Ordered and Unordered Algorithms for Parallel Breadth First Search [*]

M. Amber Hassaan
Department of Electrical and Computer Engineering
University of Texas at Austin
amber@mail.utexas.edu

Martin Burtscher
Institute for Computational Engineering and Sciences
University of Texas at Austin
burtscher@ices.utexas.edu

Keshav Pingali
Department of Computer Science
University of Texas at Austin
pingali@cs.utexas.edu

ABSTRACT

We describe and evaluate ordered and unordered algorithms for shared-memory parallel breadth-first search. The unordered algorithm is based on viewing breadth-first search as a fixpoint computation, and in general, it may perform more work than the ordered algorithms while requiring less global synchronization.

Categories and Subject Descriptors:
D.1.3 [**Programming Techniques**] Concurrent Programming – *Parallel Programming* D.3.3 [**Programming Languages**] Language Constructs and Features – *Frameworks*

General Terms: Algorithms, Languages, Performance

Keywords: Parallel Breadth First Search, Multicore processors, Amorphous data-parallelism, Galois system

1. INTRODUCTION

Breadth-first search (BFS) is one of the standard traversals of undirected graphs. One formulation of BFS is the following: label each node n with an integer *level* that is the length of the shortest path from a given node $root$ to node n. Sequential BFS is implemented by maintaining a FIFO queue of nodes as they are discovered during the traversal [2].

```
Graph g = /* read in graph */
OrderedWorkSet ws = { <root,0> };
foreach (<n,l>: ws) { /* ordered-set iterator */
  for (Node m: g.neighbors(n)) {
    if (m.getLevel() == INF)) {
      m.setLevel(l+1);
      ws.add(<m, l+1>);
    }
  }
}
```

Figure 1: Galois pseudocode for ordered BFS algorithm

An obvious approach to parallelizing BFS is to process the FIFO elements in parallel. In the Galois system [3], this is easily accomplished by using an *ordered-set iterator*, as shown in Figure 1. A Galois work-set is used to hold nodes as they are discovered in the traversal; nodes are ordered in this set by their level numbers. Threads pull nodes from the work-set in any order and speculatively execute the body of the iterator. The runtime system (i) ensures that iterations commit in order, and (ii) rolls back conflicting iterations as needed to ensure forward progress.

[*]This work is supported in part by NSF grants 0923907, 0833162, 0719966, and 0702353 and by grants from IBM, NEC and Intel.

The requirement that iterations commit in order imposes significant synchronization overhead. One approach to reducing this overhead is to formulate BFS as a fixpoint computation (this is also known as a relaxation-style algorithm [2]).
Initialization:
$level(root) = 0$
$level(k) = \infty,\ \forall k\ other\ than\ root$
Fixpoint computation:
$level(n) = min(level(m) + 1,\ \forall m\ \in\ neighbors\ of\ n)$

This approach can be implemented by maintaining a work-set, initialized with the root node, to which a node is added whenever its level is lowered. A node from the work-set is processed by examining each of its neighbors, and lowering the level of the neighbor if needed. In principle, nodes from the work-set can be processed in any order, so this is an example of an *unordered* algorithm [3]. Pseudocode for this algorithm is shown in Figure 2. When this code is run in the Galois system, iterations are executed speculatively but they do not have to commit in order. There is an implicit barrier synchronization at the end of the loop.

In general, we would expect the unordered algorithm to perform more work than the ordered algorithm, but the amount of additional work depends on the scheduling policies used to pull nodes from the work-sets. On the other hand, there is far less global synchronization in this algorithm, so one might expect it to scale better than the ordered algorithm. The rest of this paper studies a number of optimizations for these two algorithms, and presents parallel performance results.

```
Graph g = /* read in graph */
WorkSet ws = { root };
foreach (Node n: ws)) {
  int level = n.getLevel() + 1;
  for (Node m: g.neighbors(n)) {
    if (level < m.getLevel()) {
      m.setLevel(level);
      ws.add(m);
    }
  }
}
```

Figure 2: Pseudocode for Fixpoint BFS algorithm

2. OPTIMIZATIONS

There are a number of optimizations that can be applied to the two algorithms.

Reducing synchronization in the ordered algorithm: To avoid having to commit iterations in order, the ordered algorithm can be re-implemented using nested loops as shown in Figure 3. The algorithm maintains two work-sets, one for the current level and another for the next level. Most of parallel implementations proposed in the literature follow this algorithm [1, 5, 7].

```
Graph g = /* read in graph */
WorkSet currWs, nextWs;
currWs.add(src);
int nextLevel = 1;
while (!currWs.empty()) {
  foreach (Node n: currWs) {
    for (Node m: g.neighbors(n)) {
      if (m.getLevel() == INF)) {
        m.setLevel(nextLevel);
        nextWs.add(m);
      }
    }
  }
  ++nextLevel;
  currWs = nextWs; nextWs = new WorkSet();
}
```

Figure 3: Pseudocode for Wavefront BFS algorithm

Avoiding abstract locks: To ensure isolation of iteration execution, the threads in Galois acquire exclusive abstract locks on nodes touched in an iteration. For example, in the Fixpoint algorithm, each iteration of the *foreach* acquires locks on the node removed from the work-set, and on its neighbors. However, since the $level$ of a node decreases monotonically, we do not need to acquire locks on the nodes as long as the updates to the $level$ are 1) atomic and 2) made visible to all the threads immediately. We can replace the `if` statement in line 7 in Figure 2 with a `while` loop that a thread executes as long as it can decrease the value of $level$. To make updates to $level$ atomic, we can use a primitive like compare-and-swap (CAS).

Similarly, locks are not required for safe execution of the Wavefront algorithm. There is a possibility of a data race between two threads processing nodes that share a neighbor, but this race is benign because all threads write the same $level$ value to all the nodes in the next level. Eliminating abstract locks has several advantages: 1) the overhead of locking is removed, and 2) the available parallelism in the algorithm is increased [4].

Work-chunking: The amount of work done per iteration of the *foreach* in Fixpoint and Wavefront is very small in sparse graphs. This makes the work-set a point of contention for threads, because the threads are mostly waiting to obtain new work items. Work-chunking [6] is a common technique to reduce the overhead of the work-set and alleviate the corresponding sequential bottleneck. Instead of removing one active node from the work-set, the threads remove a collection of nodes to be processed at a time. Similarly, instead of adding back one element at a time, a chunk of elements is added.

3. EXPERIMENTAL RESULTS

We implemented the optimized Fixpoint and Wavefront algorithms using the Galois system on a 16-core AMD Opteron machine. The machine has 4 quad-core Opteron chips, and runs Linux, with JVM 1.6 on top of it. We used a graph implementation from the Galois library that uses arrays to represent adjacency, and maintains an array of references to node data objects. We used a concurrent FIFO implementation for the unordered work-set, which is needed to control the amount of extra work done by Fixpoint implementations. When using work-chunking, each chunk maintains FIFO order locally, while chunks themselves are also ordered as FIFO.

Scalability: Figure 4 shows how the different implementations of the Fixpoint and Wavefront algorithms scale with respect to 1 thread. The graph is a sparse random undirected graph with 10 million nodes and 54 million edges. We see that naive implementations do not scale well, while each optimization improves the scalability of the implementation. The optimized Fixpoint implementation scales slightly better than Wavefront because Wavefront requires more synchronization (there is a global barrier after each level has been processed). However, Wavefront yields better speedup than Fixpoint mainly because updates in Fixpoint are atomic operations. With 16 threads, the optimized version of Wavefront yields a speedup of 13, while Fixpoint yields a speedup of 10.08 (the baseline for both speed-up measurements is a serial implementation that uses a FIFO).

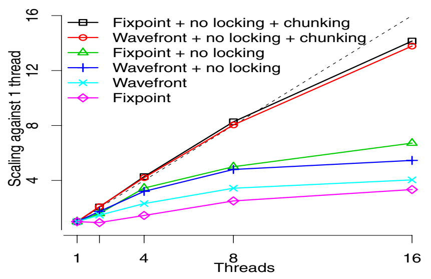

Figure 4: Scaling results for Fixpoint and Wavefront versions with different optimizations enabled. Input is a random graph with 10M nodes and 54M edges

Work statistics: As mentioned earlier, the parallel implementation of Fixpoint may perform some extra work compared to Wavefront. We define the work done as: $Ex + Up$, where Ex is the number of times a node's $level$ is examined and Up is the number of times the $level$ is updated. By this definition, the amount of work performed by sequential FIFO implementation is: $(|V| + 2|E|)$. Figure 5 compares the average amount of work done by optimized parallel implementations against a sequential implementation, for the input graph described above. We see that Fixpoint performs some amount of extra work, which increases with the number of threads. This extra work is primarily due to the deviation from FIFO order in the parallel execution, which is expected to increase with the number of threads. However, the loose FIFO order helps control the amount of extra work. Note that Wavefront performs little extra work by this measure.

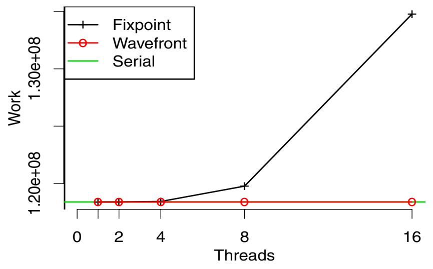

Figure 5: Amount of work done by optimized Fixpoint and Wavefront implementations

4. REFERENCES

[1] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 523–530, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[3] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI 2007)*, 42(6):211–222, 2007.

[4] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cașcaval. How much parallelism is there in irregular applications? In *PPoPP*, pages 3–14, 2009.

[5] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, pages 303–314, 2010.

[6] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, 1987.

[7] Yang Zhang and Eric A. Hansen. Parallel breadth-first heuristic search on a shared memory architecture. In *AAAI-06 Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, July 2006.