Locality Behavior of Parallel and Sequential Algorithms for Irregular Graph Problems

Guojing Cong and Tong Wen
IBM T.J. Watson Research Center
Yorktown Heights, NY, 10598
{gcong,tongwen}@us.ibm.com

June 14, 2007

Abstract

Large-scale graph problems are becoming increasingly important in science and engineering. The irregular, sparse instances are especially challenging to solve on cache-based architectures as they are known to incur erratic memory access patterns. Yet many of the algorithms also exhibit some degree of regularity with memory accesses. It is important to characterize the locality behavior in order to bridge the gap between algorithm and architecture. In our study we quantify the locality of several fundamental graph algorithms, both sequential and parallel, and correlate our observations with the algorithmic design. Our study of locality behavior brings insight into the impact of different cache architectures on the performance of both sequential and parallel graph algorithms.

1 Introduction

Graph abstractions are used in many science and engineering problems such as data mining, determining gene function, clustering in semantic webs, and security applications. Graph algorithms tend to lay high pressure on the memory system, and their locality behavior is crucial to the performance on architectures with deep memory hierarchy. Graph problems with large, arbitrary, sparse instances are hard to solve, especially in parallel, due to the irregular memory access pattern. Few implementations have achieved good performance on cache-based architectures (e.g., see [1, 2]). Characterizing the inherent locality behavior is important to high-performance solutions of such problems through either algorithmic or architectural innovation.

While measuring cache performance on a specific platform gives helpful hints to the locality behavior, the disadvantage is the dependence on the target architecture and the intricate interaction between cache and the input size. We propose using architecture-independent metrics that quantify the memory access locality as the signature for locality behavior. Independent of specific architectural choices, these metrics reveal characteristics of the algorithms that may serve as valuable input to architecture design.

In our study we are not interested in performance on a specific platform. In stead, we aim to capture the inherent feature that is relatively stable across platforms. We analyze the metrics and correlate them to the algorithmic designs. One of our focus is PRAM algorithms. Although in practice PRAM algorithms do not always achieve the best parallel performance, they are good candidates for locality behavior study for two reasons. The first is that they provide a large degree of parallelism, and usually take drastically different approaches than the sequential algorithms. In contrast, parallel algorithms based on other models resemble their sequential counterparts and have similar locality behavior. Secondly, there has been no quantitative locality study on PRAM algorithms, and such study may suggest new architectural support to speedup the execution. Interestingly, several promising results of PRAM algorithms have recently been reported on problems including connected components [3], parallel breadth-first search (BFS) [4] with multi-threaded architectures that do not rely on deep memory hierarchy to reduce the access latency.

We consider three fundamental graph problems, spanning tree (ST), minimum spanning tree (MST), and biconnected components (BiCC). We implement two sequential ST algorithms that are based on depth-first search (DFS) and breadth-first search (BFS), respectively. The parallel ST algorithm implements the Shiloach-Vishkin connectivity algorithm [14]. For the sequential MST we implement algorithms of Prim, Kruskal and Borůvka. Implicit binary heap [8] is employed in our implementation of Prim. Non-recursive merge sort is employed in Kruskal. In Borůvka the graph is not explicitly compacted. The locality of parallel Borůvka's algorithm is also studied. The sequential BiCC implementation is based on the classic Tarjan's algorithm that uses recursive DFS. The parallel implementation is based on the Tarjan-Vishkin algorithm [16]. We refer interested readers to [1, 2, 6] for implementation details of these algorithms.

In our study we do not consider algorithms explicitly optimized for cache performance, for example, external memory algorithms and cache-oblivious algorithms. These algorithms assume some memory hierarchy model, and minimize the number of block transfers between hierarchy levels. Common design techniques include divide-and-conquer and sequential scan, for which the I/O complexity is relatively easy to analyze. For other algorithms that do not employ these techniques, however, it is hard to capture the block transfers for the I/O complexity.

The main contribution of our study is the quantification of locality exhibited in graph algorithms with irregular inputs. In particular,

1. The temporal and spatial locality behavior of irregular graph algorithms are captured by two machine-independent metrics, that is, temporal and spatial distances. The two metrics provide good reference for efficient architecture support in deciding the cache size, cache line size, and multi-threading.

2. The algorithmic approach and locality behavior are correlated, thus it is possible to predict the sensitivity of an algorithm to changes in cache parameters. For example, we predict that Prim's algorithm has good performance with small cache lines, while Kruskal's benefits from large cache lines.

3. The differences in locality between parallel and sequential algorithms are com-

pared. In general, different sequential algorithms may have very different temporal locality behavior, while the parallel algorithms have similar temporal behavior. There exists a large amount of spatial locality in both the sequential and parallel algorithms with irregular inputs.

The rest of the paper is organized as follows. In Section 2 we present the metrics used to characterize locality behavior, and describe the inputs used in the experiments. Section 3 and Section 4 present the temporal locality behavior and spatial locality behavior of the graph algorithms, respectively. Section 5 shows some experimental results on current platforms. Section 6 is our conclusion and future work.

2 Characterizing Locality Behavior

Hierarchical storage system is based on the principle of locality. Temporal locality and spatial locality are proposed as two forms of locality by Madnick in [10]. General locality, both temporal and spatial, is defined as follows. *If the logical addresses* $\{a_1, a_2, \ldots, \}$ *are referenced during the time interval* $t-T$ *to* t*, there is a high probability that the logical addresses* $a_1 - A$ *to* $a_1 + A$*,* $a_2 - A$ *to* $a_2 + A$*, ..., will be referenced during the time interval* t *to* $t+T$. To characterize the locality behavior of a program, two parameters A and T are necessary. They relate to the reuse behavior of data fetched into fast memory and the spatial relationship among data recently used, respectively. In our study, both temporal locality and spatial locality are measured by analyzing the sequence of addresses issued from a program. The addresses are captured using a binary-rewriting tool SIGMA [9]. In order to compare with sequential algorithms, the locality of parallel algorithms running with p processors is captured by studying the simulation on 1 processor.

2.1 Temporal Locality Metric

Least-Recently-Used (LRU) stack distance, or reuse distance, was first used in the "stack processing" technique proposed by Mattson *et al.* for evaluating cost-performance of storage hierarchies [11]. Locality of a program can be studied by computing the LRU stack distance histogram (e.g., see [13, 7]).

Consider a trace of k memory accesses, $T = T_1, T_2, \ldots, T_k$, that access a set of c addresses, $M = M_1, M_2, \ldots, M_c$.We define function $f : T \rightarrow M$ such that $f(T_i) = M_j$, $T_i \in T$ and $M_j \in M$, if memory access T_i references address M_j. LRU stack distance for a memory access T_i is defined as follows. Suppose T_j is the latest access (before T_i) to $f(T_i)$, i.e., there does not exist l with $j < l < i$ such that $f(T_l) = f(T_i)$.

$$\Delta(T_i) = \begin{cases} \infty & f(T_i) \notin \bigcup_{k=1\ldots i-1}\{f(T_k)\} \\ \left|\bigcup_{k=j+1,\ldots i-1}\{f(T_k)\}\right| & \text{otherwise} \end{cases}$$

$\Delta(T_i)$ represents the number of distinct memory locations accessed between the previous and current access to some memory reference. It is infinity for the first access to T_i.

For a storage system with LRU replacement policy, access T_i is a hit if the size of the fast memory is larger than the stack distance $\Delta(T_i)$. A histogram is derived if we compute for each $\Delta \in [0 : c]$, the total number of accesses with reuse distance Δ. The histogram has been viewed as a machine-independent metric of locality (e.g., see [5, 13, 17]). Prior to our results, as far as we are aware of, there has been no locality study of graph algorithms with irregular inputs using the LRU stack distance metric.

2.2 Spatial Locality Metric

No machine independent metric has been proposed to measure the spatial locality of a program. Recently Snir and Yu studied the theoretical aspects of temporal and spatial locality [15]. They point out that in terms of predicting cache miss bandwidth, temporal locality and spatial locality can not be studied in isolation. In our study we use a simple metric as a measurement of spatial locality. For an address a_i and a range parameter K, the spatial distance d of a_i is defined as follows.

$$d = min\{|a_{i-K} - a_i|, |a_{i-K+1} - a_i|, \ldots, |a_{i-1} - a_i|\}.$$

Essentially d is the smallest distance between a_i and any of the A addresses issued before a_i. The parameter K defines how far back in the address trace we look to compute d. For a general program, it is impossible to predict cache behavior using d alone. However, when K is reasonably small, a small d suggests a high probability that a_i was already piggybacked into fast memory when the program accessed some address a_j ($j < i$) in history. This feature turned out to suffice for our study. We compute a histogram of the spatial distances for the input addresses.

2.3 Inputs

The locality behavior of a graph algorithm is influenced by the topology, edge density, and weight distribution (for MST) of the input. As we aim to explore the inherent locality of the algorithm, we choose random graphs to minimize regularities introduced by the input so that we can compare the locality behavior of parallel and sequential approaches. We avoid instances that can be easily partitioned and solved separately in sequential, where the locality behavior of the parallel and sequential approaches are similar.

We generate a random graph $G = (V, E)$ of n vertices and m edges by randomly picking a pair of vertices and connecting them with an edge until m edges are generated. Due to the time and memory constraint of instrumentation, in this paper all the inputs are of 1K vertices and 4K edges unless noted otherwise. Our study on larger inputs shows similar locality behavior.

3 Temporal Locality

In this section we present the temporal locality behavior of the algorithms for the ST, MST and BiCC problems. Fig. 1, 2, and 3 plot the "integral" of the histogram nor-

malized by the total number of memory accesses over the stack distance for the inputs. The x axis is the stack distance. The y value shows the percentage of accesses with stack distance no bigger than x. Alternatively, y can also be viewed as a cache hit ratio for a fully associative cache of size x with the LRU replacement policy. In the rest of the paper we refer to such plots as *ratio plots*.

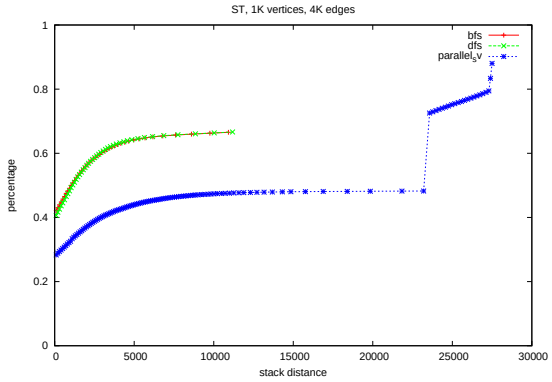

Figure 1: The ratio plots for the ST problem.

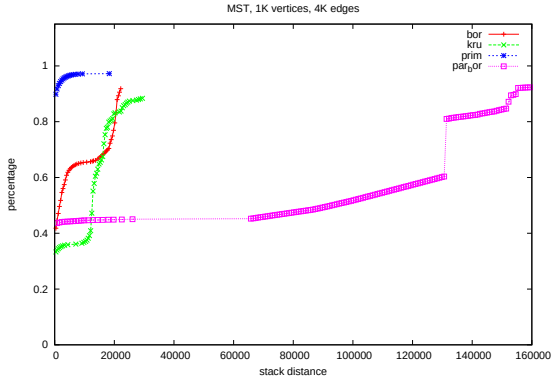

Figure 2: The ratio plots for the MST problem.

The curves in the plots have different shapes. We call the curve for the sequential algorithm the *sequential curve*, and the curve for the parallel algorithm the *parallel curve*. One observation is that, excluding infinity the maximum stack distances Δ_{max} (also the memory footprint) observed for the parallel algorithms are much larger than those of the sequential algorithms. In all three cases, the parallel algorithms consume significantly more, about 2 to 4 times, memory than their sequential counterparts, and this is largely due to the use of auxiliary data structures to facilitate parallelism.

At large reuse distances relative to the input size all the curves achieve near 100% hit ratios except for sequentail ST that will be discussed in Section 3.1. More inter-

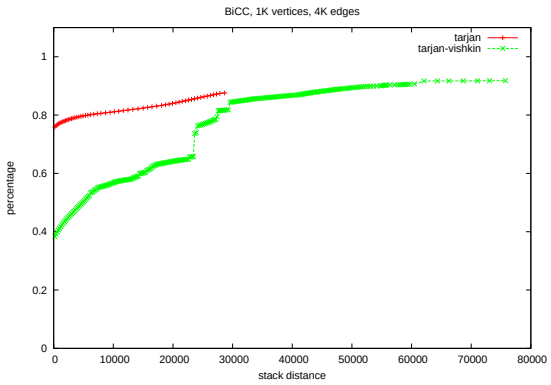

Figure 3: The ratio plots for the BiCC problem.

estingly, the parallel curves have more abrupt jumps that form a staircase pattern than the sequential curves. The staircase pattern is also observed in two of the sequential curves, that is, the curves for Kruskal and Borůvka. The jumps signify large histogram values at corresponding stack distances. The pattern is related to the phase behavior of the algorithms where the same data structures are scanned in different phases. Once the fast memory is large enough to hold these data structures, a jump in the ratio plot occurs. This pattern can also be considered as a signature for easily parallelizable algorithms, which is clearly true for Borůvka. Although Kruskal is inherently sequential, the bulk of the algorithm is about sorting. The non-recursive merge sort we employed can be parallelized to run in $O(\log n \log\log n)$ time with $n \log n$ work on EREW PRAM. In general, the sequential algorithms exhibit better temporal locality behavior than the parallel algorithms. As shown in Fig. 2, different sequential algorithms can have very different temporal locality behavior. The parallel curves, even for different algorithms, look largely similar.

3.1 ST

Neither the BFS curve nor the DFS curve achieves near 100% hit ratio. In fact the largest hit ratios they achieve are about 60%. BFS and DFS are of low complexity with small constansts. Assuming the adjacency array data structure, accesses to the graph are never reused. For a graph with n vertices and average degree d, there are dn memory accesses to the adjacency array. Reuse occurs only when the algorithm accesses the auxiliary color array and the stack (or queue). When there are only two colors possible for a vertex v with degree d_v, the color of v is queried d_v times and set once. Assuming an infinite amount of fast memory, d_v out of the total $d_v + 1$ accesses are reuse. Thus among $(d+1)n$ accesses to the color array, dn accesses are hits. Access behavior to the stack (or queue) depends on the topology of the input. A total of n elements are pushed in and popped out of the stack. The interleaving of the push and pop operations determines the reuse behavior of stack operations. The worst case is when all the vertices are first pushed into the stack and then popped out. Out of $2n$ accesses, n

of them are hits. Assuming that *push* or *pop* each involves two memory accesses, the stack operations generate $4n$ accesses. Let h be the hit ratio with a infinite cache, we have $\frac{dn+2n}{(d+1)n+dn+4n} \leq h < \frac{dn+4n}{(d+1)n+dn+4n}$. That is, $41\% \leq h < 48\%$. The upper bound is lower than the observed 60%. Further investigation shows that the number of memory accesses generated by stack operations is underestimated. Compiler listing shows that 10 and 11 load/store instructions are generated for the push and pop operations, respectively. Plugging in these numbers, the bounds become roughly $50\% \leq h < 78\%$. Our analysis shows that although very efficient, neither BFS nor DFS exhibits good resue behavior. Furthermore, the curves for BFS and DFS are almost identical. This suggests no significant advantage of either one with regard to temporal locality.

For most distances the parallel curve remains flat at a low hit ratio (about 40%). There are two steep increases. The parallel curve jumps first to around 70% at a distance slightly smaller than the input size. The second jump occurs at around Δ_{max}, which is exactly the size of input plus the auxiliary data structures. The reason that the first jump does not occur exactly at the input size is because the algorithm does not read every data element of an edge in the array when it is already in the tree.

3.2 MST

The MST plot shows that Δ_{max} for the parallel Borůvka's algorithm is significantly larger than that of the sequential algorithms. In addition to auxiliary data structures, this behavior is also due to the fact that the graph is compacted after each round and a new graph is generated.

Prim with implicit heap exhibits the best reuse behavior among the three MST algorithms. For different input sizes, ranging from $1K$ vertices to $100K$ vertices, a hit ratio of more than 70% is achieved with fewer than 20 words. In fact hit ratios of more than 80% are achieved with around 120 words, for all input sizes. Within the reuse distance range of $[0,50]$, the curves are nearly identical and the hit ratio appear to be independent of the input size, which suggests constant or logarithmic (in the input size) reuse distances for many memory accesses. Heap operations dominate the memory accesses in Prim. Our analysis estimated that about 40% to 50% of the memory accesses incurred by *Extract_Min*, *Insert* and *Decrease_Key* have constant reuse distances.

As the ratio plots show good locality of the simple binary heap, it is then interesting to compare with other more sophisticated heaps, for example, the sequence heap [12]. The sequence heap was shown to have better performance than the binary heap. However, our measurement shows that the binary heap has better temporal locality. Its poor performance is largely due to the imposed requirement of good spatial locality for performance on current architectures with long cache lines.

For Kruskal, sorting dominates the execution time, and dictates the shape of the plot. For an input with m edges, as each edge in the data structure has three elements (two vertices and the weight), the size of the total memory usage is $2m * 3 = 6m$ words (Note that the edge list representation contains two copies for each edge). The plots showed that a cache has to be of size at least $6m$ words in order to have reasonably good hit ratios. Otherwise the hit ratio is as low as 30%, even for cache size $6m-1$.

With Borůvka, the surges in the ratio plots are at distances in direct proportion to the input size, as in the case of Kruskal. More details of the temporal behavior of MST

can be found in [7].

3.3 BiCC

As the Tarjan-Vishkin algorithm employs many basic parallel primitives as building blocks such as prefix sum, pointer jumping, list ranking, sorting, connected components, spanning tree, Euler-tour construction and tree computations, it would be a laborious task to correlate the reuse behavior with the algorithmic behavior. Nevertheless, the phase behavior common to parallel algorithms shown as jumps in the curve is also observed. The small jumps in the zig-zag fashion on the parallel BiCC curve corresponds to cache sizes that can accommodate various auxiliary data structures.

4 Spatial locality

Good spatial locality behavior is crucial to performance on architectures with long memory access latency. As in the temporal locality study, we plot the integral of the histogram normalized by the total number of memory accesses over the spatial distance. We call such plots *spatial plots*. The x axis is the spatial distance, while the y value shows the percentage of accesses that have spatial distance no bigger than x. The y values can be considered as hit ratios when K addresses are fetched together into fast memory.

In Fig. 4, 5 and 6 are the spatial plots for the ST, MST and BiCC algorithms, respectively. At distance 64 all parallel algorithms achieve hit ratios of around 90%. In fact the parallel algorithms exhibit better locality behavior than the sequential algorithms in general. In Fig. 4, the spatial curves of BFS and DFS are again nearly

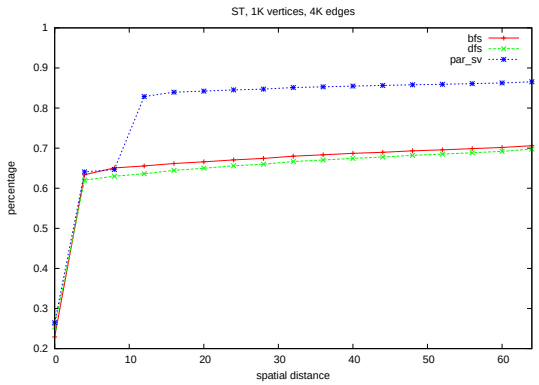

Figure 4: Spatial plots for ST algorithms with $K = 16$.

identical. The spatial curve of the parallel algorithm has two local peaks at distances 4 and 12. Recall that the Shiloach-Vishkin algorithm takes the edge list as input, and each edge $< u, v >$ is represented by a tuple (u, v, b), where b is a boolean type that records whether the edge is in the tree. There are also several auxiliary arrays of size

n, for example, the D array for applying the pointer-jumping technique to graft trees. Each of u, v, and b takes four bytes, thus initializing D and other auxiliary arrays incurs space distance 4 for each memory access. The algorithm scans through the edge array, and picks out edges that are not already labeled as tree edges for grafting. If the b field of an edge is set, the algorithm skips that edge. Accessing b when the edge is skipped incurs spatial distance 12 (one b per three words). When the edge is picked for grafting, accessing b incurs distance 4 as the two endpoints were accessed also. Pointer-jumping on the D array is expected to generate large spatial distances. However, the plots show that the majority of accesses have small spatial distances.

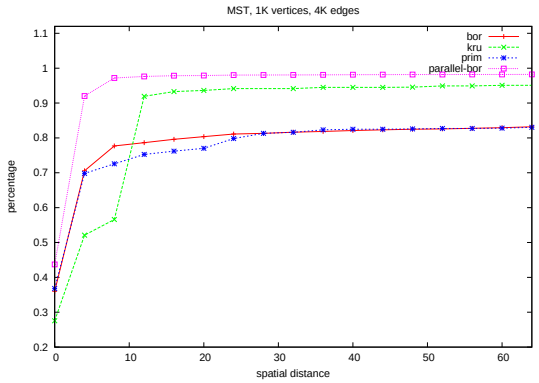

Figure 5: Spatial plots for MST algorithms with $K = 16$.

Among the sequential MST algorithms, Kruskal with merge sort shows the best spatial locality behavior with a hit ratio about 90% at distance 12. As sorting dominates the execution time, we expect to see many spatial distances with value 12 for the merge sort. Indeed, sorting is done on each edge $< u, v, w >$ of the edge array using the weight w as the key. The algorithm compares w for two edges, and as w is interleaved in the array (one w per three elements), a distance of 12 is observed if the two edges do not have to swapped. Swapping the two edges in the array generates distances of either 4 or 8. The specific 90% hit ratio observed also suggests that the Union-Find data structure for set operations does not generate many spatial distances with large values.

The locality behavior of Kruskal is much better than those of Prim and Borůvka. Both Prim and Borůvka achieve hit ratio of around 80% at distance 64. There is a big difference in spatial locality behavior between the sequential and parallel Borůvka. The sequential Borůvka exhibits the worst locality behavior of all algorithms. This is due to the extra level of indirect memory access introduced in sequential Borůvka in order not to compact the graph. For each vertex v its super-vertex is recorded in $D[v]$. Each operation on the edge goes through additional indirection to locate $D[v]$.

The parallel biconnected components algorithm achieves a ratio above 90% at spatial distance 36. In contrast, the sequential version only achieves around 70% ratio.

The y values at distance zero is worth noting. A spatial distance 0 for address a_i means that a_i appears in the K addresses issued earlier. The parameter K is related to the stack distance of a_i. More specifically, the stack distance of a_i is no bigger than

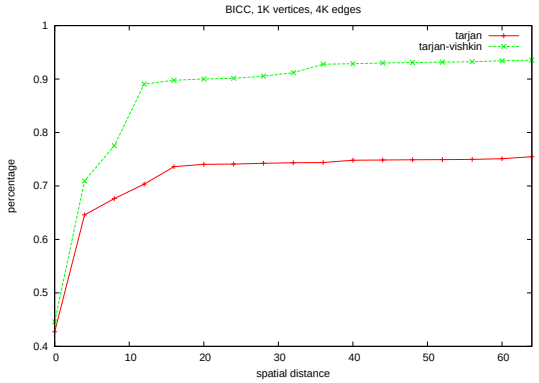

Figure 6: Spatial plots for BiCC algorithms with $K = 16$.

K. Conversely, however, a stack distance of K does not always imply a zero spatial distance for a_i with parameter K. As an example, correlating Fig 5 and Fig 7, we see the relationship of zero spatial distance with parameter K and stack distance at K. In

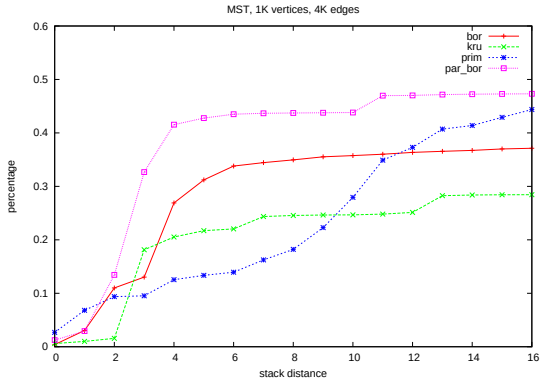

Figure 7: Relationship between zero spatial distance with parameter K and stack distance at K

Fig. 7, the hit ratio at stack distance 16 for each plot is slightly larger than the ratio at spatial distance zero in Fig 5. Similar behavior is also observed for the ST and BiCC problem.

5 Further Experiments and Discussion

It is hard to predict the caching behavior on real systems with either temporal locality or spatial locality alone. Our study of the locality behavior for the ST, MST, and BiCC algorithms suggest that between parallel and sequential algorithms, there is no clear

winner of superior locality behavior. To further validate our analysis, we conducted experiments to measure cache misses on real systems. Again we choose the MST algorithms as an example. We measure the L1 cache misses on an IBM Power4 system using hardware performance counters. On Power 4, hardware event groups 1 and 37 can be used to collect necessary metrics for the derivation of L1 cache hit ratio. Graphs with 1K vertices and 4K edges are too small for the purpose of cache performance study. Instead, we use a random graph with 100K vertices and 400K edges.

Algorithm	#Loads	#Stores	#Load misses	#Store misses	hit ratio
Prim	51058290	96477747	4452339	3699053	94.47
Kruskal	57899100	26125646	1462844	24692720	68.87
Borůvka	60672306	5340909	7366510	1976747	85.85
Par Borůvka	197261229	85735209	4392786	13551724	93.66

Table 1: Cache (L1) performance comparison of the MST algorithms on Power4.

In Tab. 1 are the numbers for loads, stores, load misses and store misses for the L1 cache. Prim has the best hit ratio, followed by parallel Borůvka. However, the rankings of hit ratio do not translate directly to the rankings of actual performance. In this case, although Kruskal has the worst hit ratio, it also has fewer load and store instructions and the best performance. Parallel Borůvka has a hit ratio of about 94%, far better than Borůvka and Kruskal. However, it also has far more load and store operations. Compared with Prim, Kruskal and Borůvka, parallel Borůvka does not have significantly worse cache performance. However, combined with the large number of memory accesses, the absolute number of misses is significant. Better architectural support is desired. For code regions in the parallel algorithms that extensively access the data structure in an irregular manner, multi-threaded architecture may effectively hide the long memory access latency.

6 Conclusion and Future Work

As memory hierarchy deepens, locality of an algorithm is becoming even more important to performance. We study the locality behavior of graph algorithms, both sequential and parallel, for the ST, MST, and BiCC problems. We propose using two machine-independent metrics for quantifying the temporal locality and spatial locality.

We show that for the problems considered, the sequential algorithms exhibit better temporal locality than the parallel algorithms. Yet good temporal locality does not always yield superior performance on architectures that impose the requirement of spatial locality. Parallel algorithms have better spatial locality, while the extra parallel overhead seems to offset the locality advantage. The locality behavior comparison of the parallel and spatial algorithms shows that the current cache-based architecture is not specially hostile to parallel graph algorithms although there are memory accesses that incur large distances. A hybrid cache-based and multi-threaded architecture might bring effective support to solving irregular graph problems in parallel.

In future work we will further investigate the impact of locality enhancing techniques on the performance of parallel and sequential algorithms. This is especially meaningful as many processors adopt multi-core designs. On one hand, it is important to design parallel algorithms with reasonable locality behavior. On the other hand, special architectural support might also be necessary to tolerate the memory access latency for parallel algorithms.

References

[1] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, Apr 2004.

[2] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, New Mexico, 2004.

[3] D.A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proceeding of the 2005 International Conference on Parallel Processing*, pages 547–556, 2005.

[4] D.A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *Proceedings of 2006 International Conference on Parallel Processing*, pages 523–530, 2006.

[5] C. Cascaval and D.A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 150–159, San Francisco, CA, 2003.

[6] G. Cong and D.A. Bader. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, Colorado, Apr 2005.

[7] G. Cong and S. Sbaraglia. A study of the locality behavior of minimum spanning tree algorithms. In *The 13rd International Conference on High Performance Computing*, page to appear, 2006.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Inc., Cambridge, MA, 1990.

[9] L. DeRose, K. Ekanadham, J.K. Hollingsworth, and S. Sbaraglia. Sigma: a simulator infrastructure to guide memory analysis. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–13, 2002.

[10] S.E. Madnick. *Storage Hierarchy Systems*. PhD thesis, MIT, 1973.

[11] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9:78–117, 1970.

[12] P. Sanders. Fast priority queues for cached memory. *ACM J. Experimental Algorithmics*, 5(7), 2000. `www.jea.acm.org/2000/SandersPriority/`.

[13] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 165–176, Bostaon, MA, 2004.

[14] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algs.*, 3(1):57–67, 1982.

[15] M. Snir and J. Yu. On the theory of spatial and temporal locality. Technical Report UIUCDCS-R-2005-2611, University of Illinois at Urbana-Champaign, 2005.

[16] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Computing*, 14(4):862–874, 1985.

[17] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 255–266, Washington, DC, 2004.