# A Study on the Locality Behavior of Minimum Spanning Tree Algorithms

Guojing Cong and Simone Sbaraglia

IBM T.J. Watson Research Center
Yorktown Heights, NY, 10598
{gcong,ssbarag}@us.ibm.com

**Abstract.** Locality behavior study is crucial for achieving good performance for irregular problems. Graph algorithms with large, sparse inputs, for example, oftentimes achieve only a tiny fraction of the potential peak performance on current architectures. Compared with most numerical algorithms graph algorithms lay higher pressure on the memory system. In this paper, using the minimum spanning tree problem as an example, we study the locality behavior of graph algorithms, both sequential and parallel, for arbitrary, sparse instances. We show that the inherent locality of graph algorithms may not be favored by the current architecture, and parallel graph algorithms tend to have significantly poorer locality behaviors than their sequential counterparts. As memory hierarchy gets deeper and processors start to contain multi-cores, our study suggests that architectural support and new parallel algorithm designs are necessary for achieving good performance for irregular graph problems.

**Keywords:** memory locality, graph algorithm, minimum spanning tree

## 1 Introduction

Graph abstractions are used in many science and engineering problems, for example, data mining, determining gene function, clustering in semantic webs, and security applications. Graph problems with large arbitrary, sparse instances are challenging to solve on current architectures (e.g., see [3, 4]). For dense linear algebra packages near peak performances are repeatedly reported. Yet we have not seen similar performance results for graph problems. Graph algorithms tend to lay higher pressure on the memory system. For architectures with deep memory hierarchy, locality features are crucial to the performance of the algorithms. In this paper, using the minimum spanning tree (MST) problem as an example, we study the locality behaviors of graph algorithms and compare their performances with different cache configurations.

The MST problem finds a spanning tree of a connected graph $G$ with the minimum sum of edge weights. MST is one of the most studied combinatorial problems with practical applications in VLSI layout, wireless communication, distributed networks [15, 24, 26], and recent problems in biology and medicine [5, 13], and national security [7]. MST is also often a key step in other graph problems [16, 17, 23, 25].

Moret and Shapiro give an empirical analysis of MST algorithms in [18]. Implementations of Prim's, Kruskal's and Cheriton-Tarjan's algorithms on several architectures are compared. Through extensive comparisons, Prim's algorithm is found to be the best candidate. Computer architectures have since evolved, and Prim's algorithm may no longer be the fastest on current platforms. Moreover, running times alone are generally not sufficient to estimate the relative performance of algorithms on new architectural configurations. As memory hierarchy gets deeper, cache performance becomes crucial to an application. Whether the locality behavior of an algorithm fits well with the cache configuration affects the overall performance. Understanding the locality behavior helps the adaptation of algorithms to target platforms and dynamic configurations (e.g., shutdown of a cache bank to reduce power consumption).

In this paper we study the locality behavior of three MST algorithms, that is, Prim's, Borůvka's, and Kruskal's, and show how cache configurations (e.g., cache size and line size) affect their performance. We include Borůvka's algorithm as it can be easily parallelized to run in poly-log time under the PRAM model. As processors increasingly adopt multi-core designs, solving a problem in parallel is important for performance. The locality behavior of a parallel graph algorithm can be very different from the sequential counterpart as the designs are drastically different. Comparison of their locality behavior brings insight to efficient parallel algorithm design and better architectural support.

Cache-friendly algorithms, for example, external memory algorithms and cache-oblivious algorithms, abound in the literature. These algorithms assume some memory hierarchy models, 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 (number of blocks transfered) is relatively easy to analyze. For other algorithms that do not employ these techniques, however, it is hard to analyze for I/O complexity under these hierarchy models. Also the locality behavior of an algorithm is an inherent property that should not depend on the memory hierarchy of a target platform (while its performance certainly depends on how well the locality behavior fits with the cache configuration). In our study we do not analyze the MST algorithms under these existing memory models. Instead we characterize locality through *Least-Recently-Used* (LRU) stack distance analysis that is discussed in Section 2.

## 2 Characterizing Locality Behavior

LRU stack distance was first used in the "stack processing" technique proposed by Mattson *et al.* for evaluating cost-performance of storage hierarchies [14]. LRU stack distance is also referred to as *reuse distance*, and the two names are used interchangeably in the literature. Locality of a program can be studied by computing the LRU stack distance histogram (e.g., see [21]).

Consider a trace of $k$ memory accesses, $T = T_1, T_2, \ldots, T_k$, that access a set of $c$ addresses. For a storage system with *Least-Recently-Used* 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 can be derived if we compute for each $\Delta \in [0 : c]$, the total number of accesses that have reuse distance $\Delta$. The LRU stack distance histogram has been used as a machine-

independent metric of locality (e.g., see [6]). With LRU stack distance analysis it is possible to perform various optimizations on a program (e.g., see [21, 27]).

In our study we use the binary rewriting approach to get a memory access trace. We intercept each *load* and *store* instruction using SIGMA [10], and compute the reuse distance histogram on the fly to avoid dumping huge traces.

## 3 Comparison of Three MST Algorithms

In this section we compare the locality behavior of the three MST algorithms, that is, Prim's, Kruskal's and Borůvka's. For each algorithm the exact process of constructing an MST is influenced by the topology, edge density, and weight distribution of the input graph. We focus on sparse random graphs with randomly-assigned edge weights. We choose random graphs because they are the most challenging to solve on parallel computers. As memory access pattern is highly dependent on the inputs, the study of arbitrary graphs can expose regular memory patterns of the algorithms. A random graph of $n$ vertices and $m$ edges is generated by randomly picking a pair of vertices and connecting them with an edge until $m$ edges are generated.

For Prim's algorithm (denoted as Prim), we use the implicit binary heap described [9]. For Kruskal's algorithm (denoted as Kruskal), we use non-recursive merge sort as the sorting routine. The union-find data structure is used to maintain the disjoint sets of elements. Borůvka's algorithm (denoted as Borůvka) is composed of Borůvka iterations that have three steps: *find-min*, *connect-components*, and *compact-graph*. The algorithm iterates until only isolated vertices are left. All MST implementations run in $O(m \log n)$ time. Fig. 1 shows the LRU stack distance histograms for each algorithm with an input graph of $1K$ vertices and $4K$ edges (we use $1K$ to denote 1024).

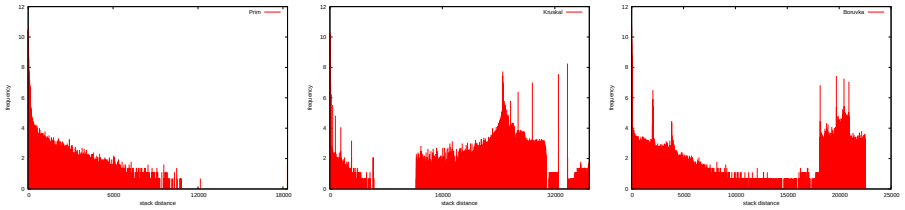


**Fig. 1.** Histograms of stack distances for three MST implementations.

One common feature of the three plots in Fig. 1 is the blanks in the histogram. The ratio of the number of observed distinct stack distances over the memory footprint size is 40% for Prim, 54% for Kruskal, and 73% for Boruvka. In each plot, the minimum reuse distance is 0, and the maximum is $c$, the size of the footprint. Large concentrations of distribution are observed around certain distances. For example, there are concentrations around small reuse distances in all plots. Each plot has a different shape. For Prim the histogram monotonously decreases with the reuse distance. For Borůvka and Kruskal, there are concentrations of distribution around large reuse distances.

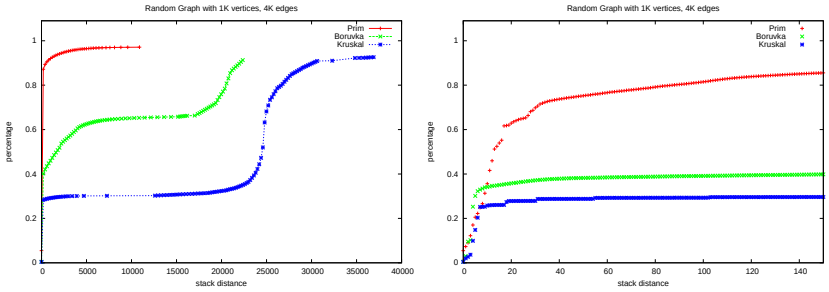


**Fig. 2.** The ratio plots for three implementation of MST algorithms with an input of $1K$ vertices, $4K$ edges.

The plot on the left of Fig. 2 presents a different view of the same histogram data. 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. For each of the three algorithms, the shape of the line in the ratio plots is different. Prim and Kruskal achieve fairly good cache hit ratios with small cache sizes. The curve of Borůvka remains flat at low ratios for a range of reuse distances, and jump abruptly at relatively large distances. The plot on the right of Fig. 2 is a zoomed-in view for reuse distances in the range of [0:150]. Prim achieves a hit ratio of over 80% at a cache size of only 120 words.

### 3.1 Locality of Prim

Starting from a single vertex, Prim grows an MST one edge at a time. Prim maintains a heap to retrieve the lightest-weight edge. During the execution, for sparse inputs, most memory accesses occur around accessing the heap data structure. Each heap operation incurs $O(\log h)$ memory accesses, where $h = O(n)$ is the size of the heap. We focus our analysis on the heap operations.

In our experiments, for all graphs of different 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 (integers), 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.

The magic numbers observed (i.e., 70% and 50 words) are dependent not only the topology, edge density, and weight distribution of the input, but also the actual programming of the algorithm. Instead of modeling the tree construction process and giving rough bounds, we show that a significant percentage of memory accesses incur short reuse distances.

Recall that a reuse distance is associated with each memory access. We now consider the reuse distance for the memory accesses incurred by *Extract_Min*, *Insert* and *Decrease_Key*. *Extract_Min* removes the top element of the heap, and places the last element as the new top. It then iteratively inspects a node and its two children starting

from the top. If the parent has larger weight, it then is swapped with one of the children. During each iteration there are three reads (reading the weights) and two writes (swapping). The parent and one of the children are accessed twice, and the second access has a distance of $O(1)$. More exactly, the distance will be 1 or 2 depending on whether the left or right child gets swapped. Here we do not consider the interference of other auxiliary data structures, for example, a temporary location to facilitate the swap. So at least $\frac{2}{5}$ of the accesses generated by *Extract_Min* are within a constant distance. *Insert* appends an element to the end of the heap and then compares iteratively from the end whether an element is larger than its parent. If the parent is larger, it then gets sifted down. Successive sifting incurs constant reuse distance, and $\frac{1}{2}$ of the accesses have distance $O(1)$. *Decrease_Key* works similarly as *Extract_Min*, and about $\frac{1}{2}$ of the accesses have distance $O(1)$. Note that although the distances are constant, in practice they can take a range of values due to book-keeping activities. For example, to enable *Decrease_Key*, the positions of each vertex in the heap are recorded in an array. Updating the positions increases reuse distances (still $O(1)$ though) for heap accesses in *Decrease_Key*. According to our analysis, an estimate of 40 to 50 percent of accesses have constant reuse distances (disregarding book-keeping activities).

In addition to constant reuse distances, some operation incur $O(\log n)$ distance. For example, to maintain the size of the heap, after each *Extract_Min* or *Insert*, a counter is either incremented or decremented. Access to the counter generates reuse distance of $O(\log n)$ as *Extract_Min* and *Insert* incur $O(\log n)$ accesses to different memory locations. The top of the heap is accessed every time in *Extract_Min*, and the largest reuse distance incurred by *Extract_Min* is $O(d \log n)$, where $d$ is the largest degree of all vertices. The distribution of reuse distances for the rest of memory operations is governed by the random process of constructing an MST. It is easy to construct scenarios that incur large (e.g., $O(n)$) reuse distances.

As the ratio plots show good locality of the simple binary heap, it is then interesting to compare with other more sophisticated implementations of heaps. Heaps (and priority queues) have been studied extensively, and quite a few data structures are proposed, for example, Fibonacci heap, pairing heap, and splay trees. Sanders presents a data structure called *sequence heap* [20] and shows that for a cache configuration with size $M$ and block size $B$, $I$ insertions and $I$ deletions can be performed with $I(2R/B + O(1/k + (\log k)/m))$ I/Os and $I(\log I + \log R + \log m + O(1))$ comparisons, where $m = \Theta(M)$, $k = \Theta(M/B)$, $R = [\log_k \frac{I}{m}]$. The motivation of *sequence heap* is based on the fact that merging $k$ sequences is I/O efficient under the external memory model. Arge *et al.* designed cache-oblivious priority queues based on similar observations [2]. In his study Sanders has four heap implementations, denoted as *hslow* (implicit binary heap), *h2* (binary heap with the "bounce" heuristic [12]), *h4*(4-ary heap), and *knh* (the *sequence heap*), respectively.

In Fig. 3 are the ratio plots for the four different heap implementations. Surprisingly, the textbook binary heap (*hslow*) has the best locality behavior in terms of reuse distances. At each distance, the ratio for *hslow* is consistently higher than the ratios for other implementations. In practice, however, *hslow* is found to be the slowest for most inputs on current architectures, for example, SUN SPARC V9 and IBM Power 4. In fact, *knh* are four times faster than *hslow* for many inputs. Although *hslow* tends to

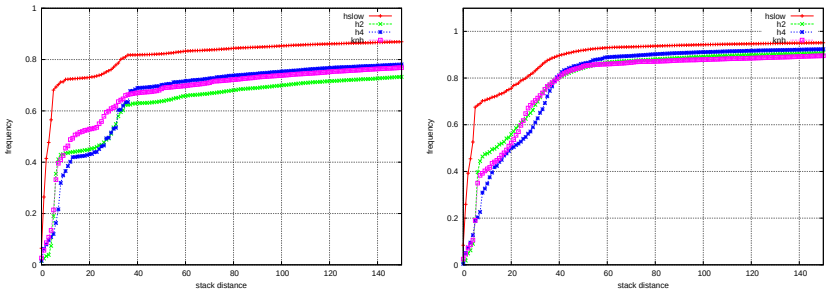


**Fig. 3.** The ratio plots for four heap implementations. The plot is for stack distance in range [0:150]. On the left are the plots for 1000 *Insert* followed by 1000 *Extract_Min*. On the right are the plots for 1000 *Insert*, *Extract_Min*, and *Insert* followed by 1000 *Extract_Min*, *Insert*, *Extract_Min*.

make more memory accesses (about 1.5 times as many as *knh*), the difference does not fully explain the observed poor performance of *hslow*, especially considering its good locality behavior. The fastest implementation is *knh*. As it mostly works with sorted sequences, it exhibits good spatial locality. Current architectures that typically have long cache lines and long latency to main memory impose the requirement of spatial locality for good performance. Unfortunately, spatial locality is scarce in *hslow*.

All heap operations start with a certain node $v$, and inspect $v$'s parent and/or children. Due to the layout of the implicit binary heap in memory, whenever a block is brought into the cache, except for node $v$ that is currently being accessed, it is unlikely that the rest of the block contains $v$'s parent or children unless $v$ is near the top of the heap. In this case, long cache line causes fetching data that is not used in the near future and wastes memory bandwidth.

There is no machine-independent metric in the literature to measure the spatial locality of a program. Recently Snir and Yu studied the theoretical aspects of temporal and spatial locality [22]. While they acknowledge that LRU stack distance analysis captures well temporal locality, they also point out that in terms of predicting cache miss bandwidth, temporal locality and spatial locality can not be studied in isolation. We present further experimental results in Section 4.

### 3.2 Locality of Kruskal

For Kruskal, sorting dominates the execution time, and dictates the shape of the plot. For the implementation with merge sort, the hit ratio remains low until the distance and hence cache size becomes very large. In fact only at a size that can hold all the data structures used for sorting does the hit ratio reach above 90%. Fig. 4 shows the ratio plots for Kruskal with three different inputs. The vertical line in each plot is $\Delta = 6m$. Recall that non-recursive merge sort employs an auxiliary buffer. 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. The plots show 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 $6m-1$. Unfortunately, $6m$ is in direct proportion to the input size, and the algorithm exhibits poor temporal locality behavior.

In practice, for many inputs, Kruskal with merge sort is the fastest among all implementations. As long as the the data structure fits in main memory, our implementation with merge sort beats the version with quick sort for large inputs on all tested platforms. This is largely due to the fact that merge sort has very good spatial locality that are especially advantageous for long cache lines. For $n$ (assuming $n = 2^k, k \in N$) elements, merge sort takes $k$ iterations. In iteration $1 \leq i < k$, $\frac{n}{2^i}$ pairs of consecutive sequences (each of length $2^i$) are merged. Whenever a block is brought into cache, it contains data that is soon to be used. We further presents experimental results in Section. 4.

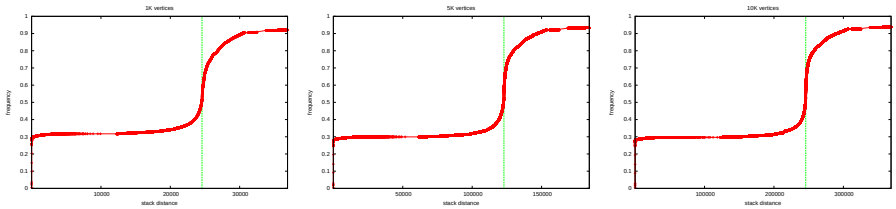


**Fig. 4.** The ratio plots for Kruskal with three inputs. The input sizes from left to right are $1K$, $5K$, and $10K$ vertices. $m = 4n$.

### 3.3 Locality of Borůvka

With Borůvka, the surges in the ratio plots are at distances in direct proportion to the input size, as shown in Fig. 5. In Fig. 5 we show the ratio plots for three different input sizes, that is, random graphs with $1K$, $5K$, and $10K$ vertices, and $m = 4n$ edges. The vertical line in each plot is $\Delta = n*3+m*4$. That is exactly the size of the input. With our adjacency list representation, for each vertex there are three data fields. Each data field takes a word of memory. For each edge incident to vertex $v$, there are two elements: $u$, the other vertex, and $w$, the edge weight. Each edge appears twice in the adjacency list. The size of the input is thus $3n+4m$.

Algorithms with such reuse behavior as shown in Fig. 5 generally scans through the data structures repeatedly for multiple runs, and each run can be considered as an algorithmic phase that may have similar or different characteristics. These algorithms generally lend themselves to parallelization as in the case of Borůvka's algorithm. In fact the Borůvka iteration (*find-min*, *connected-components* and *compact-graph*) is employed in other parallel MST algorithms (e.g., see [8, 11]).

As shown in Fig. 5, even with a fully associative cache, the cache needs to be at least of the size of the input in order to have good hit ratios. Otherwise, the hit ratio is well below 90%. The vertical line on the left of each plot ($\Delta = 4n$) crosses the curve at about *frequency*=60%. The line corresponds nicely to the size of the four auxiliary data structures used in the algorithm, that is *Min*, *Min_ind*, *D*, *Alive*. Consistently, with

a cache size of $4n$ words, the hit ratio is around 60%. In order not to contract the graph which is costly as it involves memory allocation and copying, we use the $D$ and $Min$ arrays for each vertex (and supervertex) to record the component it belongs to and the smallest weight of the adjacent edges. $Min\_ind$ records the other vertex (or supervertex) that is incident to the edge with smallest weight. The *Alive* array shows whether a vertex should be considered in the Borůvka iteration. With a cache size of $4n$, most accesses to $D$ in our implementation are cache hits. We refer interested readers to [4] for details of the algorithm.

There are two cache sizes for Borůvka that can achieve reasonable hit ratios. More specifically, one is $4n$ and the second is $3n + 4m$. The effectiveness of caching is highly dependent on the input size. In contrast to Prim and similar to Kruskal, Borůvka does not exhibit good temporal locality. What is worse is that Borůvka does not exhibit good spatial locality either, and most accesses to the arrays are irregular.

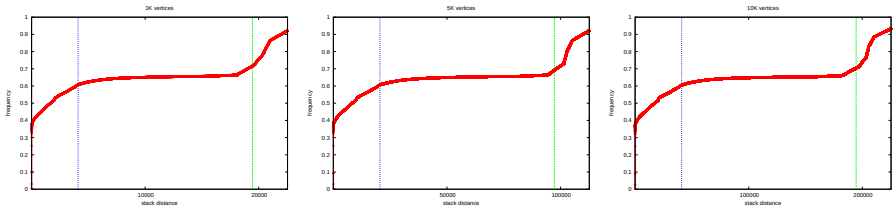


**Fig. 5.** ratio plots for Borůvka with three different inputs.

The reuse distance analysis of Borůvka suggests poor temporal locality for parallel graph algorithms (we mainly focus on PRAM algorithms since most interesting parallel graph algorithms are based on PRAM) due to the inherent phase behavior. In addition, the irregular nature of the input dictates poor spatial locality behavior.

### 3.4 Locality of Parallel MST Algorithms

In this section we consider the locality of parallel Borůvka's algorithm. The locality of parallel Borůvka's algorithm is representative for at least some stages in the more complex MST algorithms. In fact, the graft-and-shortcut approach used in parallel Borůvka's algorithm is also frequently used in other parallel graph algorithms, for this class of algorithms, we expect to see similar locality behavior.

The parallel implementations of *find-min* and *connect-components* are straightforward. We have two implementations of the *compact-graph* step. One of them contracts the graph using parallel sorting routines, while the other adopts a data structure called *flexible adjacency list* that avoids large scale sorting. We refer interested readers to [4] for details of the implementations.

Fig. 6 shows the ratio plots for two implementations of parallel Borůvka's algorithm. Again the input is a random graph with $1K$ vertices and $4K$ edges. We emulate the parallel algorithm with one thread. The locality behavior for each thread with multiple threads should be similar. The two ratio plots of Fig. 6 look roughly like the plots

in Fig. 5, and show poor locality in terms of reuse distance. At a large distance about 130,000 words, the hit ratio reaches above 80%. The range of the reuse distance is significantly larger than that of the sequential implementation. This is due to the fact in both implementations, after each iteration new instances of the graph (either fully or partially compacted) are generated and the next iteration works on the new instances. Fig. 6 partly explains why it is difficult to achieve good parallel speedup for sparse arbitrary instances on current parallel computers. The parallel algorithms have poorer locality than the sequential algorithms, and as far as we are aware of, there are no mature techniques for improving the locality behavior of parallel graph algorithms. As cache performance becomes even more crucial, the gap between theoretical results and actual performances can be increasing.

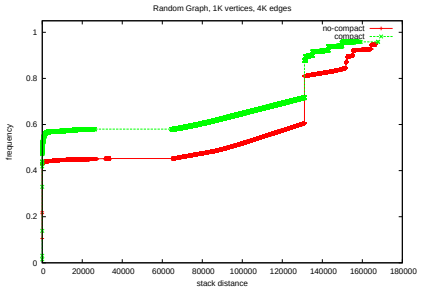


**Fig. 6.** The ratio plots for two implementations of parallel Borůvka's algorithm. The implementation labeled as *compact* compacts the graph using parallel sorting routines. The *no-compact* version uses the *flexible adjacency list* representation.

## 4 Simulation Results

We next present our experimental results with different cache configurations that support our analysis in the prior section. We run the algorithms on the RSIM simulator [19] that simulates modern processors and memory sub-system. Instead of giving pages of specifications for the processor, we use similar settings as in prior studies (e.g., see [1]). The important features include instruction-level parallelism, out-of-order scheduling, non-blocking reads and speculative execution. As we only run sequential algorithms, we do not use any of the multiprocessor features such as memory consistency protocols. In our study we use directly-mapped L1 cache and 2-way set associative L2 cache, and the input is a random graph with $1K$ vertices and $4K$ edges.

First we vary the cache line size, and measure the performance. As the cache line size increases, each transfer brings more data into cache, and the spatial locality of an algorithm becomes more important for performance.

In Fig. 7, the plot on the left shows how the performance varies with different cache line sizes. The size of the cache is kept constant ($1KB$ L1 and $4KB$ L2) in the experiments. The smallest cache line size that can be simulated is 16 bytes. With the increase

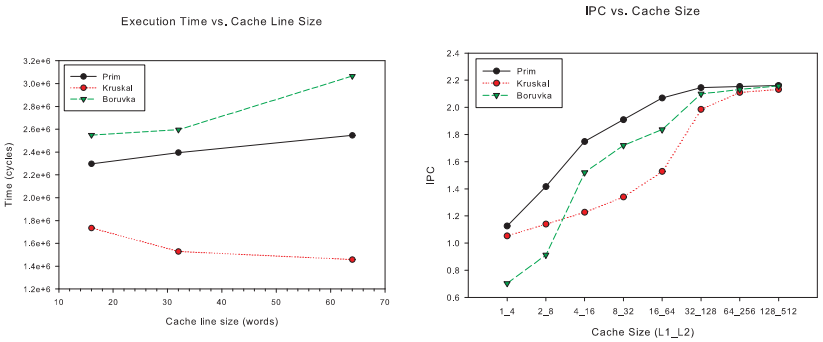


**Fig. 7.** Performance of MST algorithms with different cache configurations. For the plot on the left, we experiment with cache line sizes of 16 bytes, 32 bytes and 64 bytes. The plot on the right shows performance for different cache sizes.

of line size, the performance of both Prim and Borůvka decreases while that of Kruskal improves. The results support our analysis that both Prim and Borůvka do not have good spatial locality and is not favored by long cache lines.

The plot on the right of Fig. 7 shows the performance of the algorithms with different cache sizes, from $1KB$ L1 and $4KB$ L2 to $128KB$ L1 and $512KB$ L2. The performance, measured as instruction per cycle (IPC), improve as the cache size increases. The performance curves in this plot are correlated with the ratio plots in Fig. 2. Yet it is not straightforward to predict the performance with real cache configurations from the ratio plots. According to the ratio plot we would expect the performance curve for Prim's algorithm rise sharply at a small cache size and remain somewhat flat afterwards. This is obviously not true in the performance plot. The discrepancy is mostly due to the associativity of the cache and the cache line size. For Kruskal the IPC increases sharply at 32K bytes (L1 cache size), and the whole input (of size roughly 24K bytes) fits in L1. For Borůvka, there are two sharp increases with the performance curve. The increases correspond roughly to the sharp increases in the ratio plots.

## 5 Conclusion and Future Work

In this paper we studied the locality behavior of MST algorithms. As memory hierarchy deepens, locality is becoming even more important to the performance. We show that Prim with implicit binary heap has better temporal locality than the cache-aware implementations in our study. A significant percentage of the memory accesses incurred by the heap operation have $O(1)$ or $O(\log n)$ reuse distances. However, architectures with long cache lines impose the requirement of spatial locality for good performance, and penalize the performance of Prim with implicit binary heap. Kruskal (with non-recursive merge sort) exhibits poor temporal locality as many reuse distances are in the order of $O(n)$. Due to its good spatial locality, it runs fast on current architectures. Increasing cache line size in general improves its performance. Comparing Prim and Kruskal, it seems that good spatial locality fits better with current cache organizations.

Both the sequential and parallel implementations of Borůvka show poor temporal and spatial locality. In future work we will further investigate the locality behaviors of parallel graph algorithms. This is especially meaningful as many processors adopt multi-core designs. Our study of Borůvka's algorithm hints that poor locality might be inherent in the PRAM algorithms. In order to verify, we will need to find a metric for measuring spatial locality. On one hand, it is important to design parallel algorithms with reasonable locality behavior. On the other hand, special architectural support, for example, multi-threaded architecture, is necessary to tolerate the memory access latency for parallel algorithms. We will also investigate the impact of locality enhancing techniques such as vertex reordering on the performance of parallel algorithms. For Prim and Kruskal in our study, from the analysis of the algorithms, we do not expect to see too big a difference in the stack distance distribution. For Borůvka, however, there can be interesting findings, and we expect similar results with many other parallel algorithms.

## References


1. S.V. Adve, V.S. P, and P. Ranganathan. Recent advances in memory consistency models for hardware shared-memory systems. In *proceedings of the IEEE, special issue on distributed shared-memory*, pages 445–455, 1999.
2. G. Aloupis, P. Bose, E.D. Demaine, S. Langerman, H. Meijer, M. Overmars, and G.T. Toussaint. Computing signed permutations of polygons. In *Proc. of the 14th Canadian Conf. on Computational Geometry (CCCG)*, pages 68–71, Lethbridge, Alberta, Canada, August 2002.
3. 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.
4. 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.
5. M. Brinkhuis, G.A. Meijer, P.J. van Diest, L.T. Schuurmans, and J.P. Baak. Minimum spanning tree analysis in advanced ovarian carcinoma. *Anal. Quant. Cytol. Histol.*, 19(3):194–201, 1997.
6. C. CaΒcaval 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.
7. C. Chen and S. Morris. Visualizing evolving networks: Minimum spanning trees versus pathfinder networks. In *IEEE Symp. on Information Visualization*, Seattle, WA, October 2003. to appear.
8. R. Cole, P.N. Klein, and R.E. Tarjan. A linear-work parallel algorithm for finding minimum spanning trees. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–15, Cape May, NJ, 1994.
9. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Inc., Cambridge, MA, 1990.
10. 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.

11. D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.
12. D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Reading, MA, 1973.
13. M. Matos, B.N. Raby, J.M. Zahm, M. Polette, P. Birembaut, and N. Bonnet. Cell migration and proliferation are not discriminatory factors in the in vitro sociologic behavior of bronchial epithelial cell lines. *Cell Motility and the Cytoskeleton*, 53(1):53–65, 2002.
14. R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9:78–117, 1970.
15. S. Meguerdichian, F. Koushanfar, M. Potkonjak, and M. Srivastava. Coverage problems in wireless ad-hoc sensor networks. In *Proc. INFOCOM '01*, pages 1380–1387, Anchorage, AK, April 2001. IEEE Press.
16. G. L. Miller and V. Ramachandran. Efficient parallel ear decomposition with applications. Manuscript, UC Berkeley, MSRI, January 1986.
17. Y. Moan, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theoretical Computer Science*, 47(3):277–296, 1986.
18. B.M.E. Moret and H.D. Shapiro. An empirical assessment of algorithms for constructing a minimal spanning tree. In *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science: Computational Support for Discrete Mathematics **15***, pages 99–117. American Mathematical Society, 1994.
19. V.S. Pai, P. Ranganathan, and S.V. Adve. RSIM: an execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessor. In *Proceedings of the 3rd workshop on computer architecture education*, 1997.
20. P. Sanders. Fast priority queues for cached memory. *ACM J. Experimental Algorithmics*, 5(7), 2000. `www.jea.acm.org/2000/SandersPriority/`.
21. 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.
22. 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.
23. R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Computing*, 14(4):862–874, 1985.
24. Y.-C. Tseng, T.T.-Y. Juang, and M.-C. Du. Building a multicasting tree in a high-speed network. *IEEE Concurrency*, 6(4):57–67, 1998.
25. U. Vishkin. On efficient parallel strong orientation. *Information Processing Letters*, 20(5):235–240, 1985.
26. S.Q. Zheng, J.S. Lim, and S.S. Iyengar. Routing using implicit connection graphs. In *9th Int'l Conf. on VLSI Design: VLSI in Mobile Communication*, Bangalore, India, January 1996. IEEE Computer Society Press.
27. 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.