2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

An approach to parallelize Kruskal's algorithm using Helper Threads

Anastasios Katsigiannis, Nikos Anastopoulos, Konstantinos Nikas, and Nectarios Koziris

National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{tkats, anastop,knikas,nkoziris}@cslab.ece.ntua.gr

***Abstract*—In this paper we present a Helper Threading scheme used to parallelize efficiently Kruskal's Minimum Spanning Forest algorithm. This algorithm is known for exhibiting inherently sequential characteristics. More specifically, the strict order by which the algorithm checks the edges of a given graph is the main reason behind the lack of explicit parallelism. Our proposed scheme attempts to overcome the imposed restrictions and improve the performance of the algorithm. The results show that for a wide range of graphs of varying structure, size and density the parallelization of Kruskal's algorithm is feasible. Observed speedups reach up to 5.5 for 8 running threads, revealing the potentials of our approach.**

***Keywords*-Kruskal's Algorithm; Minimum Spanning Forest; Parallel algorithms; Helper Threads;**

I. Introduction

The widespread adaption of multicore platforms has offered the opportunity to explore new implementation techniques for many algorithms that were initially designed for uniprocessors. By devising new parallel schemes, the programmers will be able to exploit in a more efficient way the multiple hardware contexts offered in today's platforms. A category of problems among the most difficult to parallelize are the ones that exhibit inherently sequential characteristics. The discovery of the Single Source Shortest Path (SSSP) or the composition of the Minimum Spanning Forest (MSF) of a given graph fall into this category.

Kruskal's algorithm [12] is one of the most known algorithms that address the MSF problem. The strictly ordered examination of the graph's edges in order to decide whether they are part of the MSF or not, prohibits the usage of well known parallel strategies, like data partitioning. Our approach attempts to overcome the restrictions imposed by the the inherently sequential nature of the algorithm, by using a Helper Threading (HT) scheme. The evaluation reveals that using HT as an offloading technique can provide speedups up to 5.5 for a graph of 1M vertices and 20M edges for 8 threads.

The rest of the paper is organized as follows. Section II presents the basics of Kruskal's algorithm, while Section III discusses the key concept behind our parallelization scheme as well as important implementation details. Section IV describes the experimental evaluation and presents our findings. Finally, related work is discussed in Section V, while Section VI summarizes the paper and describes directions for future work.

II. The basics of Kruskal's Algorithm

Kruskal's algorithm is one of the most known algorithms for discovering the MSF of an undirected graph with real-valued weighted edges. Specifically, let $G = (V, E)$ be an undirected graph with $n = |V|$ vertices and $m = |E|$ edges, and $w : E \rightarrow \mathbf{R}$ a weight function assigning real-valued weights to the edges of G. The MSF of a given graph is an acyclic subset T of the edges that connects all the vertices that have at least one path between them and at the same time is of minimum weight. The algorithm examines each edge at an ascending order (beginning with the one with the minimum weight) and checks whether it would create a cycle if it was added to the MSF. If this is the case then the edge is discarded, otherwise it is included into the MSF.

For our study we select the asymptotically fastest implementation of Kruskal, which uses disjoint-set data structures and employs union-by-rank and path compression heuristics [7]. Each disjoint-set is used to store the vertices that belong to a single tree of the MSF at any given time of the execution. A formal representation of the algorithm is given in Algorithm 1.

At the initialization phase (lines 1–5), the algorithm creates one disjoint set for each vertex of the graph and sorts the edges into a nondecreasing order by their weight. Next, the algorithm examines whether each edge can be added to the MSF or not (line 7). Using the *find-set* operation, the algorithm locates the disjoint-set to which each vertex of the edge in question belongs. If the two sets coincide, a path that connects the two vertices exists in the MSF already and the edge must be discarded, since its inclusion would form a cycle. In the opposite case, the edge is added to the MSF and the sets are merged using the *union* operation (lines 8–9).

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.201

IEEE computer society

Algorithm 1: Kruskal's algorithm.

Input : Undirected graph $G = (V, E)$, weight function $w : E \rightarrow \mathbf{R}$
Output : Minimum Spanning Forest A

/ Initialization phase */*
1 $A = \emptyset$;
2 **foreach** $v \in V$ **do**
3   `Make-Set`(v);
4 **end**
5 sort edges of G into nondecreasing order by weight w;
/ Main body of the algorithm */*
6 **foreach** $e = (u, v) \in E$, *taken in nondecreasing order by weight* **do**
7   **if** `Find-Set`$(u) \neq$ `Find-Set`(v) **then**
8         $A = A \cup e$;
9   `Union`(u, v);
10 **end**
11 **return** A;

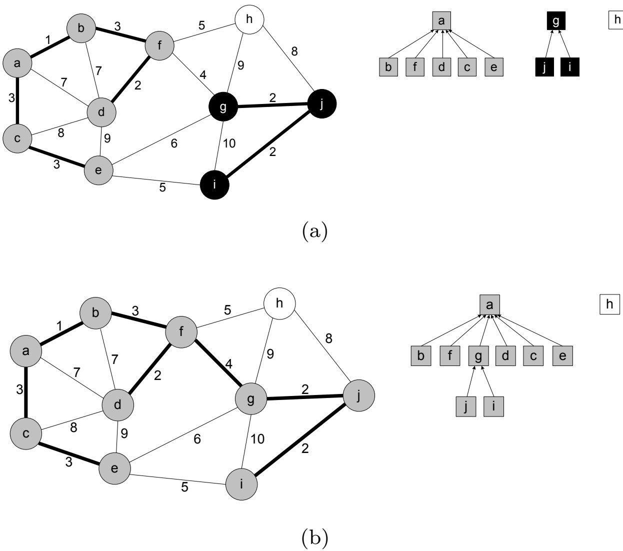

Figure 1: Example of Kruskal's algorithm execution

An example of Kruskal's execution is shown in Figure 1. Figure 1a illustrates a graph, where the first seven edges have already been examined and inserted into the MSF, which at this point consists of two separate trees. The vertices of edges (a, b), (f, d), (a, c), (b, f) and (c, e) belong to the disjoint-set $\{a, b, f, d, c, e\}$ and the vertices of the edges (j, g) and (j, i) are part of the disjoint-set $\{g, j, i\}$. To decide if the next edge, namely (f, g), is part of the MSF, we must determine if vertices f and g belong to different sets. In this case this is true, so (f, g) is inserted into the MSF and the two disjoint-sets are concatenated, as illustrated in Figure 1b.

The correct execution of the algorithm relies on checking the edges in a strictly ordered way. In the previous example, if we had examined the edge (a, d) exactly after (a, b), then we would have found that it doesn't form a cycle and would have added it to the MSF, thus creating a wrong MSF. This strict ordering makes Kruskal an inherently serial algorithm and thus difficult to parallelize.

III. Parallelizing Kruskal's algorithm

A. Discovering parallelism in Kruskal

As presented in Section II, Kruskal's algorithm is inherently serial. However it exhibits a property that can be used to extract parallelism. More specifically, if an edge that is going to be examined in a future iteration is found to already form a cycle within the MSF created up to the current iteration, then this edge can be safely discarded immediately. This property essentially allows the out of order rejection of edges.

B. Employing a Helper Threading scheme

Our proposed scheme attempts to exploit the aforementioned property. First of all, it employs a main thread that executes the regular, sequential Kruskal's algorithm and at each iteration examines the edge with the next minimum weight. At the same time, a number of helper threads run concurrently with the main one and examine edges of bigger weight, checking whether they create a cycle if added to current MSF. Whenever a cycle is discovered, the corresponding edge is marked as discarded. As these edges have been safely excluded from the MSF, the main thread needs to check only the edges that weren't rejected by the helper threads, thus performing less work compared to the sequential implementation. The more cycles found by the helper threads, the more offloading will be accomplished for the main thread.

Figure 2 illustrates the operation of the HT scheme. In this example, helper threads 1 and 2 have been assigned different sets of edges to operate on. At the end of the 7th iteration, the main thread has added to the MSF edges (a, b), (f, d), (j, g), (j, i), (a, c), (b, f) and (c, e). At this point, helper thread 1 has already discarded edges (b, d), (a, d) and (c, d) while helper thread 2 has excluded edges (d, e) and (g, i). Edges (e, g), (h, j) and (h, g) have been examined by the helper threads and since they have not been found to form cycles, their status remains undecided.

By the 10th iteration, the main thread has added (f, g) and (f, h) to the MSF, which causes the helper threads to reject edges (e, g), (h, j) and (g, i). At the end of the 10th iteration, the main thread has checked and discarded edge (e, i) and is ready to move on to the edges assigned to helper thread 1. However, by this point all the edges assigned to the helper threads have been discarded and the MSF has essentially been finalized. Therefore, the main thread had to perform less work

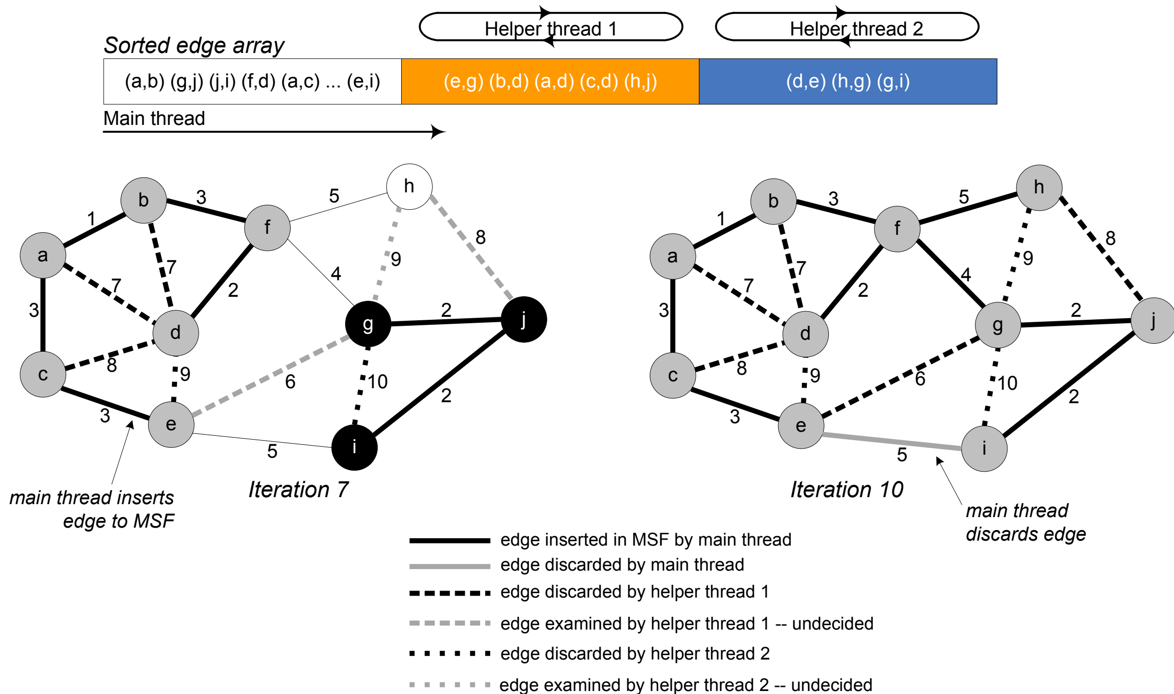

Figure 2: Example of HT scheme's execution

compared to the serial execution of the algorithm, as it examined only 10 of the 18 graph edges.

C. Implementation Details

The basic data structure of the algorithm is the `edge_array`, an array that stores the graph's edges in nondecreasing order by their weight. Moreover, our schemes requires to know at each time whether an edge has been discarded or not from the MSF. This information is stored in two separate arrays, namely `edge_color_main` and `edge_color_helper`. The main thread uses the first one to mark which edges are part of the MSF and which ones it found to form cycles. The second array is used by the helper threads to mark the edges they discover to form a cycle. Essentially, for each selected edge, the main thread consults the `edge_color_helper` array first, to decide whether it should proceed with examining the edge or ignore it if it has already been safely discarded.

To distribute work among the threads, `edge_array` is evenly divided among them. The main thread begins at the start of `edge_array` and executes Kruskal's algorithm. As the execution progresses and the MSF is expanded, the main thread enters partitions assigned to helper threads and examines the edges that have not yet been rejected. The execution finishes when the main thread reaches the end of `edge_array`.

On the other hand, the helper threads loop continuously inside their partition until they discover that the main thread entered their area. A schematic representation of this work distribution scheme is given in Figure 3. Algorithms 2 and 3 present in a formal way the code executed by the main and helper threads respectively. Note that the helper threads execute a slightly modified, read-only version of *find-set* operation (`find-set'`, line 7, Alg. 3), where all the original functionality has been maintained except for the path-compression part, which is performed only by the main thread.

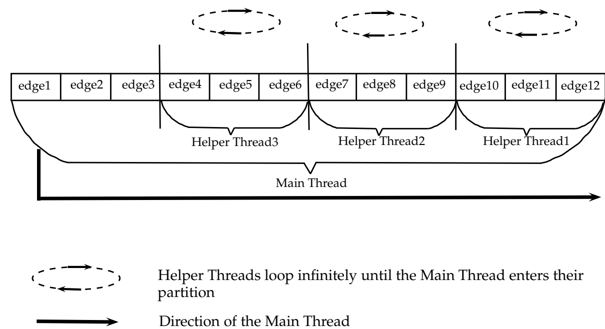

Figure 3: HT's work distribution scheme

A major advantage of our scheme is that there is no thread synchronization. First of all, the main and the helper threads perform writes on separate structures,

Algorithm 2: Main thread's code.

```
   Input  : Undirected graph G = (V, E), weight function
            w : E → R
   Output : Minimum Spanning Forest A
   /* Initialization phase */
   /* regular initializations (lines 1–5 of Algorithm 1)  */
 1 foreach e ∈ E do
 2     edge_color_main[e] = 0;
 3 end
   /* Main body */
 4 foreach e = (u, v) ∈ E, taken in nondecreasing order by
   weight do
 5     if edge_color_helper[e] ≠ CycleEdge then
 6         if Find-Set(u) ≠ Find-Set(v) then
 7             A = A ∪ (u, v);
 8             Union(u, v);
 9             edge_color_main[e] = MsfEdge;
10         else
11             edge_color_main[e] = CycleEdge;
12 end
13 return A;
```

Algorithm 3: Helper thread's H code.

```
   Input  : E_H partition of E assigned to H
   /* Initialization phase */
 1 foreach e ∈ E_H do
 2     edge_color_helper[e] = 0;
 3 end
   /* Main body */
 4 while main thread has not reached E_H do
 5     foreach e = (u, v) ∈ E_H do
 6         if edge_color_helper[e] == 0 then
 7             if Find-Set'(u) == Find-Set'(v) then
 8                 edge_color_helper[e] = CycleEdge;
 9     end
10 end
```

namely `edge_color_main` and `edge_color_helper`. At the same time, the helper threads access different parts of the shared structures `edge_color_helper` and `edge_array`. The only point where synchronization could be needed, is when the main thread consults `edge_color_helper` to check whether a helper thread has marked an edge as discarded.

Due to the semantics of the algorithm, the only pathological case arises when a helper thread discovers a cycle and marks an edge as discarded, while at the same time the main thread checks the `edge_color_helper` struct for this specific edge and misses the write. In this case, the main thread will proceed with examining itself the edge and discover the cycle, thus rejecting it from the MSF. Essentially, the algorithm is executed correctly but the scheme may not achieve maximum offloading. However, it was decided that if a synchronization mechanism was used, the overhead would outweigh the potential gains and therefore it was rejected.

D. *Parallelizing the* sort *and* union *operations*

Our HT scheme does not affect the *sort* and *union* operations. Parallel sorting is a well studied problem with known characteristics and was excluded in order to study the performance of our scheme in isolation.

On the contrary, *union* was initially a target for parallelization. However, it was rejected as, by involving more threads than the main one in the concatenation of sets and the expansion of the MSF, a synchronization mechanism, such as locks, would be necessary in order to preserve the correctness of the algorithm. This would have resulted in increased complexity with doubtful impact on the overall performance. Our decision was further justified by discovering that, based on conducted experiments with several input graphs, the *union* operation accounts only for about 10% of the execution time.

IV. Experimental evaluation

A. *Experimental Setup*

To evaluate the performance of the proposed HT scheme, we used three different multicore CPUs. The Intel Xeon X5560 ("Nehalem"), Intel Xeon X5650 ("Westmere") and Intel Xeon X7460 ("Dunnington"). Table I describes the characteristics of each platform. Figures 4, 5 and 6 present a graphical model for each one of the aforementioned platforms.

For the implementation of the HT scheme we used POSIX threads. The OS used was Linux version 2.6.30. All programs were compiled using gcc version 4.5.2 with the O3 optimization flag. Finally, in our experiments we applied a thread affinity scheme, where all the cores of a package are assigned to threads before using cores from the next package. In all the experiments Hyper-Threading was disabled.

Table I: Platforms' Characteristics

	Platforms		
	Nehalem	Westmere	Dunnington
# of packages	2	2	4
Cores/Socket	4	6	6
Threads/Core	2	2	1
CPU frequency	2.80 GHz	2.66 GHz	2.66 GHz
Chipset interface	2 × QPI 6.4GT/s	2 × QPI 6.4GT/s	FSB 1066MT/s
L1 Cache	L1D,L1I: 32KB	L1D,L1I: 32KB	L1D,L1I: 32KB
L2 Cache	256 KB, private	256 KB, private	3 MB, shared per 2 cores
L3 Cache	8 MB, shared	12 MB, shared	16 MB, shared
RAM	12 GB	48 GB	28 GB

B. *Reference graphs*

The graphs used as reference inputs vary in structure, density and size. We used the GTgraph graph generator [3] to construct graphs with $100K$ and $1M$ vertices from the following families:

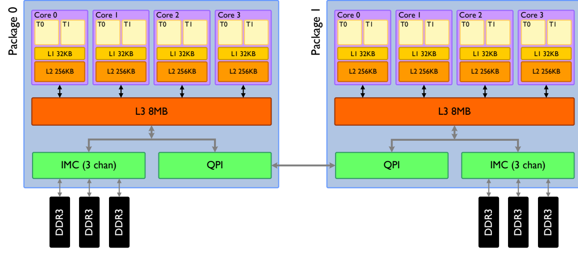

Figure 4: Nehalem platform

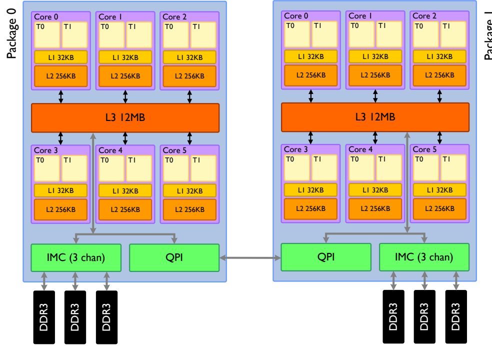

Figure 5: Westmere platform

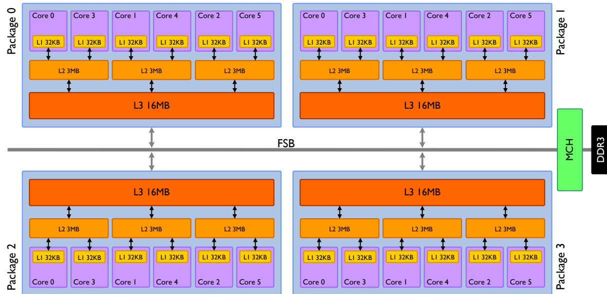

Figure 6: Dunnington platform

Random: Their m edges are constructed choosing a random pair among n vertices.

R-MAT: Constructed using the Recursive Matrix (R-MAT) graph model [4].

SSCA#2: Used in the DARPA HPCS SSCA#2 graph analysis benchmark [2].

We use the notation $N \times M$ for graphs with N vertices and M edges. For example, 1Mx5M denotes a graph with 1 million vertices and 5 million edges.

C. *Evaluation of the results*

In Figures 8, 9, 10 we present the achieved speedups of our HT scheme when run on the aforementioned platforms. The speedup is calculated as the ratio of the execution time of the serial to the parallel scheme. These

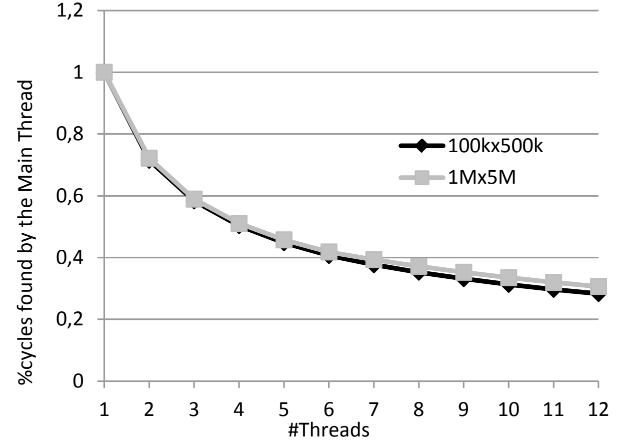

Figure 7: Main thread offloading (R-MAT graphs, Westmere platform).

results do not include the time that was spent at the sorting phase of the algorithm.

The first observation is that performance is strongly related to the density of the graphs. The maximum attainable speedup increases as density increases, reaching up to a factor of 5.5 for the R-MAT 1Mx20M graph across all architectures. This trend is expected, as a more dense graph implies more edges are creating cycles, therefore increasing the opportunity for helper threads to perform useful work.

In most cases the addition of helper threads is beneficial, leading to larger or smaller performance gains, depending on the graph density. An exception to that is the case of 100K graphs, where performance drops significantly when more than one packages are used to accommodate helper threads. At first sight, this could be attributed to insufficient work performed by the helper threads in these cases. To verify this, we measured the factor by which the main thread gets offloaded by helper threads. Figure 7 shows the percentage of cycles found by the main thread for a small and a large graph of the same density. It can be clearly seen that, as more helper threads are added, the main one consistently discovers less cycles and essentially performs less work.

More important however, is that the offloading factor is almost identical for both graphs, which contradicts our initial explanation regarding the performance drop in the case of small graphs. We believe that the fact that this offloading does not ultimately translate to performance gains can be attributed to cache behavior.

More specifically, the execution seems to be much more sensitive to cache locality for small graphs than for larger ones. In fact, the 100K graph used in Figure 7 has an overall working set of 9.4MB that fits into the last

level cache, while the 1M graph has a much larger one (94 MB). When helper threads are executed in a different package from that of the main thread, coherence misses are introduced due to read-write sharing. For small graphs this hurts the performance of the HT scheme, as the advantages of cache locality disappear. On the contrary, in the case of large graphs, the execution suffers already from poor cache locality, and therefore the addition of coherence misses has a small effect on the overall performance.

V. Related Work

Other algorithms that solve the MSF problem in parallel have been proposed in the literature. Bader and Cong [1] examined three different implementations of Borůvka's algorithm and introduced a new one that combined Borůvka's with Prim's algorithm. Kang and Bader [10] proposed another combination of the aforementioned algorithms, which they implemented using a Software Transactional Memory model. Our approach is based on Kruskal's algorithm and, according to our knowledge, this is the first attempt at efficiently parallelizing this specific algorithm. Moreover, in contrast to [1] and [10], our scheme requires few modifications on the original serial code of the selected algorithm and avoids the need for any kind of blocking or non-blocking synchronization.

Alternative usages of helper threads have been proposed in literature, mostly targeting at performance optimizations at the micro-architecture level. The most prominent example is that of prefetching helper threads [5], [6], [13]. Other scenarios include helper threads that optimize branch prediction [17], perform dynamic code optimizations [16] or even help reduce energy consumption [8]. On the contrary, in this work helper threads perform application-level optimizations. They execute in parallel with the main thread in a non-intrusive fashion, both in terms of algorithm semantics, as no significant changes are required, and main thread's progress, as no obstruction is introduced. A similar approach was followed in a prior work where we used helper threads to parallelize Dijkstra's algorithm [14]. In that case, we additionally used Transactional Memory to guarantee that helper threads would not change the algorithm's semantics.

For inherently serial, hard to parallelize applications, the Thread Level Speculation model has been proposed in the literature as an alternate parallelization approach [9], [11], [15]. The drawback of most of these research efforts is that they require support from specialized hardware, which is not available even in today's processors. Our scheme, on the other hand, relies solely on specific properties of the algorithm and thus can be applied directly on commodity multicore platforms.

VI. Conclusions - Future Work

In this paper we presented a Helper Threading scheme for parallelizing Kruskal's algorithm. The implementation is a synchronization-free one, that employs one main thread, which essentially executes the serial algorithm, and several helper threads, which run in parallel and offload the work of the main thread. The proposed scheme achieves notable speedups for a wide range of graphs when executed on various multicore platforms.

As future work, we will investigate in detail how exactly the underlying architecture affects our scheme and confirm whether cache coherence impacts its performance. In that case, we will study alternative implementations based on decentralized structures in an attempt to exploit cache locality efficiently. Additionally, we aim to explore alternative work distribution schemes in order to increase the efficiency of the helper threads and the amount of useful work they accomplish. Finally, we plan to investigate the applicability of the HT scheme to other hard to parallelize applications.

References

[1] D.A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *Journal of Parallel and Distributed Computing*, 2006.

[2] D.A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing, HiPC'05*, 2005.

[3] D.A. Bader and K. Madduri. Gtgraph: A suite of synthetic graph generators. 2006. http://www.cse.psu.edu/~madduri/software/GTgraph/index.html.

[4] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 4th International Conference on Data Mining, ICDM'04*, 2004.

[5] R. Chappell, J. Stark, S. P. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th annual International Symposium on Computer Architecture, ISCA '99*, 1999.

[6] J.D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J.P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA'01*, 2001.

[7] T. Cormen, C. Stein, R. Rivest, and C. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[8] Y. Ding, Kandemir Y., P Raghavan, and M Irwin. A helper thread based EDP reduction scheme for adapting application execution in CMPs. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed, IPDPS'08*, 2008.

[9] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-VIII*, 1998.

[10] S. Kang and D.A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP '09*, 2009.

[11] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *Journal, IEEE Transactions on Computers*, 1999.

[12] J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *In Proceedings of the American Mathematical Society*, 1956.

[13] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA'01*, 2001.

[14] K. Nikas, N. Anastopoulos, G. Goumas, and N. Koziris. Employing transactional memory and helper threads to speedup Dijkstra's algorithm. In *Proceedings of the International Conference on Parallel Processing, ICPP '09*, 2009.

[15] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture, HPCA'98*, 1998.

[16] W. Zhang, B. Calder, and D. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, PACT'05*, 2005.

[17] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA'01*, 2001.

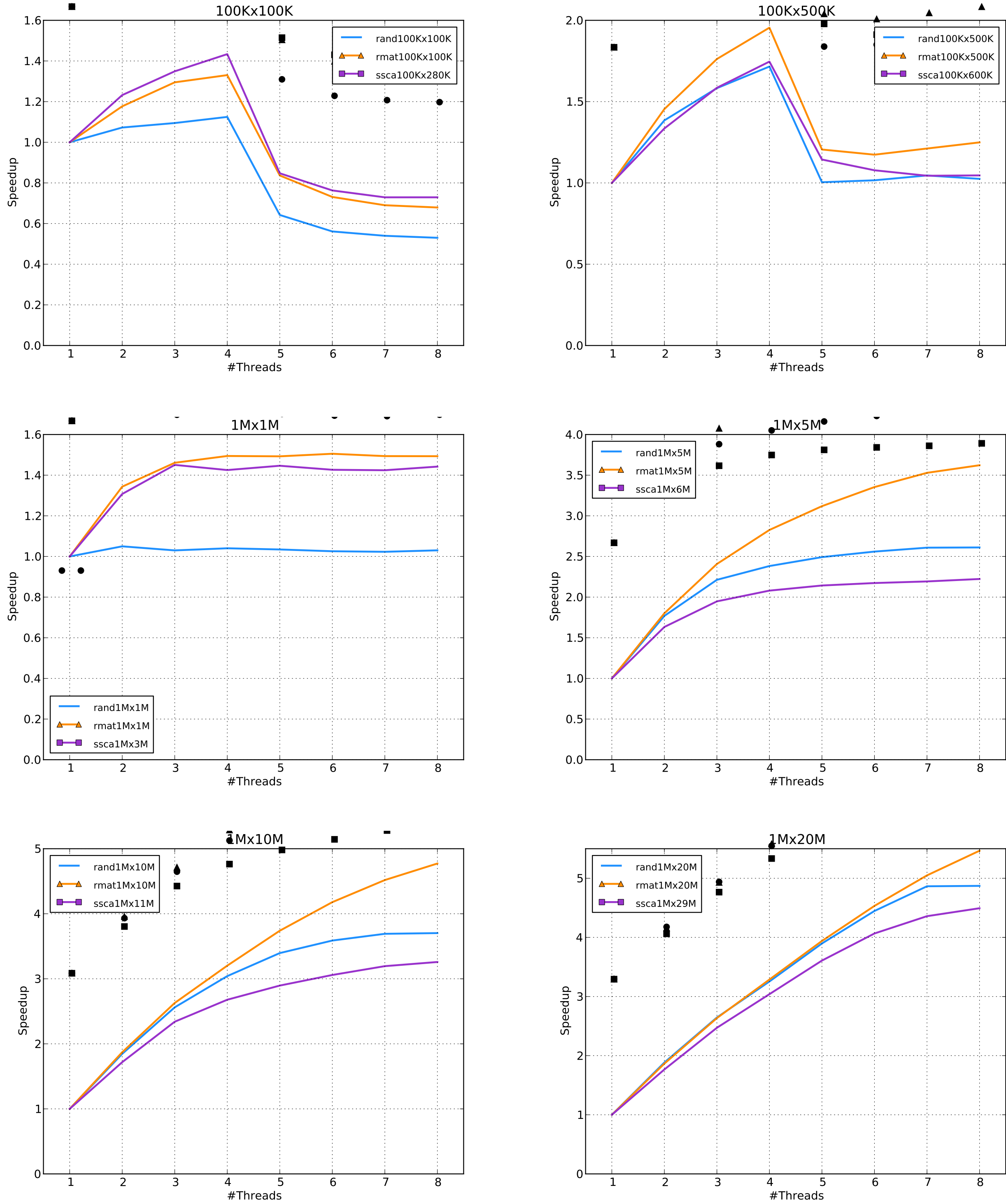

Figure 8: Speedups achieved on the Nehalem platform

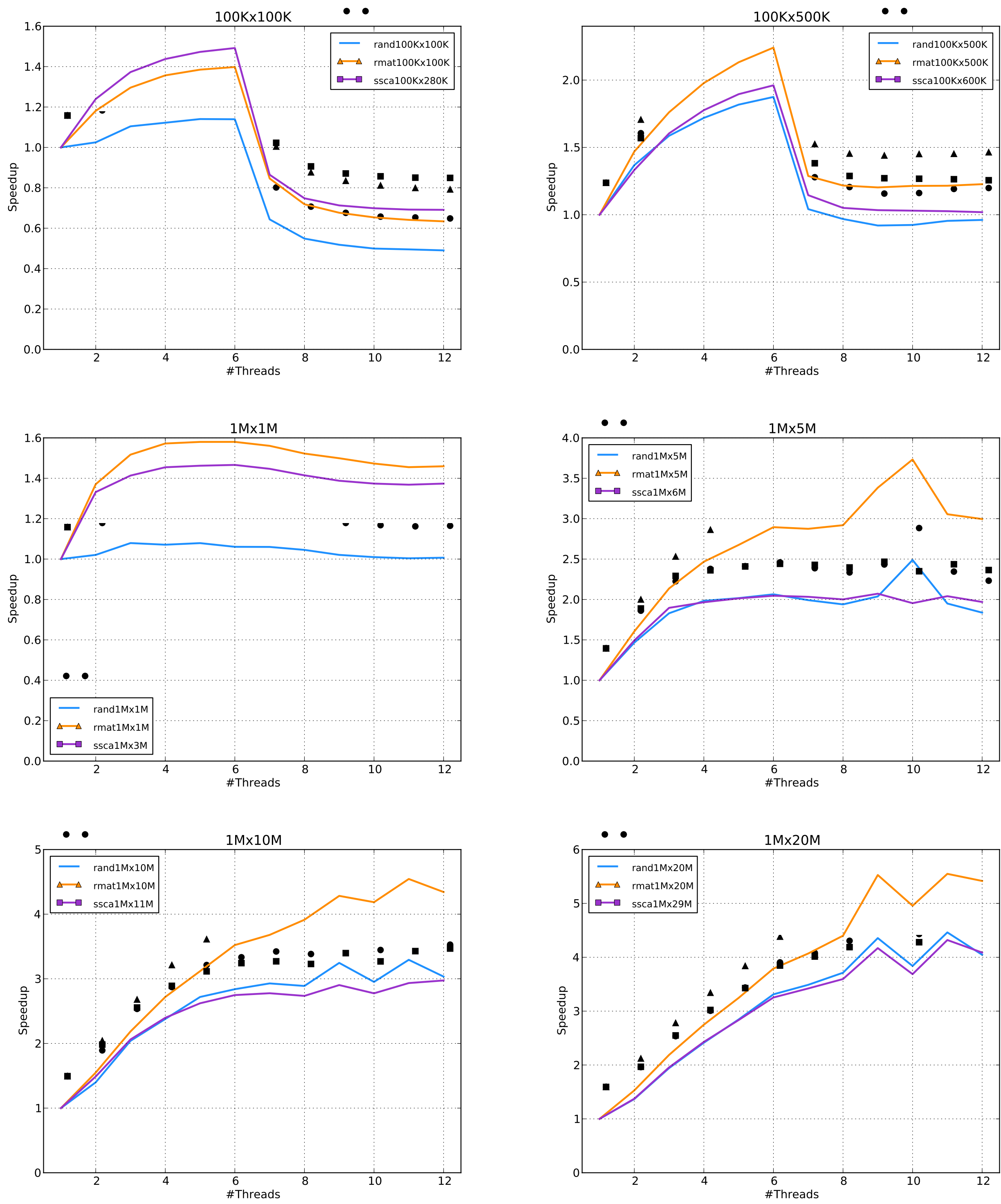

Figure 9: Speedups achieved on the Westmere platform

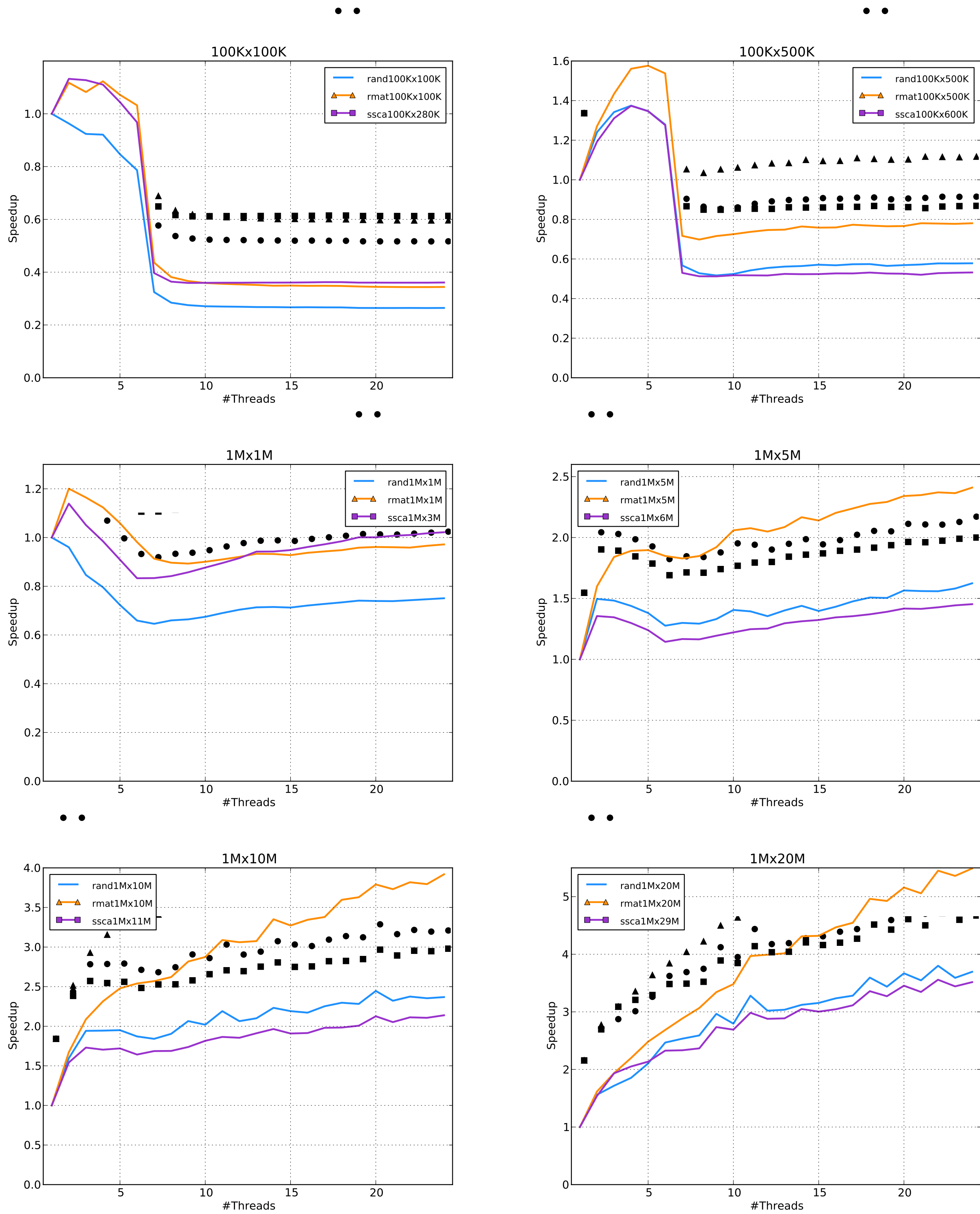

Figure 10: Speedups achieved on the Dunnington platform