Efficient Scaling Up of Parallel Graph Algorithms for Genome-Scale Biological Problems on Cray XT

Kevin Thomas, *Cray Inc.*
Nagiza F. Samatova, *North Carolina State University and Oak Ridge National Laboratory*
Matthew Schmidt, *North Carolina State University and Oak Ridge National Laboratory*
Byung-Hoon Park, *Oak Ridge National Laboratory*

ABSTRACT: *Many problems in biology can be represented as problems on graphs, but the exact solutions for these problems have a computational burden that grows often exponentially with increasing graph size. Due to this exponential growth, high-performance implementations of graph-theoretic algorithms are of great interest. We are developing a* ***pGraph*** *library for parallel graph algorithms of relevance to biological problems. This paper discusses the implementation decisions made during the development of a class of enumeration algorithms on graphs. The data-intensive nature and highly irregular structure of the search space make efficient scaling up of such algorithms quite challenging. A specific scalable and efficient implementation of a ubiquitous maximal clique enumeration problem in biology is discussed. Performance results on real biological problems from Cray XT are presented.*

KEYWORDS: Parallel Graph Algorithms, Maximal Clique Enumeration, Biology, Cray XT

1. Background

A *graph* is a set of *vertices* and the *edges* that connect them. Two vertices are said to be *adjacent* if an edge between them exists. The *degree* of a vertex is the number of edges which connect to it. A *clique* is a set of vertices in which each vertex is adjacent to every other vertex in the set. A clique is *maximal* if no other vertex can be added to the set and the set remain a clique. *Maximal clique enumeration* (MCE), which is the focus of this work, takes a graph description as input and produces the maximal cliques of the graph as output.

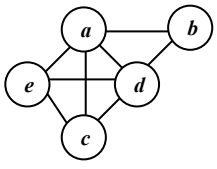

Figure 1. A graph with five vertices and eight edges. The maximal cliques include (a,c,d,e) and (a,b,d).

2. Introduction

Data-driven construction of predictive models for biological systems is often considered as a combinatorial optimization problem, where a search for a particular object or enumeration of all the objects with given properties is being sought. Exact algorithms for combinatorial problems on biological systems frequently use recursive strategies for exploring the search space to find the optimum solution.

Backtracking [1] is one of the most widely used recursive strategies. Unlike brute force methods, which exhaustively search through the input, a backtracking method avoids exploring unpromising paths by applying the specific property of the sought solution. For example, in case of MCE, the property that constrains the search space is the completeness of the clique, i.e. all vertices in a search path should be pair-wise connected. An efficient backtracking algorithm may adopt additional constraints to further limit unnecessary traverses. All possible paths that a

backtracking algorithm can explore are represented as a *search tree*. A path (from the root) in a search tree corresponds to a sought solution (e.g. clique), and the tree is expanded as an algorithm traverses the graph. The algorithm stops expanding a path and returns to its previous level (backtracks), if no further expansion in the current direction leads to new feasible solutions (e.g. cliques). A search tree can be split into a number of disjoint sub-trees. Since each sub-tree can also be recursively further split and independently explored, with a balanced and efficient decomposition strategy, backtracking-based algorithms can be adapted to a high performance parallel computing environment.

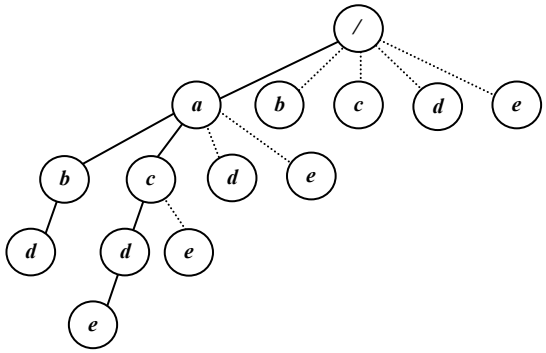

Figure 2. A backtracking search tree for the graph of Figure 1. The redundant paths shown as dashed lines.

Graph algorithms have characteristics that distinguish them from many classic HPC applications. An obvious difference is that graph algorithms do not take advantage of floating point hardware. The code tends to be "branchy," because the dominant operations are comparisons of the properties of vertices and edges. The memory access pattern tends to be random across the data structures, so spatial and temporal localities are low. Memory access latency is an important component in algorithm performance.

3. Parallel MCE Implementation

A parallel application to enumerate the maximal cliques of a graph, called **pDFS**, has been developed as part of a larger effort to develop a **pGraph** library of parallel graph algorithms. pDFS is a parallel implementation of the backtracking algorithm proposed by Bron and Kerbosch [2], with a modification to dynamically identify the most suitable ordering in expanding each path.

The earliest known parallel MCE application was a shared-memory parallel algorithm [6] inspired by the breadth-first MCE algorithm of Kose et al [3]. Earlier work by our group developed a shared-memory parallel (SMP) *Clique Enumerator* based on pthreads [5]. This has been extended in pDFS with a distributed-memory parallel (DMP) implementation using MPI.

For flexibility and efficiency, the two parallel programming techniques, SMP and DMP, are used in a complimentary fashion. Shared memory parallelism with pthreads is efficient, but its scalability is limited to a single compute node. MPI expands this by providing distributed memory parallelism, with some extra cost in data replication, communication, and programming complexity. By combining both techniques, the best performance can be achieved. Flexibility is preserved by allowing the application to use pthreads, MPI, both together, or neither, depending upon the target computer system.

3.1. Design of the Work Pool Data Structure

Parallelization, the process of converting a single sequential execution into multiple sequential threads, requires that the overall task be decomposed into multiple tasks. Ideally, each task is independent. Since MCE has obvious independent work, the traversal of the search sub-trees, a parallel implementation seems straightforward. An immediate difficulty encountered is in developing the mechanics of this decomposition in a way that works efficiently within the constraints of the available programming models and computer systems. In particular, how can this work be divided evenly between compute nodes within a distributed memory environment, with minimum synchronization and data transfer. This is especially important for a data-intensive application such as MCE which can require the output of an exponential number of cliques in the worst case [4]. One of the key features of the SMP implementation is an explicit representation of an unexplored search sub-tree. Such a data structure enables dynamic load balancing, which requires this data to be easily migrated from one thread to another. In a shared-memory environment, this data does not need to be copied between threads, but the representation is crucial to the DMP implementation, where the candidate path must be transferred between compute nodes.

The devised data structure, called a *candidate path*, contains these components:

- The clique represented by the path from the root to the current node in the search tree.
- All eligible vertices for the path (i.e. common neighbors for all the nodes in the above clique).
- Vertices covered earlier in expanding the parent path (to avoid redundant coverage).

The heart of the implementation is a set of work pools, one for each thread, containing candidate paths. Each

thread explores its portion of the search tree by removing items from its pool, and then fills the pool as new paths are discovered. When a pool becomes empty (all work has been completed), a load balancing step is used to keep the thread busy. Traversal of the search sub-trees is independent work, so no communication is required during this phase.

The pool is accessed like a stack as the search progresses, with last-in-first-out (LIFO) ordering. The most recently explored node is expanded into additional candidate paths, which are pushed back onto the stack. This results in a depth-first traversal of the search tree.

3.2. Reading the Input Graph

Although it seems like a simple task, the reading of the graph description file can become a factor limiting scalability. All processes require a copy of the graph data.

A simple implementation where all processes read the input file is limited by file system scalability. To get around this, the file is read by a single process that broadcasts the data to the other processes. One efficient broadcast operation can be used because the graph data is packed into a single data structure containing an adjacency list for each vertex.

A header in the input graph file contains the vertex and edge totals for the graph. This allows for pre-allocation of sufficient contiguous space to hold the graph as the rest of the file is read in. The adjacency list for each vertex is managed as an array rather than a linked list. This makes the graph data more compact, results in better cache behavior when the adjacency list is accessed, and also eliminates address pointers. Address pointers are not portable between processes, and would need to be regenerated locally after the broadcast.

The array construct leaves open the possibility of using binary search of the adjacency list when memory constraints prevent the use of other techniques. This issue is important for per-thread performance, and is detailed in Section 4 below.

It is convenient to have the vertex and edge totals known before the graph data is read, but the same data structure can be used without these values. A simple approach would be to add an extra pass through the input data to obtain the graph size before allocating space and storing the data. A more efficient technique is to initially allocate based on a graph size estimate, then reallocate a larger block if the estimate proves to be too small. With a good estimate and careful expansion, the overhead of reallocation can remain small in comparison to the cost of reading the input data.

Auxiliary data structures, such as the adjacency hash table or adjacency bit matrix, are generated redundantly by each process. It is cheaper to generate this data locally than to transfer it through the network.

3.3. Initial Work Partitioning

There is no practical way to determine a well-balanced distribution of work for MCE *a priori*. Since dynamic load balancing is required, the initial partitioning can be performed by simply dividing the graph vertices equally among the threads. Each vertex added to the work pool becomes a root node of a search sub-tree. In order to be added to the work pool as a root, the vertex must satisfy all the normal constraints to establish a candidate path.

Given this partitioning scheme, this step is done independently by the threads and no communication is required. Each thread selects root vertices based on its global thread number, which is computed using the local thread number and the MPI process rank.

In the example graph of figure 1, which corresponds to the search tree of figure 2, the imbalance in the initial partitioning is obvious. The thread that selects vertex ***a*** will have all of the remaining work, while the threads selecting the other vertices will have no further work. Graphs derived from real biological data can result in similar search tree imbalances. An additional source of imbalance comes from the discovery of maximal cliques, which requires extra work be performed at a search tree node.

3.4. Dynamic Load Balancing

When a work pool becomes empty, load balancing is performed by stealing work from the pool of a random thread. *Stealing* means moving work from one pool to another without the direct involvement of the pool's owner. Unlike normal access to the pool, stealing uses first-in-first-out (FIFO) order, removing items from the opposite end of the pool. This has the effect of transferring nodes from high up in the search tree, which have a greater chance to expand.

Two levels of load balancing are used, corresponding to the levels of parallelization. Within a process, threads can steal from one another by simply removing an item from the pool. Locks are used to prevent the owner thread, or another stealing thread, from accessing the pool during this operation. This local load balancing is used until all threads of the process have no work.

Between processes, a more complex scheme is needed. A load balance request message is sent to a random target process, which responds with any items that it wishes to share. The term stealing still applies, since the responding target process must steal the work from local threads in order to send it to the requesting process.

In order to make the remote stealing efficient, more than one item may be stolen per request. The request message specifies the number of idle threads in the requesting process, and up to that many items may be returned.

In both cases, local and remote, if an attempt to steal work fails because the work pools are empty, the search continues on to the next target, until all targets (threads or processes) have been polled.

In addition to the case where the target work pool is empty, and the attempt to steal work fails, if the pool contains only one item, the item is not moved, since it could just as well be processed by the current owner. This presents a special case, since it is possible, though unlikely, that only a single candidate path remains to be examined. For SMP load balancing, the threads other than the owner of the non-empty work pool enter the DMP load balancing state, waiting for the final thread to complete its work so the DMP load balancing step can begin. In this way, the transition from SMP to DMP load balancing within a process is a thread barrier. But what if this last candidate path expands into a large number of new items added to the work pool?

A last-step explosion of local work could leave a single thread to continue processing while the other threads are idle at the barrier. This would create a significant load imbalance. This work stealing race condition is seen in rare instances, and overall performance can be significantly reduced.

To avoid this problem, a work stealing attempt has three possible outcomes: success, empty pool, and nearly empty pool. Since the last case indicates that local work remains, threads receiving this work stealing result enter a new cycle to recheck all local work pools. For most cases, all work pools are empty after the second iteration, but, any sudden work explosion can be handled gracefully.

3.5. Termination Condition

Since the enumeration proceeds without global synchronization, a technique to complete processing in an orderly way is needed. The first issue is that an individual process must poll all other processes to determine that no process has work to share, i.e. no further load balancing can occur. Secondly, each process must handle load balancing requests until it can determine that all other processes have reached the same point. This implies that a process cannot terminate (i.e., call MPI_Finalize) even when no further work is available.

Shutdown is therefore takes place in phases. First a process polls all other processes for work to share. If all requests fail, the process is ready to enter an idle state. In the idle state, it continues to respond to load balancing requests, but first it sends an idle status notification to every other process. When idle notifications are received from all other processes, then the entire application has reached a quiescent point and all processes can terminate.

While this procedure does generate a flurry of communication, the effect on overall application performance has not been significant. If this simple broadcast scheme of communication becomes a performance issue, either at an extreme scale of compute nodes or on a particular computer system, then hierarchical communication may be necessary by partitioning the processes into smaller groups.

3.6. Asynchronous Requests and Notifications

In order to manage load balance requests and idle notifications, a flexible asynchronous message handling capability was developed. One thread of each process is responsible for this work, which is handled via MPI asynchronous point-to-point communication functions.

Asynchronous communication was used for better portability. Many MPI implementations use spin-waiting within blocking calls, which conflicts with multi-threaded execution within the main computation. In effect, a processor core becomes dedicated to communications.

To avoid this, one computational thread within a process also handles communication. Polling for incoming messages takes place after each iteration of the main work loop, taking advantage of the fine-grained nature of the underlying algorithm. The overhead placed on this thread due to this extra responsibility is effectively spread amongst all of the threads via local load balancing. Since random work stealing is used, the overhead is well-distributed over all the processes.

4. Improving the Base Algorithm Efficiency

In the clique enumeration base algorithm, a *connected* predicate is used repeatedly to determine if the current vertex shares an edge with a test vertex. Since this operation occurs many times, it should have an efficient implementation.

4.1. Linear Search of the Adjacency List

The simplest implementation is to search the list of vertices adjacent to the current vertex, since this list is part of the basic graph description. While this uses no additional memory, it is also the slowest method. It is especially bad for a common case, where the test vertex is not in the list.

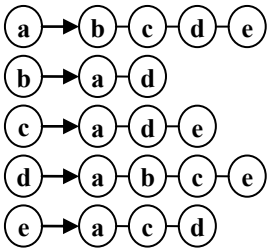

Figure 3. The adjacency lists for the vertices of the graph in Figure 1.

A linked list is a flexible data structure, but searching a linked list is slow because of its pointer-chasing nature. A linear list is much faster to search, both because it removes the pointer chasing behavior, but also because it takes advantage of cache structure. As noted in section 3.2, another advantage of the linear list is that it can be moved between processes, since it contains no address pointers.

4.2. Adjacency Matrix Lookup

The fastest method is to construct an adjacency matrix from the graph description with the value for a given pair of vertices determining whether the vertices are connected or not. A bit matrix is used, so the memory required is n^2 bits, where n is the number of vertices in the graph. This is reduced by half when symmetry of the matrix is exploited. A limitation of this technique is that a matrix for a graph with a large number of vertices may not fit in memory.

	a	**b**	**c**	**d**	**e**
a	-	1	1	1	1
b	1	-	0	1	0
c	1	0	-	1	1
d	1	1	1	-	1
e	1	0	1	1	-

Figure 4. The adjacency matrix for the vertices of the graph in Figure 1.

4.3. Adjacency Hash Table Lookup

A fast technique which often requires only a limited amount of extra memory is a hash table. In place of the list of adjacent vertices, a hash table of adjacent vertices is used. The size of the table is a small constant factor larger than the list representation, and the average lookup requires few memory accesses.

A simple hash function (the test vertex number modulo the table size) is used to compute its index into the current vertex's hash table. A table collision occurs when two vertices have the same hash index. In the worst case, the hash table lookup time becomes linear in the vertex degree if all vertices have the same hash index. To avoid this, a size expansion factor is used to allocate a table several times larger than vertex degree when the table is created. A sparsely populated hash table has much less chance of encountering the worse case. In the average case, the lookup time is constant with respect to the vertex degree.

The memory required for the hash table grows with the number of edges in the graph. The hash table approach is applicable when the average vertex degree is much smaller than the number of vertices; otherwise, the adjacency matrix is more compact.

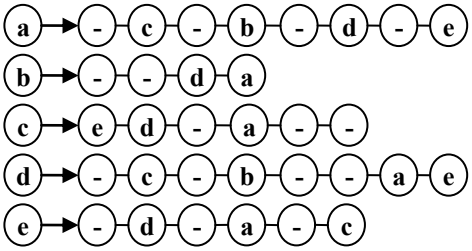

Figure 5. An example adjacency hash table with expansion factor of 2. "-" indicates an empty table entry.

4.4. Algorithm Selection

The library implements all three adjacency test methods. pDFS uses the adjacency matrix if enough memory is available, defaulting to the hash table for large graphs. If insufficient memory for the hash table is available, the adjacent vertex list search is used.

An alternative to the linear search of the adjacency list is to sort the list and use binary search when the list size (or sub-list size) is large. For the graph sizes and compute node sizes considered, the lookup techniques are the best choice. If the graph sizes of interest grow substantially, or the compute nodes used are significantly smaller, binary search may be the preferred algorithm.

At the shared-memory level of parallelization, multiple threads can share these graph description data structures, reducing the memory requirement.

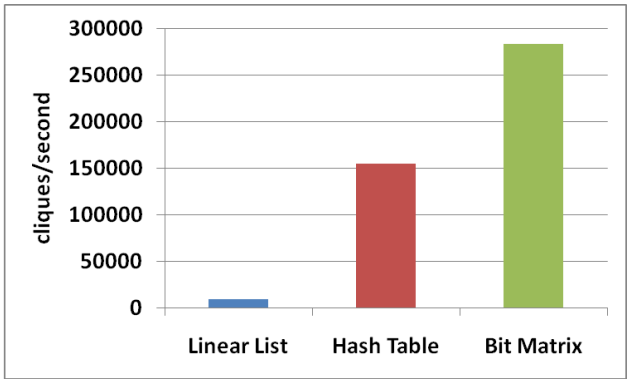

Figure 6. Performance of MCE with different vertex adjacency test methods, in cliques generated per second.

The hash table is 17 times faster than the linear list search, and the bit matrix is 1.8 times faster than the hash table. The absolute performance varies depending upon the input graph, but the relationship between them is constant.

5. Performance

5.1. Parallel Programming Model Comparison

A benchmark was run using a single node server with two quad-core Opteron processors to compare the SMP and DMP implementations. Each run used all 8 processor cores, but varied how the cores were used. There are four possible combinations of threads and processes for this benchmark. One case used multiple threads within a single process, and at the other extreme each process used a single thread. Two runs mixed multiple threads and processes in a hybrid parallel mode (MPI and pthreads used together).

The MPI library used shared memory to transfer messages between processes. This allows the comparison to focus on the application differences for the two levels of parallelization.

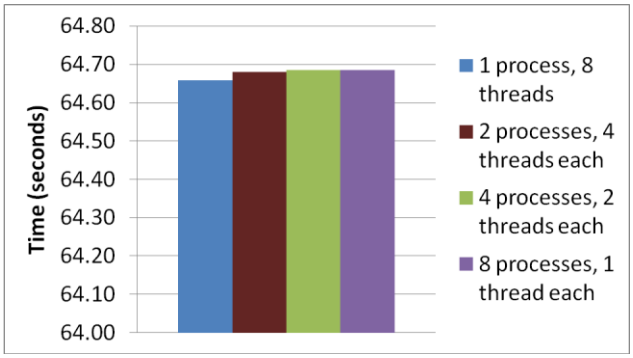

Figure 7. Performance comparison of processes versus threads. The lower bound of the chart is set to magnify the differences.

This result shows that, at small scale, the DMP implementation of MCE is nearly as efficient as SMP one. This is important because one goal of this work was to retain the efficiency of the SMP implementation, but to extend the scalability of the application via DMP.

5.2. Parallel Scaling

To assess the overall scalability of the application, benchmark runs were performed on a quad-core Cray XT4 system using graphs derived from biological data. Figure 8 was derived from runs on a yeast stress response graph that has 3472 vertices and 246342 edges. Each edge represents a correlation between two genes. Similar behavior was observed for runs on other biological networks.

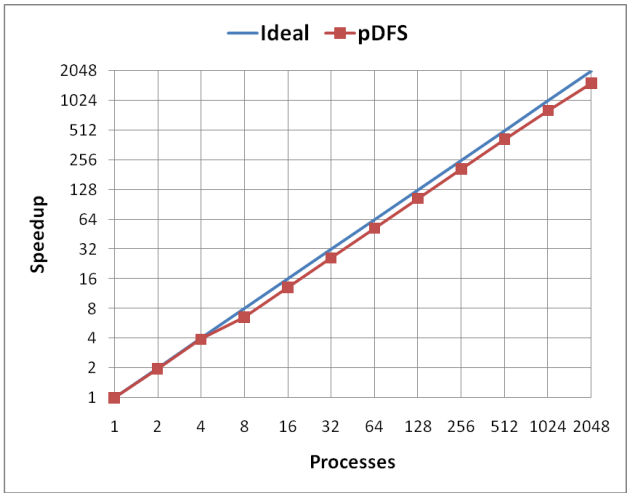

Figure 8. Parallel speedup (log-log scale).

At the highest scale, the parallel speedup begins to drop off from the ideal. This result is due to the parallel overhead growing in proportion to the MCE calculation time. For the 1 process run, the MCE calculation time is 4109 seconds, but this shrinks to 2.1 seconds at 2048 processes.

Another boundary to the scalability of this benchmark is that with 2048 processes and 3472 vertices, the initial work partitioning results in each process starting with at most 2 vertices. This can place an extra burden on the dynamic load balancing.

Writing of the maximal clique output files was disabled in these benchmark runs. The scalability of the underlying file system would have otherwise impacted the algorithm scalability. Section 7.2 comments further on this issue.

5.3. Overhead due to message passing

The DMP load balancing time can be broken down into two components: the time it takes to steal work from another process (fetch time) and the time it takes to

process incoming requests to steal work (request time). At the end of the run, some time is required to come to a globally idle state (idle time). These three parts of the run time account for most of the overhead due to message passing during parallel execution.

The same benchmark runs from the parallel scaling study were analyzed to separate out these components.

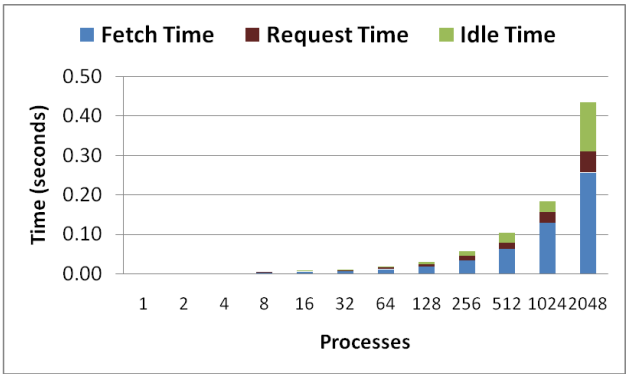

Figure 9. Message passing overhead times.

Although these times are only a small percentage of the overall run time for the benchmark runs, the trend shows that the overhead will become significant at extreme scale (10s of thousands of processes). This could be managed by grouping processes and using a hierarchical load balancing scheme. Load balancing would take place first within a group of processes, and then between groups. A two-level scheme may be sufficient for the foreseeable future (up to a million processes).

6. Conclusion

The distributed-memory parallel maximal clique enumerator pDFS runs with good parallel efficiency at high scale with input graphs from real-world biological data. The pGraph library can be used as a basis for the development of additional parallel graph algorithms.

One lesson learned from the work to add an MPI level of parallelization to an application written to use pthreads is that the selection of algorithm and supporting data structures is critical to success. It was a lucky circumstance that the approach used in the SMP implementation was compatible with adding MPI at a later point.

It is apparent that the use of high-level parallelism with pthreads created this compatibility. Had the application been parallelized at the much lower level of computational loops, no infrastructure to be expanded for multi-process use would have been available. As it was formulated, the pthreads implementation led smoothly to a combined MPI-pthreads application which greatly leveraged the earlier work.

This leads to an observation about the interaction of SMP and DMP programming techniques. It is possible to apply these independently to a program, choosing low-level parallelism for pthreads or OpenMP and a high-level explicitly-decomposed strategy for MPI. But if the required effort is applied to create a high-level decomposition for MPI, why not leverage this work for both SMP and DMP? In an era when the number of processor cores per shared-memory compute node is growing, hybrid parallel programming becomes more and more attractive.

The benefits seen for this application include:

- Less data replication, since many data structures can be shared by the threads within a process. This reduces the memory requirement, and can result in better cache use, reducing the demand on main memory bandwidth.
- More efficient shared-memory parallel execution, because it is realized at a high level.
- Fewer MPI processes are required for a given job size, reducing the scalability requirement at this level.

7. Future Work

7.1. Application to Other Computer Architectures

A key target application area for Cray XMT systems is graph algorithms. The XMT architecture employs 128 hardware threads per processor, with all processors of the system sharing a global memory. Although it supports neither pthreads nor MPI, there is a straightforward path to port pDFS to XMT systems. Global memory is required for extremely large graphs (millions of vertices), but applications can also exploit the local memory capability available on XMT systems. Local memory use has the potential to deliver higher performance, since average memory access latency is reduced and network congestion is avoided.

Remarkably, initial analysis shows that few source code changes are required to port the code to the XMT architecture. Within the existing framework of pDFS, each hardware thread can manage a private work pool, with work stealing used to load balance within a processor, much like the pthreads level of parallelism works on Cray XT. This operates more efficiently than pthreads due to the fine-grained synchronization primitives supported by the Cray XMT hardware. This can be extended seamlessly to multiple processors through the use of global memory, leveraging the simple Cray XMT parallel programming

mode, allowing it to replace both the pthread and MPI levels within pDFS.

7.2. Output

A substantial issue with enumeration is the volume of output. For many types of graphs, the number of maximal cliques found is large enough that the time to write the cliques to files is a significant portion of the overall run time. The technique currently implemented is to have each thread or each process open a unique output file. For a large job, this can overwhelm the file system. A scheme to best manage the output from a large parallel job remains to be devised. A likely direction is to group processes together for output, allowing an extra level of flexibility. For small runs, a single writer may suffice, but at larger scale, multiple writers will be needed. The optimal number of processes in a group is difficult to determine because different graphs will result in different volumes of output in an unpredictable manner. Ultimately, a dynamic scheme may be necessary, where additional output servers are added as needed.

8. Acknowledgements

This research has been supported by the "Exploratory Data Intensive Computing for Complex Biological Systems" project from U.S. Department of Energy (Office of Advanced Scientific Computing Research, Office of Science). The work of NFS was also sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory. Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E. under contract no. DEAC05-00OR22725.

The authors are thankful to Cray Inc. for access to large-scale Cray XT systems during the development and testing of this software.

9. References

[1] G. Brassard and P. Brately, Fundamentals of Algorithmics, *Prentice Hall,* 1996.

[2] C. Bron and J Kerbosch, Algorithm 157: finding all cliques of an undirected graph, *Comm. ACM*, vol 16, pp. 575-577, 1973.

[3] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn, Visualizing plant metabolomic correlation networks using cliquemetabolite matrices. *Bioinformatics*, vol 17, no. 12, pp. 1198-1208, 2001.

[4] J. W. Moon and L. Moser, On cliques in graphs. *Israel J. Math*, vol 3, pp. 23-28, 1965.

[5] B.H. Park, M. Schmidt, K. Thomas, T. Karpinets, and N.F. Samatova, Parallel, scalable, memory-efficient backtracking for combinatorial modeling of large-scale biological systems. Upcoming in *Proceedings of IPDPS 2008*

[6] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova, Genome-scale computational approaches to memory-intensive applications in systems biology, *The 2005 ACM/IEEE conference on Supercomputing*: IEEE Computer Society, 2005.