Software Prefetching for a Serial Breadth-first Search Algorithm

James J. Thomas and William Hasenplaugh

Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{jjthomas, whasenpl}@mit.edu
http://toc.csail.mit.edu/

Abstract.

1 Introduction

Graph algorithms (e.g. coloring, breadth-first search, max-flow etc.) often have poor cache performance due to poor data-locality. That is, for a general graph, algorithms which follow edges from vertex to vertex invariably experience poor data-locality unless the graph has a convenient structure (e.g. a grid). In this project, we examine the cache behavior of a breadth-first search algorithm and develop our own implementation which uses software prefetching to overcome some limitations of this poor data-locality. Figure 1 gives an example of a typical graph representation, also used to represent sparse matrices in linear algebra applications, which is composed of a vertex array and an edge array. The vertex array contains an index into the edge array for each of n vertices. The edge array has a vertex ID, itself an index into the vertex array, for each out-edge of the associated vertex. The indirection associated with these indices causes poor data-locality and confounds the effectiveness of hardware prefetchers which only manage to find constant strides in the memory access pattern.

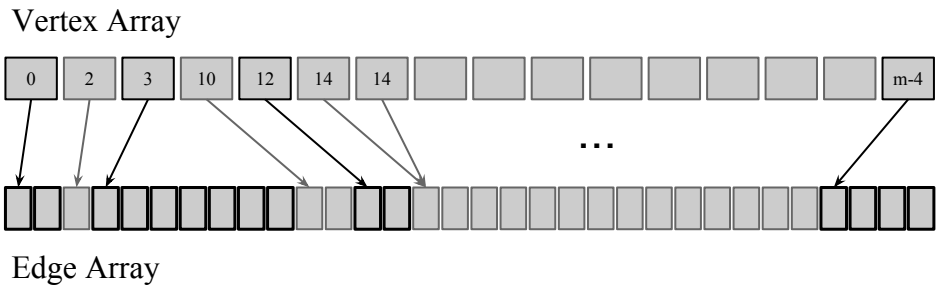

Fig. 1

Breadth-first Search

```
BFS(G, s)
 1  visited[1 : n] = False
 2  visited[s] = True
 3  s.dist = 0
 4  curQ.PUSH(s)
 5  nextQ = ∅
 6  while curQ.EMPTY() == False
 7      v = curQ.POP()
 8      for w ∈ v.Adj
 9          if visited[w] == True
10              nextQ.PUSH(w)
11              visited[w] = True
12              w.dist = v.dist + 1
13          if curQ.EMPTY() == True
14              curQ = nextQ
15              nextQ.CLEAR()
```

Fig. 2

We use *breadth-first search* (BFS) as a case study for improving performance of graph algorithms. Figure 2 shows pseudocode for the BFS algorithm which measures the minimum number of hops away from a source vertex s lies every other vertex in the graph. It begins by marking s at distance 0 and then proceeding in rounds, finding all vertices at each incremental distance. That is, it finds all neighbors of s, which lie at distance 1, and then find all unvisited neighbors of distance-1 vertices and so on.

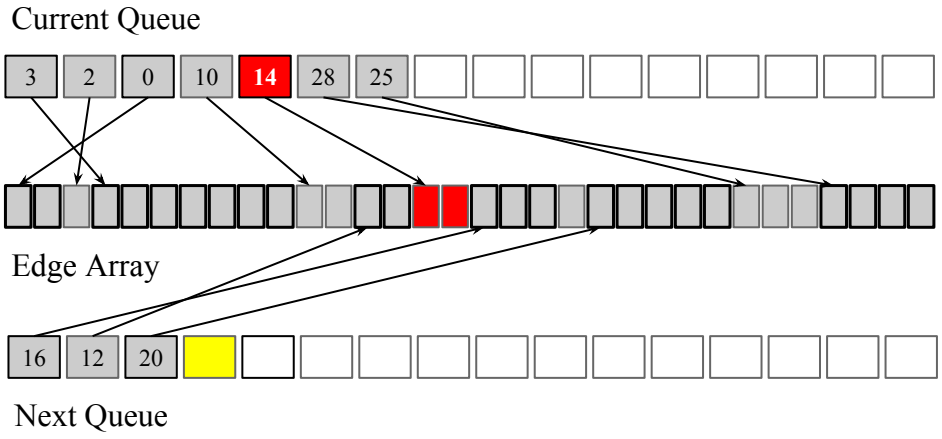

Fig. 3

In Figure 3 we see a pictorial view of the data access pattern induced by BFS in Figure 2. We use a slightly modified graph representation in order to store the index into the edge array in $curQ$ and $nextQ$, rather than the vertex ID, as in Figure 2. The

modification involves using the highest order bit in the unsigned integer used for the vertex ID to toggle between runs of neighbors in the edge array, pictured in the edge array in Figure 3 by alternating light and dark borders of edges. In this way, we can detect the boundaries between sets of neighbors by the toggling of this upper bit.

Related Work

As a baseline for serial BFS, we used Andrew Goldberg's code, which was developed for the 9th DIMACS implementation challenge in 2006. In addition to suffering from the inherent indirection in implementations of BFS, Andrew's implementation also introduces indirection between elements of $curQ$ and $nextQ$, since he uses a linked-list for those data-structures [1].

2 Software Prefetching

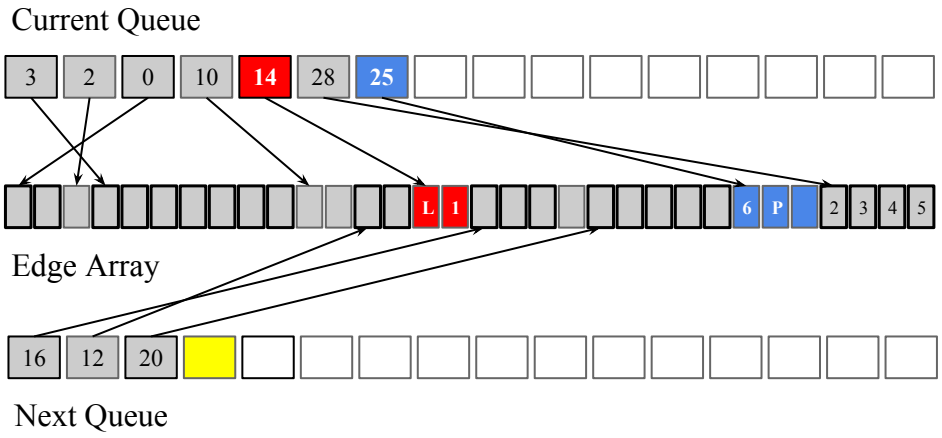

Fig. 4

This section discusses our use of software prefetching to combat poor cache performance due to memory indirection. In order to combat the indirection inherent in the sparse matrix representation of graphs, we can use software prefetching to marshal data in advance of its use. In particular, we use the software-prefetch primitive exposed by `GCC`, `ICC`, and `Clang/LLVM`: `__builtin_prefetch(&`ptr`,`R/W`,`$Level$`)`, where $\&ptr$ is the address to be prefetched, R/W is a boolean specifying whether the intention is to read or write the data, and $Level$ ($\in \{0, 1, 2, 3\}$) specifies what level of temporal locality is expected of the data. In particular, we walk $curQ$ at a constant number of edges ahead, as depicted in Figure 4, prefetching the vertex pointed to by the associated entry in the edge array. In addition, we use an array to hold $curQ$ and $nextQ$, as opposed to a linked-list as with Andrew Goldberg's implementation. This allows us to easily follow

[1] Should we need to make a footnote ...

$curQ$ without cache misses, since walking it linearly triggers the stream prefetcher in the processor.

N	M	AVG	BFS (speedup vs. AVG)	BFSPREFETCH (speedup vs. BFS)
100K	1M	0.012s	0.008s (33%)	0.008 (0%)
1M	10M	0.292s	0.172s (41%)	0.140s (19%)
10M	100M	4.160s	2.868s (31%)	2.296s (20%)

Fig. 5

Our results are summarized in Figure 5.

3 Virtual Memory

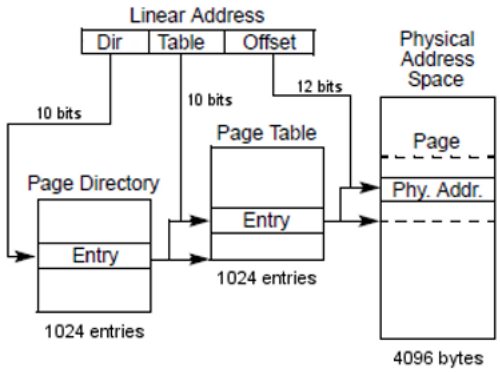

Fig. 6

This section discusses our findings from the process of trying to apply software prefetching to BFS. In particular, we found that the application is totally dominated by TLB misses. BFS applied to the test graph with 10M vertices and 100M edges caused 150M TLB misses over the course of initialization and BFS itself. This means that essentially every reference to the vertex array caused a TLB miss.