Implementing and Evaluating Multithreaded Triad Census Algorithms on the Cray XMT

George Chin Jr.[1], Andres Marquez[1], Sutanay Choudhury[1], and Kristyn Maschhoff[2]
[1]High-Performance Computing
Pacific Northwest National Laboratory
{George.Chin, Andres.Marquez, Sutanay.Choudhury}@pnl.gov
[2]Cray, Inc. kristyn@cray.com

Abstract

Commonly represented as directed graphs, social networks depict relationships and behaviors among social entities such as people, groups, and organizations. Social network analysis denotes a class of mathematical and statistical methods designed to study and measure social networks. Beyond sociology, social network analysis methods are being applied to other types of data in other domains such as bioinformatics, computer networks, national security, and economics. For particular problems, the size of a social network can grow to millions of nodes and tens of millions of edges or more. In such cases, researchers could benefit from the application of social network analysis algorithms on high-performance architectures and systems.

The Cray XMT is a third generation multithreaded system based on the Cray XT-3/4 platform. Like most other multithreaded architectures, the Cray XMT is designed to tolerate memory access latencies by switching context between threads. The processors maintain multiple threads of execution and utilize hardware-based context switching to overlap the memory latency incurred by any thread with the computations from other threads. Due to its memory latency tolerance, the Cray XMT has the potential of significantly improving the execution speed of irregular data-intensive applications such as those found in social network analysis.

In this paper, we describe our experiences in developing and optimizing three implementations of a social network analysis method known as triadic analysis to execute on the Cray XMT. The three implementations possess different execution complexities, qualities, and characteristics. We evaluate how the various attributes of the codes affect their performance on the Cray XMT. We also explore the effects of different compiler options and execution strategies on the different triadic analysis implementations and identify general XMT programming issues and lessons learned.

1 INTRODUCTION

1.1 Cray XMT

The Cray XMT system is a third generation multithreaded system from Cray. The XMT infrastructure is based on the Cray XT-3/4 platform, including its high-speed interconnect and network 3D-Torus topology, as well as service and I/O nodes. The compute nodes of the XMT utilize a configuration based on 4 multithreaded processors – called Threadstorms -- instead of 4 AMD Opteron processors. Each Threadstorm maintains 128 hardware threads (streams) and their associated contexts. Assuming all data- and resource-dependencies are met, a stream can be scheduled for instruction issue in a single cycle. Each instruction can hold up to three operations, comprising a control, arithmetic and memory operation. The XMT system enables the execution of applications built entirely for the Threadstorm processors, in a similar manner as the previous generation MTA-2 did, as well as the execution of hybrid applications, in which portions of the application execute on the Threadstorm processors and por-

tions execute on mainstream AMD Opteron processors. The two types of processing elements coordinate their execution and exchange data through a high-performance communications library.

Our large scale-out experimental runs were made on a 128 processor, 1 TB shared memory, XMT system in Cray's development lab (in Seattle, Washington) using Threadstorm 3.0.X pre-production processors that run at 500MHz. Each processor is attached over HT to 8GB DDR1 memory. Network capability between the 32 compute blades is provided by a 3D-Torus Seastar-2 interconnect that exhibits a round trip latency of 1.8us. For code development and testing purposes we used a smaller 16 processor, 128GB shared memory system located at Pacific Northwest National Laboratory in Richland, Washington. This system is equipped with 500MHz Threadstorm 2.X processors. The smaller system is configured as a 2D-Torus.

1.2 Graph Analysis

Graph algorithms fall into an important class of applications that are becoming memory-bound due to the increasing gap between memory and processor speeds. Their performance is determined by the speed of the memory subsystem (the processor will spend most of its time waiting for data to arrive from memory without any useful work to execute). This class of applications exhibit irregular memory access pattern that cannot be predicted statically at programming or compile time. Dynamic latency reduction techniques that exploit spatial locality are limited by scant discovery of the application's sparse data set shape and the data structures's dynamic traversal pattern. In addition, large data structure sizes will render temporal locality techniques mostly ineffective. Applications that fall into this class tend to focus on deriving scientific knowledge from vast repositories of empirical data. As time progresses, their data structures will change and as knowledge is accumulated, their traversal might change as well.

At a varying degree, multithreaded architectures do not rely on exploiting reference locality and instead leverage performance gains from latency hiding techniques in the form of concurrent data traversal. Maintaining massive concurrency is costly in terms of scheduling and synchronization overhead and therefore these architectures try to strike a balance between sequential and concurrent execution. The XMT represents one multithreading extreme, similar to pure data-flows machines, by providing low cost scheduling for thousands of concurrent actors in conjunction with low cost fine-grained synchronization capabilities. The XMT's peculiar fine grained thread management techniques make the machine an ideal candidate to process this class of applications.

Various graph applications have been implemented and evaluated on Cray MTA-2 and XMT systems including general graph theory [1], power system state estimation [2], partial dimension trees [3], and Boolean satisfiability solvers [4]. The research described in this paper will add to this evolving body of research.

1.3 Social Networks and Triadic Analysis

Applied widely in social and behavioral sciences, social network analysis [5] encompasses mathematical and statistical methods for studying and measuring social networks. A social network describes relationships among social entities and is often conveyed as a directed graph that shows ties or relationships among people, groups, and/or organizations. The shape and properties of a social network helps determine the network's usefulness to a person, group, or organization. For example, a dense or tight network may illustrate close social and/or working relationships among members, but may also indicate an environment where the generation of new ideas and views are limited due to the effects of groupthink and peer pressure.

Triadic methods in social network analysis are statistical methods focused on the concept of a *triad*. A triad is a subgraph of size 3 consisting of three actors and all the directed relationships among them. Overall, a directed graph has exactly $n(n-1)(n-2)/6$ triads, where n = number of nodes. Triadic methods are considered local methods in the sense they separately examine the properties of subsets of actors as opposed to global methods that simultaneously examine the properties of all actors in the social network.

A triad had sixty-four possible states based on the existence of directed edges among the three actors. Triad states indicate important properties about the triad. For example, the triad states shown in Figure 1 exhibit properties of reciprocity, transitivity, and intransitivity. As shown in Figure 2, we can build a *triad census* by capturing the frequencies in which the triads of a network fall into one of the 64 possible triad states. We may condense a 64-element triad census down to a 16-element triad census by considering isomorphic cases, where certain triad types are structurally-equivalent and may directly map onto one another.

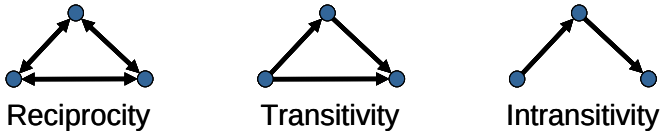

Figure 1. Triads with specific graph properties.

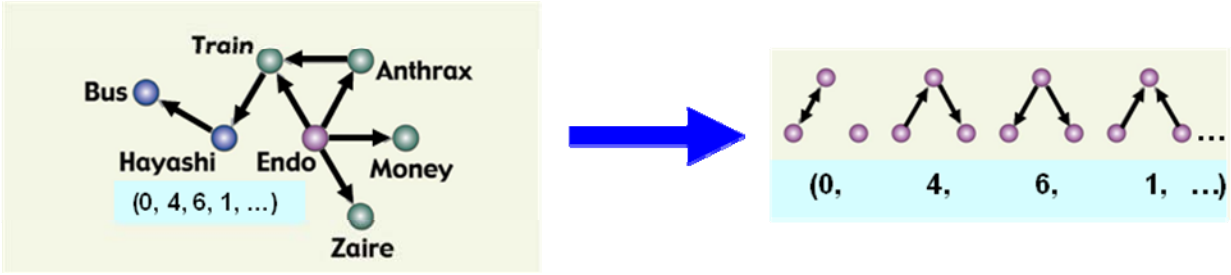

Figure 2. Creation of a triad census.

A triad census may be statistically compared to the triad census of a directed graph with a random distribution to examine which triad states and their associated properties are more or less prominent in the social network under analysis. For example, if we believe that a particular social network should exhibit an overall transitive property, then the frequencies of triads in transitive triad states should be greater than that of a randomly distributed graph. Through triadic methods, we analyze the overall structural properties of a social network based on local views into that network.

Social network and triadic analysis methods represent compelling applications for the Cray XMT. Such methods may be applied to very large networks or graphs, which are data representations that the XMT architecture is well-suited to support as previously described. Furthermore, these analysis methods have wide applicability across many fields and disciplines, and their migration to high-performance systems can only benefit scientists in allowing them to attack and solve larger networks and scientific problems.

Triadic analysis has been applied in a variety of areas including the study of organizations [6, 7, 8], Internet network traffic [9], email traffic [10], online communities [11], international trade [12], and intelligence analysis [13]. Like other social network analysis methods, triadic analysis examines and

reveals properties about the general structure of networks, and thus, should have wide applicability to other kinds of network analyses such as in biology and electric power grids.

2 TRIADIC ANALYSIS ALGORITHMS

In terms of published triadic analysis algorithms, two are generally well-known. Moody [14] developed a triad census algorithm with a computational complexity of $O(n^2)$, where n is the number of nodes. Batagelj and Mrvar [9] offer a triad census algorithm with a subquadratic computational complexity for sparse networks. In our social network research efforts, we have adapted from Batagelj and Mrvar's algorithm as well as developed many of our own.

2.1 Brute Force

A brute force triad census approach involves iterating through and examining every three possible combination of nodes in the network as shown in Figure 3. In this algorithm, the input $G = (V, E)$ is a directed graph, where V is the set of vertices and $E \subseteq V \times V$ is the set of directed edges. The output is the triad census stored in a 16-element *Census* array. The 16 array elements conform to the 16 possible isomorphic triad states. Each *Census* array element acts as a counter that sums up the number of triads in graph G found in the associated state.

```
        INPUT: G = (V, E)
        OUTPUT: Census array with frequencies of triadic types

1       for i := 1 to 16 do Census[i] := 0;    \\ initialize census
2       for each u ∈ V do begin
2.1         for each v ∈ V do if v < u then begin
2.1.1           for each w ∈ V do if w < v then begin
2.1.1.1             TriType := IsoTricode(u,v,w);
2.1.1.2             Census[TriType] := Census[TriType] + 1;
                end;
            end;
        end;
```

Figure 3. Brute force subquadratic triad census algorithm.

In step 1, we initialize the *Census* array. In steps 2, 2.1, and 2.1.1, we iterate through the nodes of V in three nested loops to assign the three nodes of a triad. To ensure that we do not count the same triad multiple times (e.g., (u, v, w) is equivalent to (w, u, v)), we canonically select the triads such that nodes $u > v > w$ is always true.

Once we have selected the three nodes u, v, and w of a triad, we call the *IsoTricode* function in step 2.1.1.1 to identify the isomorphic triad state. In step 2.1.1.2, the triad state is then used to index into the *Census* array and to increment the counter for the corresponding triad state.

The brute force triadic analysis algorithm has a computational complexity of $O(n^3)$ for all networks.

2.2 Subquadratic

We implemented a variation of Batagelj and Mrvar's algorithm as presented in Figure 4. In the algorithm, the inputs include a directed graph $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of directed edges, and an array of neighbor lists N, which may be indexed by a node number. The output *Census* array is the triad census. Given nodes u and v, the relation uEv is true should a directed edge exist from u to v in E or $\{u,v\} \in E$. The relation $u\hat{E}v$ is true should u be a neighbor of v.

```
          INPUT: G = (V, E), N – array of neighbor lists
          OUTPUT: Census array with frequencies of triadic types

1         for i := 1 to 16 do Census[i] := 0;    \\ initialize census
2         for each u ∈ V do begin
2.1           for each v ∈ N[u] do if u < v then begin
2.1.1             S := N[u] ∪ N[v];
2.1.2             if uEv ∧ vEu then TriType := 3 else TriType := 2;
2.1.3             Census[TriType] := Census[TriType] + n − |S| − 2;
2.1.4             for each w ∈ S do if v < w ∨ (u < w ∧ w < v ∧ ¬uÊw) then begin
2.1.4.1               TriType := IsoTricode(u,v,w);
2.1.4.2               Census[TriType] := Census[TriType] + 1;
                  end;
              end;
          end;
3         sum := 0;
4         for i := 2 to 16 do sum := sum + Census[i];
5         Census[1] := (1/6)n(n − 1)(n − 2) − sum;
```

Figure 4. Modified Batagelj and Mrvar's subquadratic triad census algorithm.

The algorithm works by following existing edges in the network. In step 2, u is assigned to every vertex in V. In step 2.1, v is assigned to every neighbor of u that has a smaller value. In step 2.1.4, w is assigned to each node of S, which is the union of the neighbors of u and v. u, v, and w make up the nodes of the triad that is currently being processed. In the inner processing of the algorithm, u will always have the smallest value among u, v, and w.

The algorithm computes three different types of triads: null, dyadic, and complete. D*yadic* triads have edges between two of three vertices. Given a pair of connected nodes, we can compute the number of dyadic triads arising from the connected pair as $n - |S| - 2$ as shown in step 2.1.3. If a third node connects to either node of the connected pair, we then have a *connected* triad, where each node of the triad is connected to at least one edge. In step 2.1.4, we examine every node in S as the possible third node to the current triad. Here, we wish to avoid counting the same three nodes through different iterations of the code by only counting the canonical selection from (u, v, w) and (u, w, v). If $u < w < v$ and $u\hat{E}w$, then (u, w, v) had already been considered in the algorithm. However, if $\neg u\hat{E}w$, then (u, w, v) is the canonical selection. In step 2.1.4.1, given the nodes of a connected triad, the *IsoTricode* function identifies the triad's isomorphic state, which may then be used to index into the *Census* array. In step 5, the number of *null* triads is computed as $(1/6)n(n - 1)(n - 2) - sum$, which is the total number of possible triads minus the number of triads with at least one edge.

For sparse graphs, the complexity of the algorithm is O(k(n)*n), where k(n) << n. For complete graphs, the complexity is $O(n^3)$.

2.3 Parallel Tasks

For the parallel tasks triad census algorithm, we use our knowledge of the logic of the subquadratic algorithm to load balance the execution of the triad census computation. In examining the subquadratic algorithm in Figure 4, we see that the amount of processing that occurs within the second nested loop at step 2.1 is variable depending on the size of *S*. As shown in Figure 5, to load balance, we can construct task queues containing node pairs that identify two of the three nodes of a triad. The two nodes *u* and *v* are assigned in steps 3 and 3.1 in the same way the subquadratic algorithm assigns the first two nodes of a triad (steps 2 and 2.1 in Figure 4). In step 3.1.2, we maintain a counter that sums the sizes of combined neighbor sets we have encountered for the current queue *D[i]*. We continue to add node pairs to the task queue until a certain *MaxNeighborSetSize* is reached in step 3.1.3, upon which a new task queue is started.

```
        INPUT: G = (V, E), N – array of neighbor lists
         OUTPUT: D – array of task queues

1       i := 1;
2       counter := 0;
3       for each u ∈ V do
3.1         for each v ∈ N[u] do if u < v then begin
3.1.1           D[i] := D[i] ∪ (u, v);
3.1.2           counter := counter + |N[u]| + |N[v]|;
3.1.3           if counter > MaxNeighborSetSize then begin
3.1.3.1             i := i + 1;
3.1.3.2             counter := 0;
                end;
            end;
        end;
```

Figure 5. Task queue generation in parallel tasks triadic census algorithm.

As shown in Figure 6, the triad census portion of the parallel tasks algorithm loops through the task elements of each queue in steps 2 and 2.1 to pull out node pairs for processing. Given the first two nodes *u* and *v* of a triad, the rest of the code follows the same logic as the subquadratic algorithm to identify the third node and to compute the census elements.

The computational complexity of the parallel tasks algorithm should be the same as the subquadratic algorithm, which is O(k(n)*n), where k(n) << n, for sparse graphs and $O(n^3)$ for complete graphs. This assumes that the number of task queues is small enough such that no task queues are empty.

3 XMT Programming Features

A salient characteristic of the MTA architecture and its latest incarnation, the XMT, is the underlying programming model. Although the programmer is required to expose parallelism to the compiler in the form of loop parallelism and/or task parallelism (futures), there is no requirement to reason about locality. Given enough exploited parallelism, the programming model abstraction gives the illusion of single unit instruction latency. This is in contrast to other noteworthy programming models such as OpenMP that provide semantically similar parallelism constructs, yet do not make any latency hiding guarantees.

```
        INPUT: G = (V, E), N – array of neighbor lists, D – array of task queues
        OUTPUT: Census array with frequencies of triadic types
1       for i := 1 to 16 do Census[i] := 0;
2       for each T ∈ D do
2.1         for each (u, v) ∈ T do
2.1.1           S := N[u] ∪ N[v];
2.1.2           if uEv ∧ vEu then TriType := 3 else TriType := 2;
2.1.3           Census[TriType] := Census[TriType] + n − |S| − 2;
2.1.4           for each w ∈ S do if v < w ∨ (u < w ∧ w < v ∧ ¬uÊw) then begin
2.1.4.1             TriType := IsoTricode(u,v,w);
2.1.4.2             Census[TriType] := Census[TriType] + 1;
                end;
            end;
        end;
3       sum := 0;
4       for i := 2 to 16 do sum := sum + Census[i];
5       Census[1] := (1/6)n(n − 1)(n − 2) − sum;
```

Figure 6. Main part of parallel tasks triadic census algorithm.

In practice the "unit latency" contract can break down in situations that are either application or machine driven: The former would manifest itself as an application's lack of strong scaling capabilities. The latter might be observed in limit cases by oversubscribing the network or memory. The XMT programming model's characteristics are underwritten by a strong compiler platform that is able to extract automatically parallelism from well-formed loops. Hence, a major XMT coding task consists in generating such loops. Explicit parallelism is supported by task-parallelism constructs (futures) and several intrinsics that support fine grain synchronization in hardware. The compiler handles inter procedural analysis and is capable of detecting and rewriting linear recurrences as well as reductions.

3.1 General Code Optimization

In optimizing the triad census codes to run on the Cray XMT, we generally programmed for implicit parallelism [15] using standard C language constructs and relied on the compiler to automatically parallelize loops in the code. We iterated on the development and optimization of the codes by repeatedly analyzing the codes using Cray's Compile Analysis (Canal) [16] tool to identify and address dependencies and parallelization issues.

One common modification we made to the original sequential codes was to replace linked list data structures that are inherent to graphs with compact data structures. With a compact data structure, we allocate a large chunk of memory upfront and use array indices for random access into the data. This allows the program to access the data through inductive loops, for which the compiler may identify the number of iterations a loop contains before the loop is entered. A compact data structure also replaces a large number of dynamic memory allocations, which may be computationally expensive.

In our codes, we removed multiple exit conditions from *for* loops, which the compiler had difficulty parallelizing. We also consistently used the *int_fetch_add* generic function to synchronize updates to data without using locks. This function accesses the underlying atomic *int_fetch_add* machine operation. We also made of habit of "inlining" functions, so that we could better diagnose and optimize code near function calls.

For fast I/O, we utilized the Lightweight User Communication (LUC) library [17], which gave us access to Linux service nodes to perform faster sequential I/O operations than the XMT compute nodes. This provided significant reduction in the time required to load large input datafiles.

4 Performance Evaluation and Results

Our performance evaluation focuses specifically on the triad census generation portions of the triadic analysis codes. Each implementation, however, requires a certain level of preprocessing of the input data prior to triad census generation. For the brute force algorithm, the network is directly read in from a file in a specific graph format. The subquadratic algorithm requires as input both a network and neighbor lists for each of the nodes. In this case, the subquadratic implementation reads a file containing node adjacency lists, which are used to generate both the network and the neighbor lists. The task parallel algorithm requires as input the network, the neighbor lists, and the task queues. Like the subquadratic implementation, the parallel tasks implementation also reads in a file containing adjacency lists to construct the network and neighbor lists, but then generates the task queues as described back in Figure 5.

To focus on the triad census generation portions of the algorithms is reasonable provided that the triad census generation is more computationally-intensive than the preprocessing stages. We have optimized the preprocessing portions of the brute force and subquadratic implementations such that they execute much faster than the triad census computation. Further optimization, however, is required for the queue generation portion of the parallel tasks implementation, which is not parallelizable in its current form because the queues are sequentially filled one at a time. We are exploring other strategies to filling the queues such as using hash maps or presorting the task elements going into the queues and expect that we can reduce the execution time of the preprocessing stage to be much smaller than that of the triad census computation.

4.1 Triad Census Algorithm Comparison on 16-processor XMT

In our first evaluation, we examined the performance of the three triad census algorithms on a moderate-sized sparse random graph consisting of 10,000 nodes and 100,000 edges for different numbers of processors on the 16-processor XMT. Through testing, we found that the parallel tasks algorithm executed most efficiently with 1,000 task queues for the specified network and configured the parallel tasks code accordingly. As shown in the performance results of Table 1, the brute force algorithm required hours of execution to complete compared to the subquadratic and parallel tasks algorithms, which completed in seconds. The higher computational complexity of the brute force algorithm significantly hampered its performance compared to the other triad census algorithms.

Table 1. Execution times (in seconds) of triad census algorithms processing a 10,000-node, 10,000-edge sparse random graph. Execution is conducted on a 16-processor Cray XMT.

Triadic Analysis Algorithms - Execution Time (s) (10,000-node, 100,000-edge Random Graph)			
Processor Count	Brute Force	Subquadratic	Parallel Tasks
1	154239.83	3.21	6.22
2	85182.29	1.71	4.66
4	47168.17	0.93	4.01
8	25982.44	0.66	3.69
12	16948.09	0.48	3.40
16	13086.49	0.36	3.23

With respect to speedup, all three algorithms exhibited linear speedup rates up to 16 processors, but the brute algorithm was the most efficient, presumably due to its relative simplicity and tight production of available work as shown in Figure 7. Total memory utilization for the brute force algorithm was about 1.5 GB, while maximum CPU utilization stood at about 46%. We did not collect memory and CPU utilization data for the subquadratic and parallel tasks algorithms, since they executed so quickly.

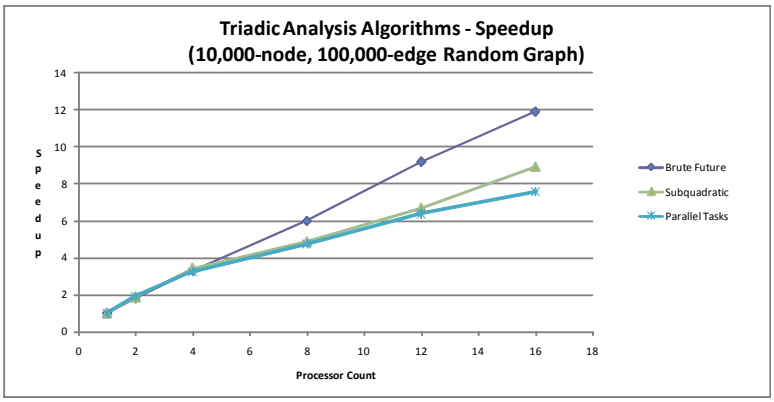

Figure 7. Speedup rates of triad census algorithms processing a 10,000-node, 10,000-edge sparse random graph. Execution is conducted on a 16-processor Cray XMT.

Next, we ran the three triad census algorithms on a much larger network consisting of 3.8 million nodes and 16.5 million edges. The network links US patents granted between 1963 to 1999 to patent citations made between 1975 and 1999. With this network, the brute force algorithm did not complete after 96 hours of dedicated execution on the 16-processor XMT. We configured the parallel tasks algorithm to use 1.6 million queues, which we found to be optimal by testing different queue counts. As shown in Figure 8a, the subquadratic algorithm executed faster than the parallel tasks algorithm on the patents network. Both these algorithms showed comparable linear scaling as shown in Figure 8b. In terms of utilization rates, the subquadratic algorithm consumed up to 12.6 GB of memory while the parallel tasks algorithm consumed up to 13.3 GB, presumably requiring a little more memory to store and manage the task queues. Processor utilization was comparable with the subquadratic algorithm achieving up to 66% CPU utilization and the parallel tasks algorithm reaching 64%. From our experiences with the Cray XMT, these CPU utilization rates are extremely high. Most graph applications that we observed in the past had achieved around 30% CPU utilization.

The Cray XMT programming environment supports a construct known as futures, which designates a section of code that may be executed by a newly created thread. The new thread runs concurrently with other threads of a program. Code depending on values computed by a future will be suspended until that future finishes executing. We may direct the compiler to schedule and manage each iteration of a loop as a future by issuing the following pragma statement immediately above the loop in the code.

```
#pragma mta loop future
```

In the case of a nested loop, the pragma statement would be placed above the outer loop.

We developed loop future versions of both the subquadratic and parallel tasks algorithms and ran them with the patents network. As shown in Figure 9a, both loop future implementations executed faster than the original implicit parallelism implementations. Overall, the loop future parallel tasks imple-

mentation outperformed the loop future subquadratic implementation and showed a significant performance improvement over the implicit parallelism parallel tasks implementation. Looking at Figure 9b, the loop future parallel tasks implementation also exhibited the best linear scaling among the implicit parallelism and loop future versions of the subquadratic and parallel tasks triad census algorithms, while the loop future subquadratic implementation exhibited the worst scaling.

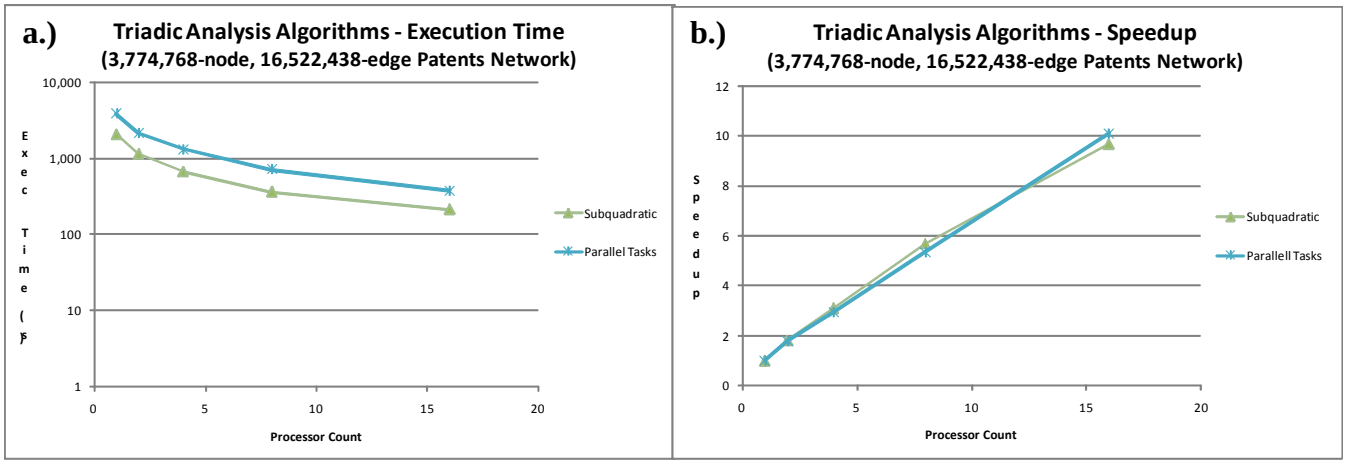

Figure 8. a.) Execution times and b.) speedup rates of subquadratic and parallel tasks triad census algorithms processing a 3,774,768-node, 16,522,438-edge patents network. Execution is conducted on a 16-processor Cray XMT.

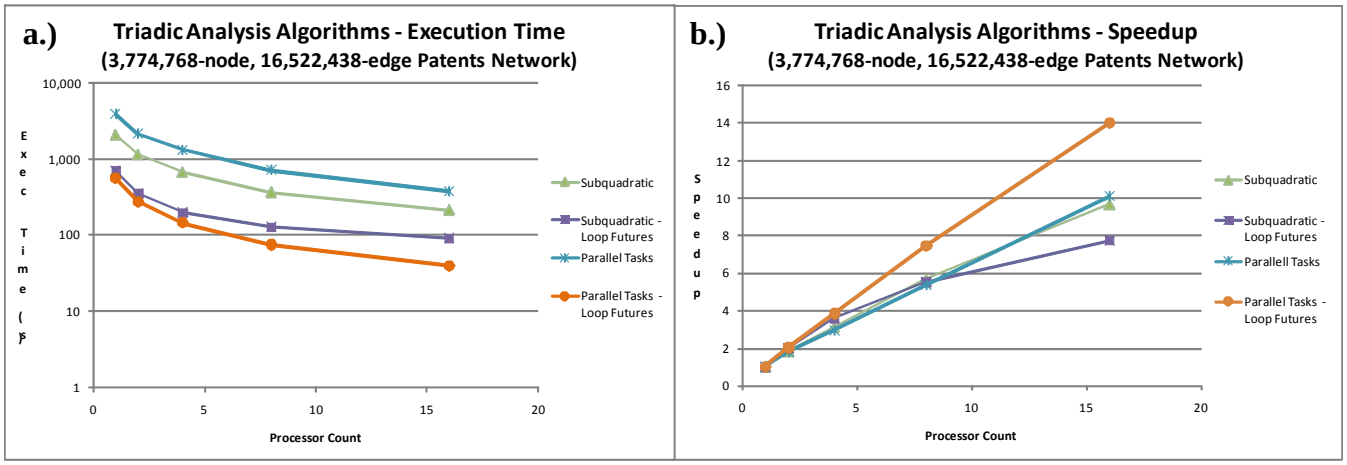

Figure 9. a.) Execution times and b.) speedup rates of implicit parallelism and loop future versions of the subquadratic and parallel tasks triad census algorithms processing a 3,774,768-node, 16,522,438-edge patents network. Execution is conducted on a 16-processor Cray XMT.

In reviewing the compile logs for the implicit parallelism and loop future codes, we find that for the implicit parallelism versions, given well defined iteration spaces, the compiler automatically parallelizes the outer two loops of the triad census computation through a general loop collapse. The generated iterations are executed on threads across multiple processors. For the loop future versions, the compiler also automatically optimizes the outer two loops, but schedules only the outer loop iteration across multiple processes. The inner loop executes on a single processor in what is known as *fray* mode, where the compiler implements fork and join operations inline using very short instruction sequences. Fray parallelism has low overhead. We believe this contributes to the higher performance of the loop future versions over the implicit parallelism versions of the triadic analysis algorithms.

4.2 Triad Census Algorithm Comparison on 128-processor XMT

With limited access to the 128-processor XMT, we continued to evaluate the performance and behavior of the subquadratic triad census algorithm on larger network problems and a larger XMT machine. For performance testing, we created two random networks consisting respectively of 12 million nodes with 120 million edges and 35 million nodes with 350 million edges. We continued to evaluate both the implicit parallelism and loop future versions of the subquadratic algorithm on these two large networks. During execution, the processing of the 12 million-node network required up to 85 GB of memory, while the 35 million-node network required up to 220 GB.

As shown in Figure 10a, the loop future version of the subquadratic algorithm continues to execute faster than the implicit parallelism version for both large networks. In examining the scaling performances of the subquadratic versions in Figure 10b, we may observe that the speedup rate levels off or tails down for all versions as we move from 96 to 128 processors. The degradation is most pronounced for the implicit parallelism version processing the 35 million-node network and least pronounced for the loop future version also processing the 35 million-node network. Since the degradation is evident in the processing of both the 12 and 35 million-node networks and is most pronounced with the 35 million-node network, this behavior is not likely to be attributed to the graph problem being too small and the processors running out of work. Rather, we believe in these instances that the codes are saturating some aspect of the system such that the hardware threads are not fighting for execution slots but delayed by network latencies of remote loads. These runs may be hitting the network tuning limits of the system.

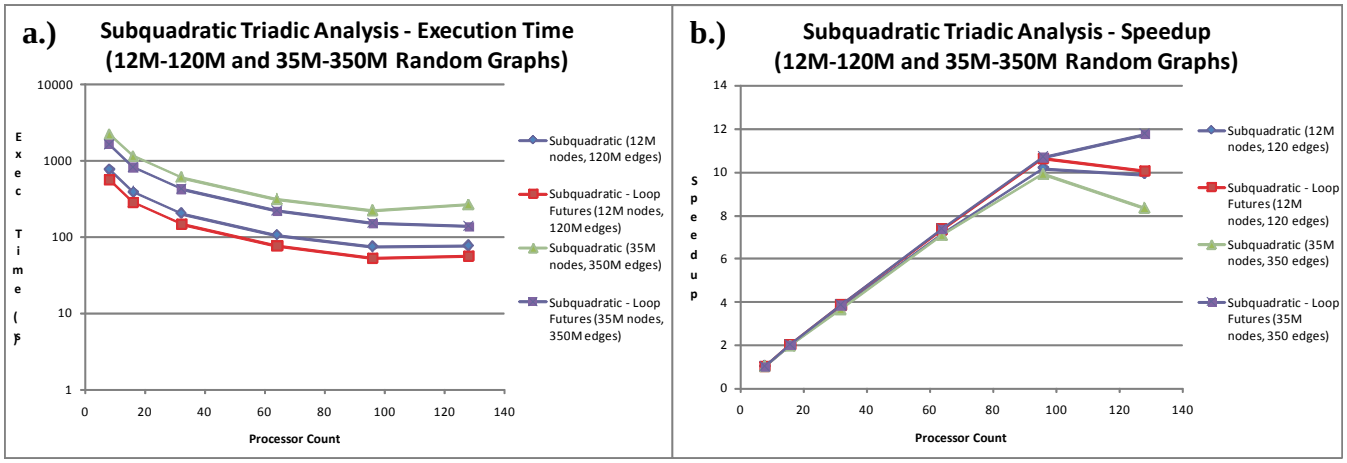

Figure 10. a.) Execution times and b.) speedup rates of implicit parallelism and loop future versions of the triad census algorithms processing a 12 million-node, 120 million-edge network and a 35 million-node, 350 million-edge network. Execution is conducted on a 128-processor Cray XMT.

To test this theory, we re-executed the implicit parallelism subquadratic algorithm using different stream limits to identify the potential network saturation point. On the XMT, the default stream limit is 100. One may set the stream limit at runtime in the user environment by issuing the *bash* command

export MTA_PARAMS=”stream_limit 70”

or *csh* command

setenv MTA PARAMS “stream_limit 70”

Figure 11 shows the performances of the subquadratic algorithm processing the 12 million-node network with stream limits of 65, 70, 75, and 100. As shown, the performance of the subquadratic algorithm improves with higher stream limits up to 96 processors. At 128 processors, however, the execution of the algorithm at the higher stream limits degrades. Among the four stream limits, the performance of the subquadratic algorithm on the 12 million-node network was best for the stream limit of 70.

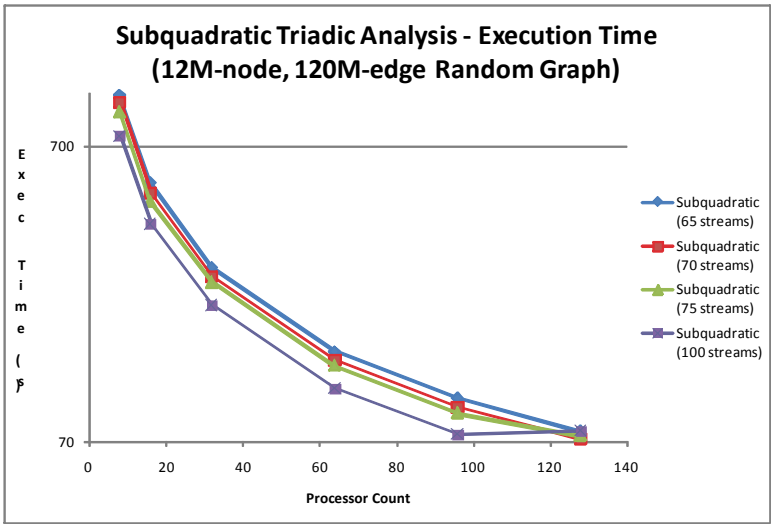

Figure 11. Execution times of implicit parallelism versions of the subquadratic triad census algorithm processing a 12 million-node, 120 million-edge network with different stream limits. Execution is conducted on a 128-processor Cray XMT.

Examining the speedup rates across the different stream limits, Figure 12a shows the degradation in the scaling of the 75 and 100 stream limit runs compared to the 65 and 70 stream limit runs. To better illustrate the stream saturation behavior, we may plot the algorithm speedup rates against the total number of available hardware threads (number of processors × stream limit). As shown in Figure 12b, the algorithm performances scale reasonably well for the four stream limits up through about 9000 total hardware threads, upon which the speedup rates tails off. This finding is consistent with our observation that running the subquadratic algorithm with a stream limit of 70 is optimal since the total number of available hardware threads would be 128 x 70 = 8960.

We can repeat this evaluation with the 35 million-node network. As shown in Figure 13, the performance of the implicit parallelism subquadratic algorithm is better for the 100 stream limit over the 70 stream limit up to 96 processors. For 128 processors, however, the algorithm performance is better for the 70 stream limit. Also, the scaling of the algorithm with a 100 stream limit starts to degrade after 96 processors as shown in Figure 14a, and after 9,000 total number of available hardware thread as shown in Figure 14b, which mirrors the behavior and results found with the 12 million-node network.

4.3 CPU Utilization Profiles

We took a closer look at the CPU utilization of the subquadratic algorithm for both the implicit parallelism and loop future versions. Figure 15 shows the CPU utilization of both subquadratic triad census versions over time when processing the 12 million-node network using 36 processors on the 128-processor Cray XMT. The two humps in the figure illustrate two stages of the code. The first hump plateauing around 20% CPU utilization identifies the input stage of the code where the network is read in from file, while the second hump identifies the triad census computation. For both the implicit par-

allelism and loop future versions, the triad census computation reaches about 50-55% CPU utilization. The CPU utilization of the implicit parallelism version, however, tails downward over an extended period of time, while the loop future version maintains the higher CPU utilization longer and then tails down quickly. These utilization profiles show that the loop future version executes more efficiently than the implicit parallelism version.

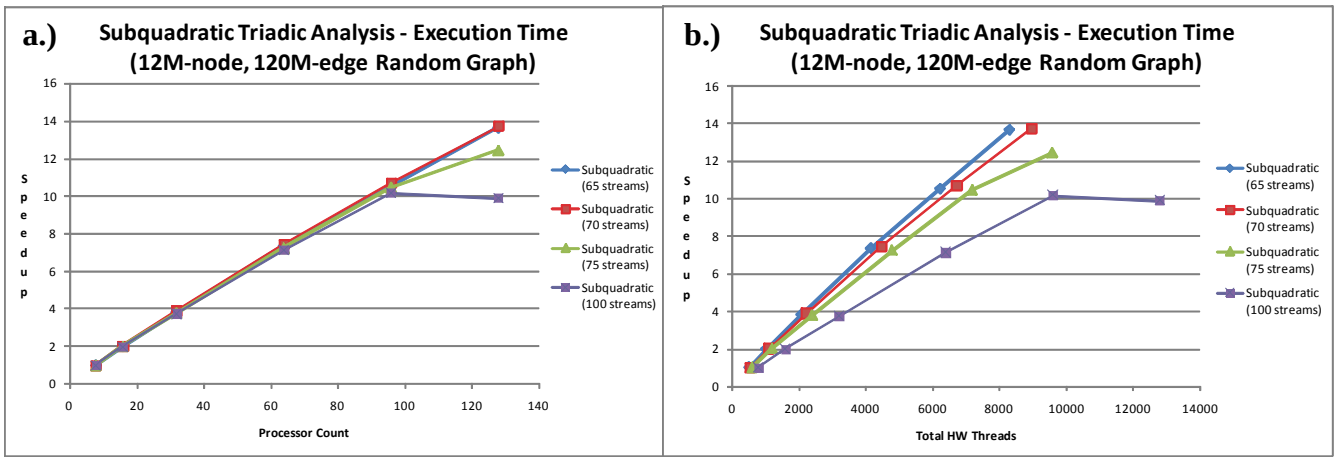

Figure 12. Speedup rates of implicit parallelism versions of the subquadratic triad census algorithm processing a 12 million-node, 120 million-edge network with different stream limits mapped against total number of a.) processors and b.) available hardware threads. Execution is conducted on a 128-processor Cray XMT.

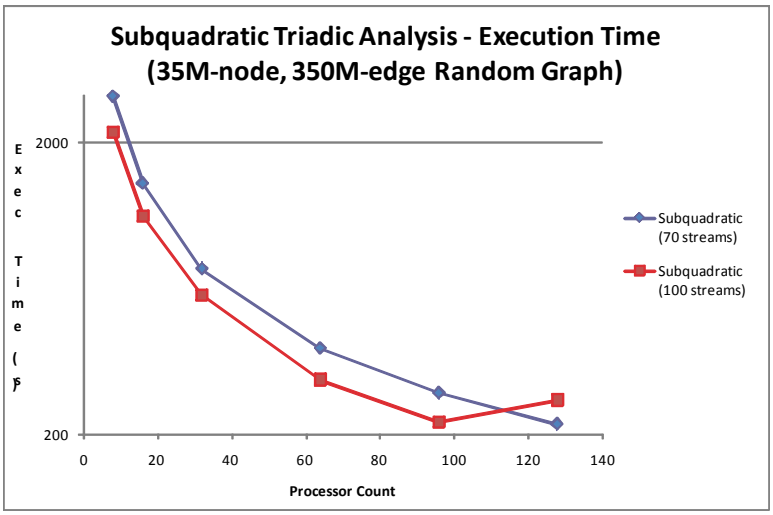

Figure 13. Execution times of implicit parallelism versions of the subquadratic triad census algorithm processing a 35 million-node, 350 million-edge network with different stream limits. Execution is conducted on a 128-processor Cray XMT.

In an attempt to improve the load balancing of the implicit parallelism version, we inserted the following compilation directive into the code above the outer loop of the triad census computation.

#pragma mta interleave schedule

In an interleaved schedule for a parallel loop, the compiler assigns contiguous iterations to distinct streams. For a loop with 100 iterations and 10 streams available, one stream performs iterations 1, 11, 21, …, another performs iterations 2, 12, 22, ..., etc. An interleaved schedule results in better load ba-

lancing for triangular loops. With the subquadratic algorithm, the nested loop has some triangular properties - the variable of the outer loop traverses through every node, while the inner loop traverses through every neighbor of the outer node but through only those that have a smaller index value.

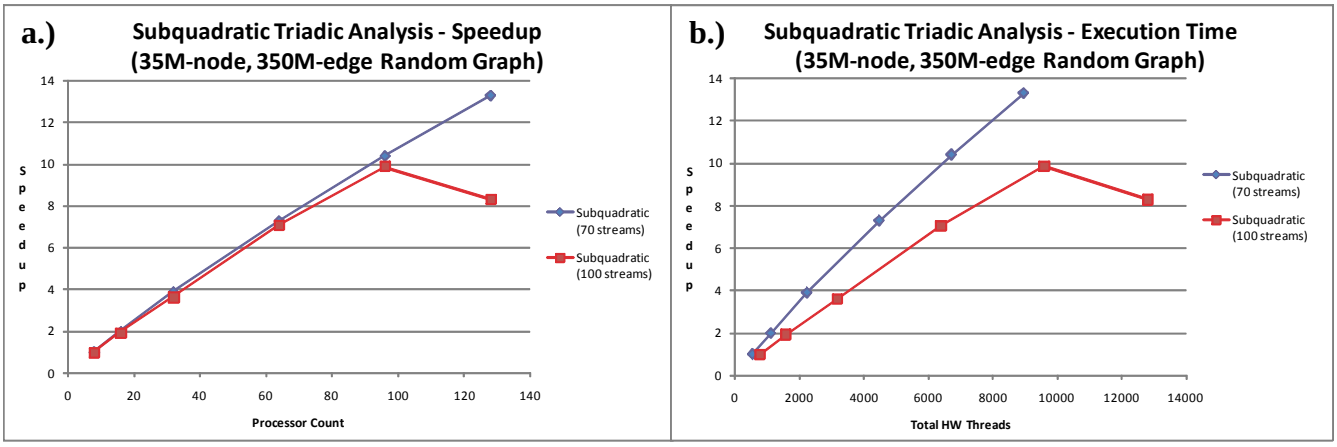

Figure 14. Speedup rates of implicit parallelism versions of the subquadratic triad census algorithm processing a 35 million-node, 350 million-edge network with different stream limits mapped against total number of a.) processors and b.) available hardware threads. Execution is conducted on a 128-processor Cray XMT.

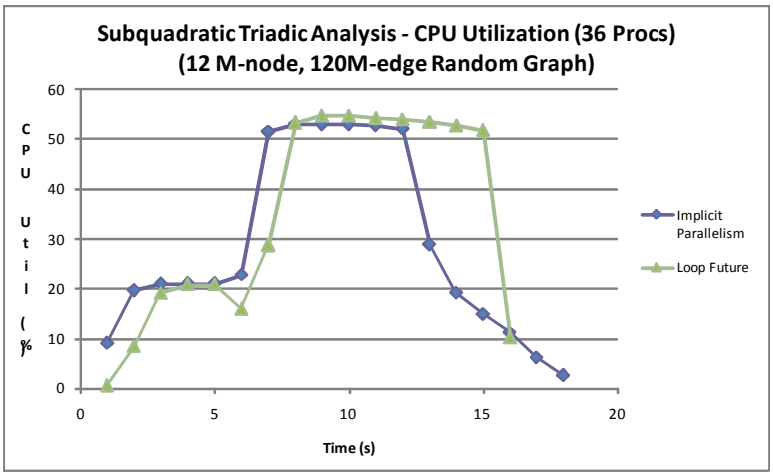

Figure 15. CPU utilization profiles for the implicit parallelism and loop future versions of the subquadratic triad census algorithm processing a 12 million-node, 120 million-edge network. Execution is carried out on 36 processors of a 128-processor Cray XMT.

As shown in Figure 16, the interleaved schedule version of the subquadratic algorithm better maintains the higher CPU utilization rate and avoids the trailing tail found in the utilization profile of the original implicit parallelism code. Furthermore, the execution time improved to be comparable to the loop future version of the code in processing the 12 million-node network over 36 processors. In testing the scheduled interleave version over a range of processors, we found that both the interleaved schedule and loop future versions have comparable execution times up to 96 processors as shown in Figure 17. At 128 processors, the performance of the loop future version is still better than the scheduled interleave version due to the stream saturation effects we previously described.

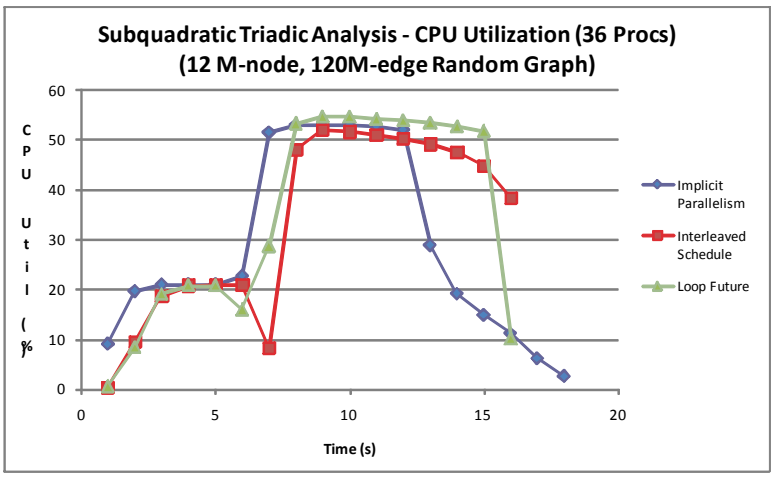

Figure 16. CPU utilization profiles for the implicit parallelism, interleaved schedule, and loop future versions of the subquadratic triad census algorithm processing a 12 million-node, 120 million-edge network. Execution is carried out on 36 processors of a 128-processor Cray XMT.

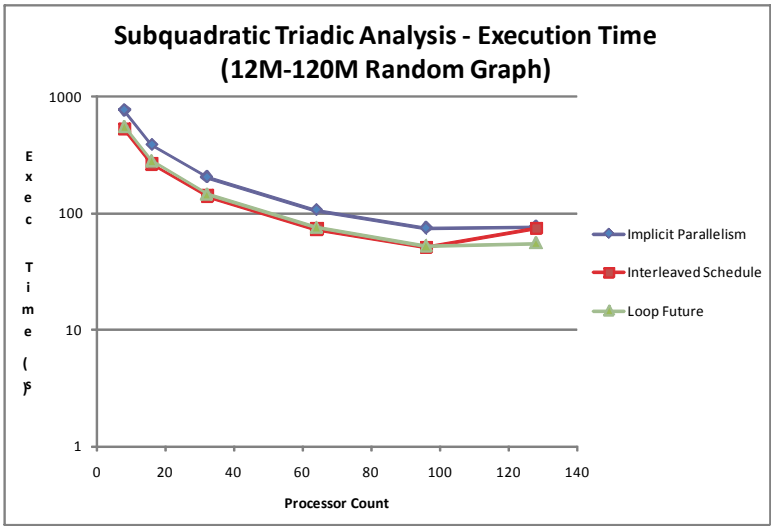

Figure 17. Execution times for the implicit parallelism, interleaved schedule, and loop future versions of the subquadratic triad census algorithm processing a 12 million-node, 120 million-edge network. Execution is conducted on a 128-processor Cray XMT.

We further tested the interleaved schedule version with the 35 million-node network and did not find the same improvements as we did with the 12 million-node network. As shown in Figure 18, the CPU utilization profiles for the implicit parallelism and interleaved schedule versions both had extended tails. Furthermore, both versions closely tracked one another and continued to trail the loop future version in execution time across different numbers of processors as shown in Figure 19.

One possible explanation for the CPU utilization differences might be that the triangular loop pattern is less pronounced for the 35 million-node network than the 12 million-node network. Both networks had ten times as many edges as nodes, and thus, each node should average 20 neighbors (each edge contributes two neighbors). Recall that the inner loop in the triad census computation traverses over the neighbors (with smaller index values) of the node identified by the variable of the outer loop. Thus, the size of the bounds of the inner loop should be the same for both networks, but the bounds of the outer loop is much larger for the 35 million-node network than the 12 million-node network. Giv-

en that the triangular pattern was less pronounced for the 35 million-node network, the load-balancing effects of an interleaved schedule may have been limited.

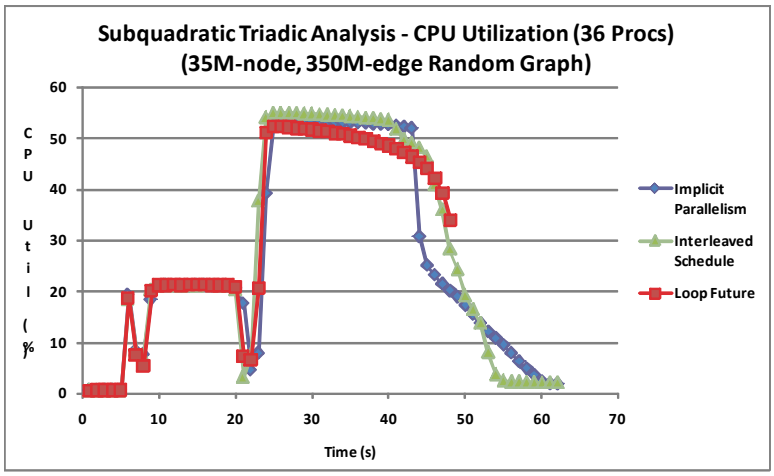

Figure 18. CPU utilization profiles for the implicit parallelism, interleaved schedule, and loop future versions of the subquadratic triad census algorithm processing a 35 million-node, 350 million-edge network. Execution is carried out on 36 processors of a 128-processor Cray XMT.

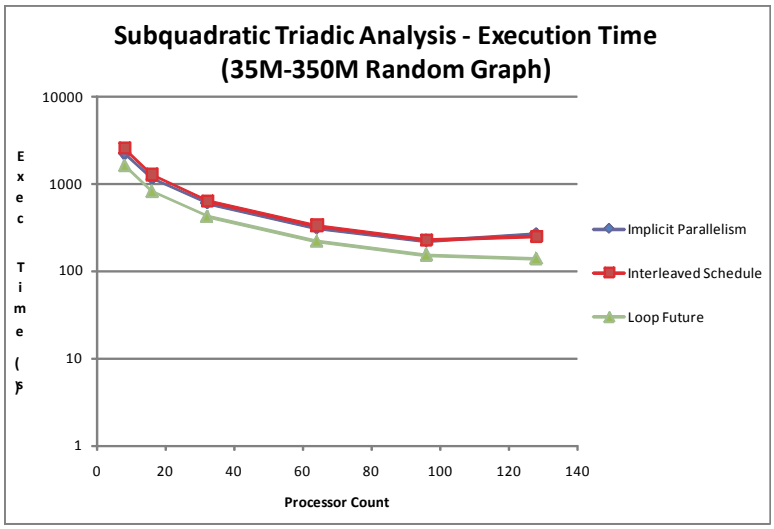

Figure 19. Execution times for the implicit parallelism, interleaved schedule, and loop future versions of the subquadratic triad census algorithm processing a 35 million-node, 350 million-edge network. Execution is conducted on a 128-processor Cray XMT.

Another explanation might simply relate to the variability of random networks. Although the average number of neighbors of a node in the two networks we tested was the same, any particular node could have 0 or much more than 20 neighbors. So, for any random network, the shape of the nested loops in the subquadratic algorithm could appear roughly triangular or totally random. This reminds us that as we parallelize and optimize particular algorithms, we must also parallelize and optimize for the specific problems and data that we are addressing.

5 Conclusions and Future Plans

In this paper, we present three difference triad census algorithms, each possessing different structural and execution characteristic. The brute force algorithm is simple and clean, possesses a triangular

loop, and has the highest computational complexity ($O(n^2)$, n = number of nodes). The subquadratic algorithm provides lower computational complexity (O(k(n)*n), n = number of nodes) for sparse networks and also possesses a triangular loop. The parallel tasks algorithm also provides lower computational complexity (O(k(n)*n), n = number of nodes) for sparse networks and attempts to load-balance triad census computations in code.

We implemented, optimized, and evaluated the three triad census algorithms on the Cray XMT. We explored the use of various compiler directives to tune the performances of the algorithms. From our evaluations, we found that the subquadratic and parallel tasks algorithms performed comparably and magnitudes better than the brute force algorithm. Furthermore, we found that loop futures generally offered the best performance in terms of load balancing. For certain networks, we were are to able to achieve comparable performance using interleaved schedules. The loop future versions of the algorithms, however, exhibited less performance degradation at 128 processors than the implicit parallelism and schedule interleave versions.

Our performance evaluation of triad census algorithms would not be complete until we are able to compare our XMT results with those of other parallel systems and architectures. We are porting and will be evaluating our triad census codes on other shared memory architectures such as Sun Niagara 2, Silicon Graphics Altix, Hewlett-Packard Superdome, and SMP (x86, multicore).

Currently, we are also evolving the triad census algorithms to support the analysis of dynamic networks. In dynamic triadic analysis, we are not only interested in the proportion of triads in the triad census, but how those proportions change over time. The transition of triads across time intervals or frames would reveal the dynamic structure and evolution of a social network. As certain types of triads dissipate and give way to other triad types, we see transitions in behavior of people or entities over time. In this way, one might see groups or hierarchies forming or disbanding, transactions moving across the network, or activities decreasing or intensifying – all based on how the triad census changes over time. Consequently, the added time dimension further increases the complexity of the triad algorithms.

ACKNOWLEDGMENTS

This work was funded under the Center for Adaptive Supercomputing Software - Multithreaded Architectures (CASS-MT) at the Dept. of Energy's Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

REFERENCES

1. Bader, D.A. and Madduri, K. (2006). Designing multithreaded algorithms for breadth-first search and *st*-connectivity on the Cray MTA-2. In *Proceedings of the 35th International Conference on Parallel Processing,* Columbus, OH, August 2006, pp. 523-530.
2. Chavarría-Miranda, D., Márquez, A., Maschhoff, K., and Scherrer, C. (2008). Early experience with out-of-core applications on the Cray XMT. In *Proceedings of the 2008 IEEE International Parallel and Distributed Processing Symposium*, Miami, FL, April 2008, pp. 1-8.
3. Nieplocha, J., Márquez, A., Feo, J., Chavarría-Miranda, D., Chin, G., Scherrer, C., and Beagley, N. (2007). Evaluating the potential of multithreaded platforms for irregular scientific computations. In *Proceedings of the 4th International Conference on Computing Frontiers*, Ischia, Italy, May 2007, pp. 47-58.

4. Chin Jr., G., Chavarria, D.G., Nakamura, G.C., and Sofia, H.J. (2008). BioGraphE: high-performance bionetwork analysis using the Biological Graph Environment. *BMC Bioinformatics*, 9 Suppl 6:S6.
5. Wasserman, S. and Faust, K. (1994). *Social Network Analysis*, Cambridge University Press, Cambridge, UK.
6. Batallas, D.A. and Yassine, A. (2004). Information leaders in product development organizational networks: Social network analysis of the design structure matrix. In *Proceedings of the Understanding Complex Systems Symposium*, University of Illinois at Urbana-Champaign, May 2004.
7. de Nooy, W., Mrvar, A., and Batagelj, V. (2005). *Exploratory Social Network Analysis with Pajek*, Cambridge University Press, Cambridge, UK.
8. Wittek, R. and Wielers R. (1998). Gossip in organizations. Computational & Mathematical Organization Theory, 4(2), pp. 189-204.
9. Batagelj, V. and Mrvar, A. (2001). A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, 23(3), pp. 237-243.
10. Diesner, J., Frantz, T., and Carley, K. (2005). Communication networks from the Enron email corpus "It's always about the people. Enron is no different." *Computational & Mathematical Organization Theory,* 11(3), pp. 201-228.
11. Aviv, R., Erlich, Z., Ravid, G., and Geva, A. (2003). Network analysis of knowledge construction in asynchronous learning networks. *Journal of Asynchronous Learning Networks*, 7(3).
12. Ingram, P., Robinson, J., and Busch, M.L. (2005). The intergovernmental network of world trade: IGO connectedness, governance, and embeddedness. *American Journal of Sociology*, 111(3), pp. 824-58.
13. Chin Jr., G., Kuchar, O.A., Whitney, P.D., Powers, M.E., and Johnson, K.E. (2004). Graph-based comparisons of scenarios in intelligence analysis. In *Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics*, The Hague, The Netherlands, October 2004.
14. Moody, J (1998). Matrix methods for calculating the triad census. *Social Networks*, 20, pp. 291-299.
15. Cray, Inc. (2008). Cray XMT™ Programming Environment User's Guide, Seattle, WA.
16. Cray, Inc. (2008). Cray XMT™ Performance Tools User's Guide, Seattle, WA.
17. Cray, Inc. (2008). Cray XMT™ System Overview, Seattle, WA.