (64 million edges). For this particular experiment, only 6 cores (1 socket) were used with 6 threads. We obtain a processing rate from 153 to 790 MEPS, noticing an increasing performance with more available tasks; reaching its peak around 128. Having more tasks available allows a more fine-grained graph subdivision, providing the beneficial side effects described in Section IV. For this particular experiment, the overhead associated with task management starts to be noticeable at 256 tasks, making the performance drop afterwards.

Figure 10.a shows the performance and processing rate of our workload-aware scheme when traversing the benchmark set. All of the experiments were conducted using a variable number of threads as well as a variety of task subdivisions. The results show the performance of the best combination of these variables in each case for graphs between 1M and 8M vertices and an arity from 8 to 128. The lambda value was also tailored specifically for each execution in order to obtain optimal efficiency. As a general rule, the execution was moved to the GPU before the vertex discovery rate became exponential. We can distinguish three different patterns in our results depending on the graph size. For graphs with fewer than 256 million edges, the performance rate increases with the arity and scales well with an increasing number of vertices. It reaches a processing rate of 2.8 TEPS for arities of 128 by benefiting from graph information duplication in the GPU and task pipelining whenever possible.

Unlike the CPU behavior, the synchronization overheads increase much faster with a growing number of tasks in the GPU due to the high communication latencies associated with them. Graphs with more than 256 million edges (4M nodes with 128 arity and 8M nodes with 64 arity) need to be divided into a higher number of tasks in order to fit correctly in the GPU memory space. As a result, performance is severely affected due to such synchronization overhead.

Finally, the largest graph (8M vertices with 128 arity) also needs to be mentioned given its extremely low processing rate of 100 Million TEPS. For this particular case, the graph and the task information cannot be allocated entirely in the GPU. Because of this, the OmpSs runtime keeps evicting task information in order to allow free space for upcoming tasks, thus losing the benefit of graph information caching. As a result, the memory bus becomes the limiting factor, making performance drop. Save for this specific case, the hybrid scheme outperforms the CPU-only implementation by a factor

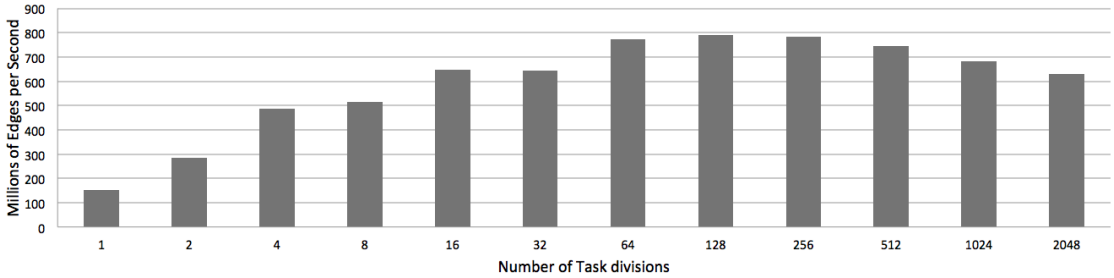

Fig. 9. CPU-only implementation performance

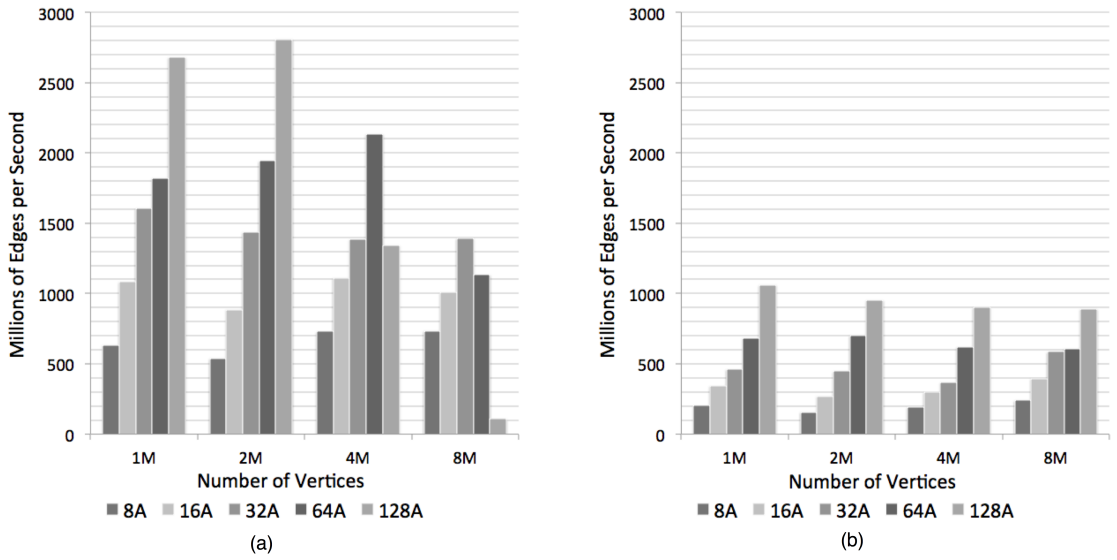

Fig. 10. (a) Workload-aware hybrid BFS performance (b) Fixed-partitioned-space hybrid BFS performance

of 1.5 in the worst case. Depending on the graph arity, the difference increases.

The fixed partitioning approach introduces the notion of cooperative execution between components by assigning a set of tasks to be executed in the CPU and another bound to the GPU. Figure 10.b shows the performance rates obtained with the fixed partitioning implementation under the same experiment conditions. Contrary to our initial expectations, the results obtained are much lower than those obtained with our alternative scheme; ranging from 200 Million to 1 Billion TEPS depending on the graph and the available parallelism. It is also interesting to note that, despite the poorer rates, the implementation scales well with larger graphs.

The general drop is caused by a performance bottleneck associated with the asymmetric processing properties of both CPUs and GPUs. It has been assumed in the paper that both components perform best under a different sort of circumstances. We take advantage of this fact in the first scheme by using the optimal component every iteration. However, employing both cooperatively in a synchronized fashion implies summing together the weaknesses as well as the virtues. In effect, small vertex frontiers cause a high GPU overhead when traversed, incurring a performance penalty. Large vertex frontiers also cause an unbalanced execution, where GPUs severely outperform CPUs. Because the approach is level synchronous, the performance is seriously unbalanced at every iteration. As a result, the CPU-only implementation outperforms the fixed partitioning scheme because of its superior efficiency when traversing small vertex frontiers and the avoidance of GPU transfer costs.

Along with OmpSs, the Barcelona Supercomputing

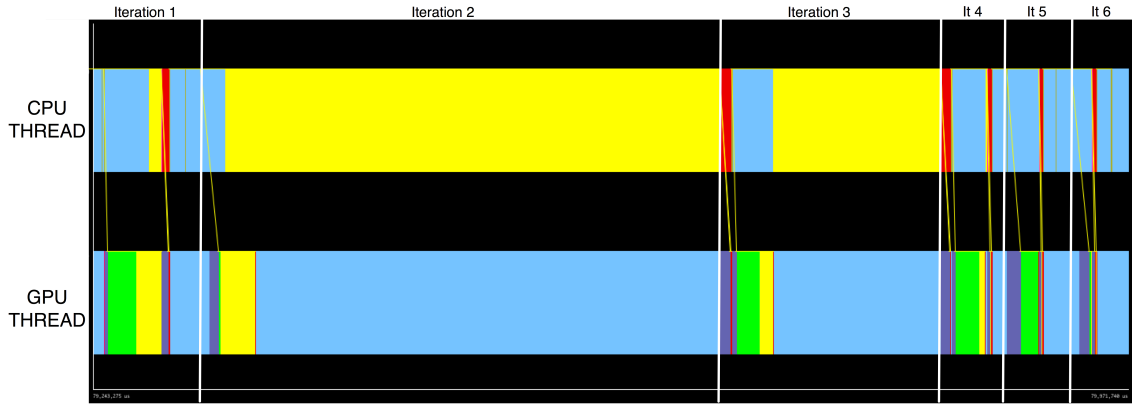

Fig. 11. Execution flow and load balancing during a graph traversal. Execution is shown in yellow and the idle time in blue

Center team also provides Paraver, a comprehensive suite of analysis and profiling tools. Figure 11 shows an execution diagram captured with Paraver that depicts the evolution of a graph traversal with 6 iterations. For illustrative purposes, the number of threads used for the experiment is reduced to two. The first bar shows the task executions (yellow) bound for CPU, while the second displays the ones bound to the GPU. The idle time is marked in sky blue. We can observe a remarkable load imbalance in the suboptimal component every iteration.

VIII. Conclusions And Future Work

In this paper, we introduce two strategies for tackling the Breadth-First Search graph traversal using a hybrid system composed of CPUs and GPGPUs. While one approach proves to be more effective than the other, we show that common problems associated with the GPU can be overcome and high processing rates can be achieved.

In addition, we decouple the parallelization strategy from the underlying hardware by introducing the task notion applied to graph traversals. The task subdivision adds several beneficial side-effects to the graph traversal problem, such as effective CPU load-balancing and an intuitive solution to process graph explorations on hybrid environments.

Adapting the hardware to the execution flow and the graph requirements has proven to be a successful strategy for tackling graph traversals despite the limitations GPGPUs show at present. Some of the shortcomings, such as memory latencies and bandwidth capacities, are being improved with new GPU generations, making hybrid environments more effective and efficient.

It is also worth noting that our implementation supports the traversal of graphs that do not fit the GPU main memory. Although the performance drops severely in this case, it may become a viable and effective solution if memory latencies improve in the foreseeable future.

Further research will be oriented toward exploring new hybrid strategies and improving the coordination between the CPU and GPU executions. One of the aspects to be examined is the automatic tuning of the lambda parameter to achieve optimal performance in each execution case regardless of the graph topology.

With our approach, the BFS problem can be subdivided and distributed through multiple processing nodes. Even though we propose two strategies, more execution schemes will be contemplated as part of the continuation work, including the combination of the described strategies and a dynamic load-balanced version of the Fixed-partitioned-space hybrid BFS.

IX. Acknowledgements

We would like to thank the Barcelona SuperComputing Center for the computing resources and specifically the team members Rosa Badia, and Judit Planas for their helpful support with OmpSs.

This work is partially supported by the Spanish Ministry of Science under contract TIN2006-60625.

This work (Bader) was also supported in part by NSF Grant CNS-0708307, Pacific Northwest National Laboratory Center for Adaptive Supercomputing Software for Multithreaded Architecture (CASS-MT), and by the Intel Labs Academic Research Office through the Parallel Algorithms for Non-Numeric Computing Program.

References

[1] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, U. Catalyurek, A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene, *ACM/IEEE SC Conference (SC05), Seattle, WA, USA* 2005 , pp. 25–25.

[2] D. A. Bader, K. Madduri, Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA–2, *2006 International Conference on Parallel Processing (ICPP06)*, Columbus, OH, USA, pp. 523–530.

[3] P. Harish, P. J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, *Proceedings of the 14th international conference on High performance computing,* Berlin, Heidelberg, 2007, p. 197-208.

[4] P. Harish, V. Vineet, P. J. Narayanan, Large Graph Algorithms for Massively Multithreaded Architectures, *Tech. Rep. IIIT/TR/2009/74.*

[5] M. Hussein, A. Varshney, L. Davis, On Implementing Graph Cuts on CUDA, *First Workshop on General Purpose Processing on Graphics Processing Units,* Boston, MA, 2007.

[6] D. Merrill, M. Garland, A. Grimshaw, High Performance and Scalable GPU Graph Traversal, Technical Report CS–2011–05

[7] Y. Deng , B.D. Wang, S. Mu, Taming Irregular EDA Applications on GPUs, *ICCAD 2009. IEEE/ACM International Conference,* 2–5 Nov. 2009.

[8] L. Luo, M. Wong, W. Hwu, An effective GPU implementation of breadth-first search, *Proceedings of the 47th Design Automation Conference,* New York, NY, USA, 2010, p. 52-55.

[9] S. Hong, S. K. Kim, T. Oguntebi, K. Olukotun, Accelerating CUDA graph algorithms at maximum warp, *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming,* New York, NY, USA, 2011, p. 267-276.

[10] S. Hong, T. Oguntebi, and K. Olukotun, Efficient Parallel Graph Exploration for Multi-Core CPU and GPU, *Parallel Architectures and Compilation Techniques*, New York, NY, USA, 2011.

[11] A. Buluc, K. Madduri, Parallel breadth-first search on distributed memory systems, *Proc. ACM/IEEE Conference on Supercomputing* Seattle, WA, USA, 2011.

[12] V. Agarwal, F. Petrini, D. Pasetto, D.A. Bader, Scalable graph exploration on multicore processors, *Proc. ACM/IEEE Conference on Supercomputing (SC10),* November 2010.

[13] Y. Xia, V.K. Prasanna, Topologically adaptive parallel breadth-first search on multicore processors, *Proc. 21st International Conference on Parallel and Distributed Computing Systems,* November 2009.

[14] A. Duran, E. Ayguade, R. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures, *Parallel Processing Letters,* vol. 21, no. 2, pp. 173-193, 2011.

[15] The Graph500 List, Internet: http://www.graph500.org, [May 04, 2012]