Parallel Graph Exploration on Multi-Core CPU and GPU

Vlad-Doru Ion

University of Bucharest

6 februarie 2015

Overview

Abstract

- In graph-based applications, a systematic exploration of the graph such as a breadth-first search (BFS) often serves as a key component in processing massive data sets.
- The paper presents a new method for implementing parallel BFS on multi-core CPUs which exploits a **fundamental property of randomly shaped real-world graph instances**[1].
- We then propose a hybrid method which, for each level of the BFS algorithm, dynamically chooses the best implementation from:
  - A sequential execution.
  - Two different methods of multicore execution.
  - A method which uses the GPU rather than the CPU.

[1] This will represent a key aspect in the construction of the solution.

Introduction

- Multi-core CPUs have become commonplace.
- The GPU [2] can be used to improve the performance of many traditional computation-intensive workloads.
- However, graph exploration is a problem that demands fast computation for whom efficient parallel or heterogeneous (using both CPU and GPU) implementations have yet to be identified.
- Significant research has been conducted to efficiently implement a parallel BFS for a wide array of computing systems.
- One problem is the random nature of the memory access patterns that occurs in graph processing algorithms.

[2] SIMD - Single Instruction Multiple Data architecture

Existing algorithms

- Agarwal et al's work which presented a state-of-the-art BFS implementation for multi-core systems.
  - Their implementation utilized sophisticated data structures to reduce cache coherence traffic between CPU sockets.[3]
- Hong et al solved the workload imbalance issue when processing irregularly shaped graphs, which had a devastating effect on previous GPU implementations.
  - They demonstrated good performance improvement compared to multi-core CPU implementations.
- This paper builds upon ideas from both previous works and incorporates them into a universal solution that utilizes both the CPU and GPU on a **heterogeneous system**.

[3] These structures try to address the random nature of the memory access patterns of the algorithm.

Contributions of the paper

- Present a BFS implementation method for multi-core CPUs which performs better than current state-of-the art implementations for large graph instances, while being simpler to implement.
- Present a hybrid method which dynamically chooses the best execution method for each BFS-level iteration from among: sequential execution, multi-core CPU executions, and GPU execution.
- Provide a fair comparison of the BFS performance on multi-core CPU and GPU, which reveals that singlesocket high-end GPU performance can be matched by a quad-socket high-end multi-core CPU.

Nature of Parallel BFS Algorithm on Irregular Graphs

Two different strategies have been proposed for parallel (and distributed) execution of BFS.

- The first method, known as the **fixed-point algorithm**, continuously update the BFS level of every node, based on BFS levels of all neighboring nodes until no more updates are made
- This method is preferred in distributed environments since it can naturally be implemented as message passing between neighboring nodes
- This method potentially wastes computation, since it processes the same edge multiple times whenever a corresponding node is updated

Nature of Parallel BFS Algorithm on Irregular Graphs

The second strategy is called **the level synchronous** BFS algorithm.

Algorithm 1 Level Synchronous Parallel BFS

```
1:  procedure BFS(r:Node)
2:      V = C = ∅; N = {r}          ▷ Visited, Current, and Next set
3:      r.lev = level = 0
4:      repeat
5:          C = N
6:          for Node c ∈ C do                          ▷ in parallel
7:              for Node n ∈ Nbr(c) do                  ▷ in parallel
8:                  if n ∉ V then
9:                      N = N ∪ {n}; V = V ∪ {n}
10:                     n.lev = level + 1
11:         level++
12:     until N = ∅
```

In short, this method visits all the nodes in each BFS level in parallel, with the parallel execution being synchronized at the end of each level iteration.

Nature of Parallel BFS Algorithm on Irregular Graphs

Here we can see how a real world graph would have it's nodes distributed across levels.

Level	Num. Nodes	Fraction (%)$^{+}$
0	1	$3.1*10^{-6}$
1	4	$1.3*10^{-5}$
2	749	$2.0*10^{-3}$
3	109,239	0.34
4	7,103,690	22.20
5	9,088,766	28.40
6	130,298	0.41
7	172	$5.3*10^{-4}$
total visited nodes	16,432,919	51.35
total visited edges	255,962,977	99.99

$^{(+)}$ Fraction of the total number of nodes (edges) in the graph

TABLE I
NUMBER OF NODES IN EACH BFS LEVEL: A RESULT FROM TYPICAL EXECUTION IN AN RMAT GRAPH

Observations

- We need to pay the price of synchronizing at each level and also the parallelism is limited by the number of nodes in a given BFS level.
- Nevertheless, the strategy works quite well in practice for real-world graph instances that are irregularly shaped by nature. This is because it has been observed that the diameters of real-world graphs are small even for large graph instances, i.e. **the small world phenomenon.**
- Similarly, because of the small world phenomenon the number of nodes in each BFS level cannot help but grow very rapidly,
- Therefore the total execution time is bounded by the traversal of the levels with a higher number of nodes, but the degree of parallelism is large there.

A new method for Multi-Core CPU

- In implementing the level synchronous parallel BFS algorithm there exists a rather direct implementation based on the presented algorithm which uses lock-protected shared queues.
- In order to avoid significant locking overhead the following optimizations can be applied:
  - Use of a bitmap to compactly represent the visited set
  - Application of the 'test and test-and-set' operation when atomically update the bitmap.
  - Use of local next-level queues; process node insertions into the global queue in batch.
- We refer to this method as the Queue-based method in the rest of this paper.

Queue-based method

```
BFS_Queue(G: Graph, r: Node) {
  Queue N, C, LQ[threads];
  Bitmap V;
  N.push(r); V.set(r.id);
  int level = 0; r.lev = level;
  while (N.size() > 0) {
    swap(N,C); N.clear(); // swap Curr and Next
    fork();
    for(c: C.partition(tid)) {
      for(n: c.nbrs) {
        if (!V.isSet(n.id)) {
          if (V.atomicSet(n.id)) {
            n.lev = level+1;
            LQ[tid].push(n); // local queue
            if (LQ[tid].size()==THRESHOLD){
              N.safeBulkPush(LQ[tid]);
              LQ[tid].clear();
            }
          }
        }
      }
    }
    if (LQ[tid].size() > 0) {
      N.safeBulkPush(LQ[tid]);
      LQ.clear();
    }
  join;
  level++;
  }
}
```

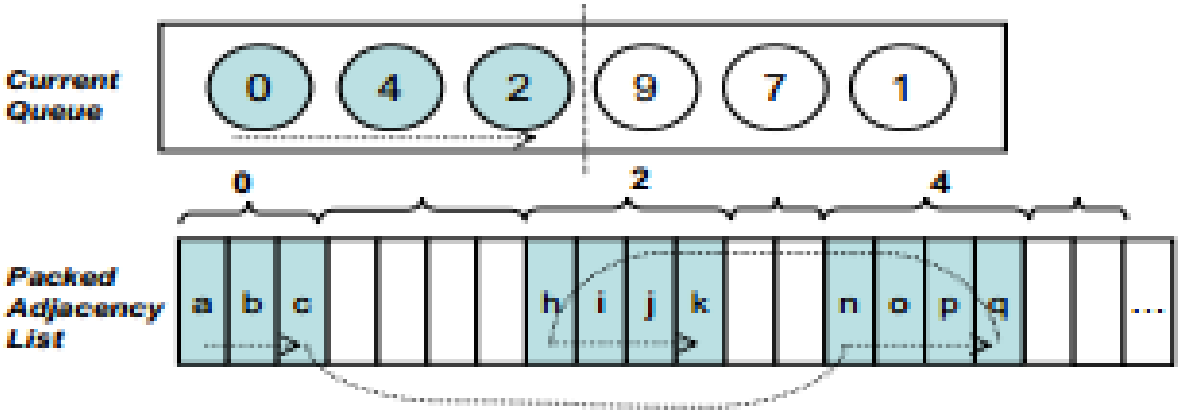
The read-based method

- Instead of a shared queue, the GPU implementations manage a single O(N) array.
- Fundamentally, our approach merges and builds upon the key ideas from the previous approaches for CPU and GPU
- In contrast to the Queue-based method, the next-level set and the current-level set are implemented together as a single O(N) array as in the GPU implementation
- The Read-based method provides two major advantages. First, it is completely free from queue overhead. Not only do we remove atomic instructions previously used for the queue operations, we also save on cache and memory bandwidth. Second, the Read-based method's memory access pattern is more sequential.

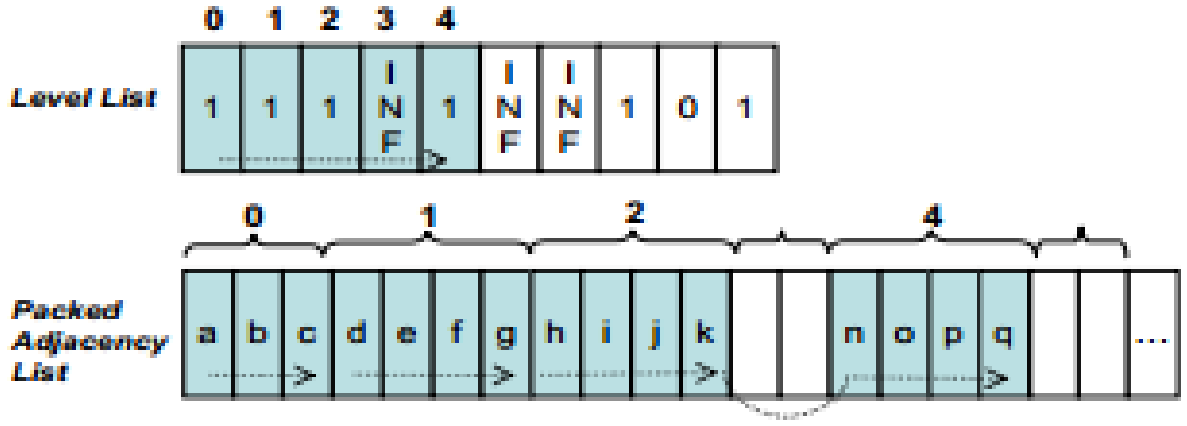
The read-based method

```
BFS_Read(G: Graph, r: Node) {
  Bitmap V;
  Bool fin[threads];
  V.set(r.id);
  int level = 0; r.lev = level;
  bool finished = false;
  while (!finished) {
    fork();
    fin[tid] = true;
    for(c: G.Nodes.partition(tid)) {
      if (c.lev != level) continue;
      for(n: c.nbrs) {
        if (!V.isSet(n.id)) { // test and test-and-set
          if (V.atomicSet(n.id)) {
            n.lev = level+1;
            fin[tid] = false;
          }
        }
      }
    }
    join;
    finished = logicalAnd(fin, threads);
    level++;
  }
}
```

The read-based method memory access



(a) Data-Access Pattern of Queue-Based Method



(b) Data-Access Pattern of Read-Based Method

The read-based method performance

- The primary disadvantage of the Read-based method is that it reads out the entire level array at every level iteration, even if only a few nodes belong to that level. However, this seldom affects the overall performance because of the following characteristics of real-world graph:
  - The diameter of the graph is small so the maximum amount of re-read is bounded
  - there are a few critical levels in which the number of nodes is O(N)
  - In addition, the total algorithm execution time is already governed by the processing time of these critical levels.
- *We remind the reader that this small world property is not merely an observation made in certain graph instances, but rather a fundamental characteristic of randomly-shaped realworld graphs*

Hybrid Methods

- To address the inefficient processing of non-critical levels, we propose a hybrid scheme that dynamically determines which method to apply when processing each level.
- The basic idea is simple: If the current level contains only a few nodes, use the Queue-based method. Otherwise, use the Read- based method.
- Our hybrid method can be represented as a state machine

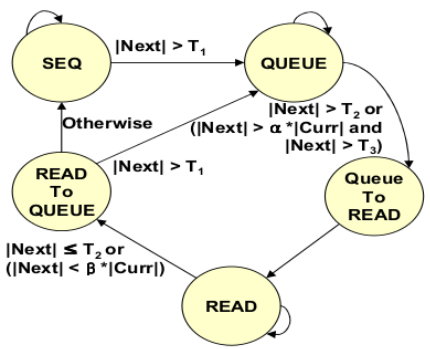


Fig. 4. State machine of Hybrid Read and Queue Method (for CPU execution).

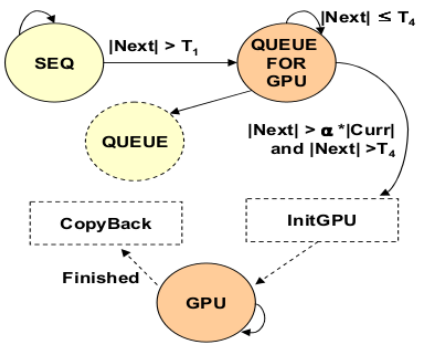
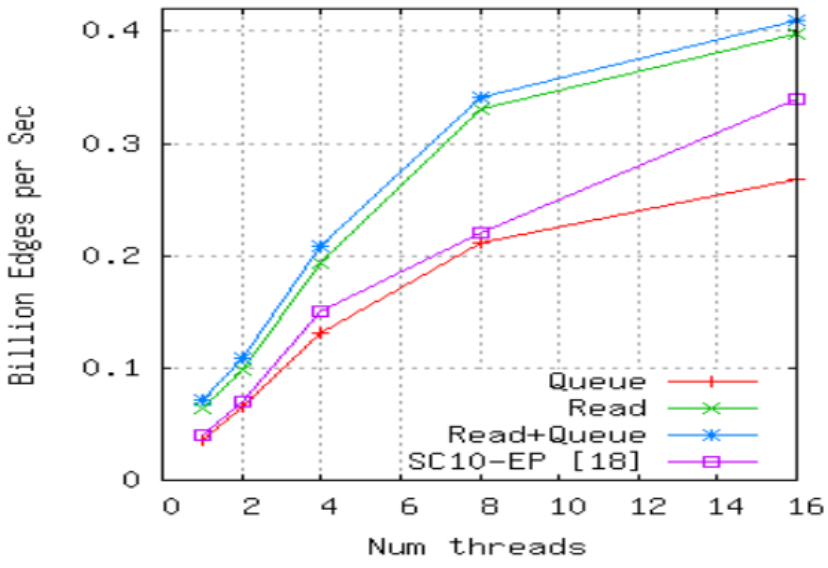
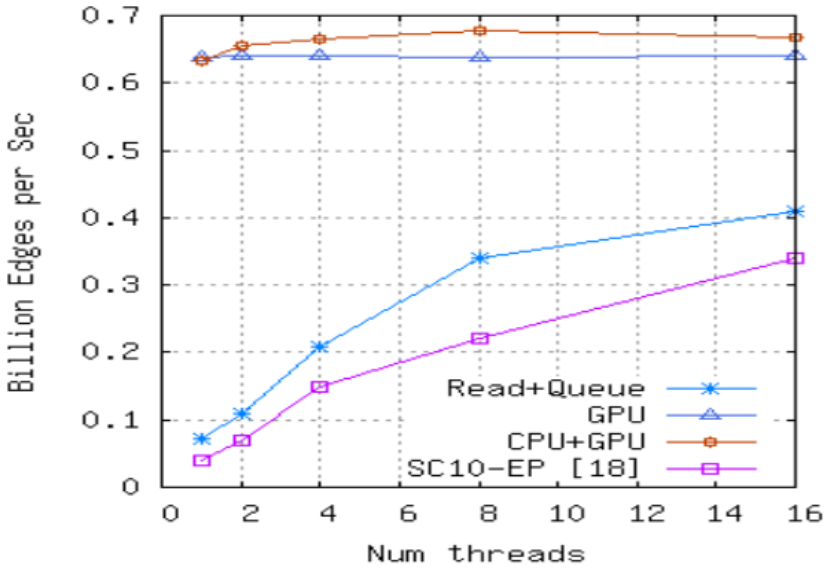


Fig. 5. State machine of the Hybrid CPU and GPU Method.

Results



(b) RMAT



(b) RMAT

Conclusions

- The multi-core CPU methodology is simple to apply yet efficient in utilizing memory bandwidth.
- The method outperforms the state-of-the-art method by up to 45%, with the performance gap widening as the graph size grows.
- We also propose a hybrid method that dynamically chooses the best implementation for each BFS-level iteration–such a method benefits both large and small graphs while preventing poor worst case performance.

Referințe

Sungpack Hong, Tayo Oguntebi and Kungle Olukotun

Efficient Parllel Graph Exploration on Multi-Core CPU and GPU.

Stanford Unifersity (2011)

Mulțumesc