Early Experience with Out-of-Core Applications on the Cray XMT *

Daniel Chavarría-Miranda[1], Andrés Márquez[1], Jarek Nieplocha[2], Kristyn Maschhoff[4], and Chad Scherrer[3]

[1]High-Performance Computing
[2]Computational Sciences and Mathematics Division
[3]Computational Mathematics
Pacific Northwest National Laboratory
{daniel.chavarria, andres.marquez, jarek.nieplocha, chad.scherrer}@pnl.gov
[4]Cray, Inc. kristyn@cray.com

Abstract

This paper describes our early experiences with a preproduction Cray XMT system that implements a scalable shared memory architecture with hardware support for multithreading. Unlike its predecessor, the Cray MTA-2 that had very limited I/O capability, the Cray XMT offers Lustre, a scalable high-performance parallel filesystem. Therefore it enables development of out-of-core applications that can deal with very large data sets that otherwise would not fit in the system main memory. Our application performs statistically-based anomaly detection for categorical data that can be used for analysis of Internet traffic data. Experimental results indicate that the preproduction version of the machine is able to achieve good performance and scalability for the in- and out-of-core versions of the application.

1 Introduction

The increasing gap between memory and processor speeds has caused many applications to become memory-bound. That is, 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). Several hardware and software mechanisms have been proposed to increase the performance of such applications by reducing the exposed stall times seen by the processor. Most mainstream processors utilize a cache hierarchy, whereby small sections of high-speed memory hold data which has been recently fetched from main memory. Cache mechanisms are highly effective for applications that exhibit good temporal and spatial locality. However, many irregular applications do not exhibit such locality: their memory access patterns are essentially random. This is particularly true for data-intensive applications [5] that use large, pointer-linked data structures such as graphs and trees. Data-intensive applications focus on deriving scientific knowledge from vast repositories of empirical data. More traditional model-driven scientific applications focus on deriving scientific knowledge by obtaining precise numerical solutions to mathematical models.

Given the expected widening of the processor-memory speed gap, it becomes imperative to consider the use of alternative computer architectures for executing irregular, data-intensive applications. Multithreaded architectures, designed to tolerate memory access latencies by switching context between threads, offer a very appealing alternative for such applications. The processors on these machines 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. The Cray MTA processor [1] and the newer Cray Threadstorm [8], used on Cray's MTA-2 and XMT systems, respectively, are instances of modern multithreaded processors. Each one supports up to 128 threads with a single instruction pipeline and no data cache on the

*This work was funded by the U.S. Department of Energy's Pacific Northwest National Laboratory under the Data Intensive Computing Initiative. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

processor. Due to their memory latency tolerance, multithreaded platforms have the potential of significantly improving the execution speed of irregular data-intensive applications.

The Cray XMT system is a third generation multithreaded system from Cray. The XMT infrastructure is based on the Cray XT3/4 platform, including its high-speed interconnect and network topology, as well as service and I/O nodes. The difference is that the compute nodes of the XMT utilize a configuration based on 4 multithreaded Threadstorm processors instead of 4 AMD Opteron processors. The XMT system enables the execution of applications built entirely for the Threadstorm processors, in a similar manner as the MTA-2, as well as the execution of *hybrid* applications, in which portions of the application execute on the Threadstorm processors and portions execute on mainstream AMD Opteron processors. The two types of processing elements coordinate their execution and exchange data through a high-performance communications library named Lightweight User Communication (LUC).

Data-intensive applications have another very important component: very large I/O requirements. An effective way to address their massive I/O needs is through the use of high-performance parallel filesystems such as Lustre [2] and PVFS [19]. These parallel filesystems provide very large I/O bandwidth capabilities, in comparison to locally attached filesystems, through the use of multiple file servers that distribute the storage blocks for data files between them. In a High Performance Computing system, these file servers are usually attached through a high speed communications interconnect to the compute nodes of the systems, thus enabling high bandwidth, low latency, concurrent access to very large data sets.

In this paper, we present an early experience with a preproduction Cray XMT system using a data-intensive application with very large out-of-core datasets and complex, in-core irregular memory access patterns. Our application performs statistically-based anomaly detection for categorical data that can be used for Internet traffic analysis. The Cray XMT, unlike its predecessor the Cray MTA-2 which had very limited I/O capability, includes support for the powerful Lustre filesystem. The Lustre implementation in the Cray XMT implementation, similar to the Cray XT3/4, relies on the same high performance network with the Seastar [3] adapter and uses the Portals [17] messaging layer for data movement and synchronization. The combination of high-performance I/O subsystems with multithreaded processing capabilities was designed to provide a capable execution platform for irregular, data-intensive applications.

Section 2 provides background information about the Cray XMT and its hybrid architecture and programming model. Section 3 describes the Partial Dimensions Tree application and our parallel XMT implementation. Section 4 presents our experimental design and results. Section 5 discusses related work and finally, Section 6 presents our conclusions.

2 Background

The Cray XMT is the commercial name for the new multithreaded machine developed by Cray under the code name "Eldorado" [8]. By leveraging the existing platform of the XT3/4, Cray was able to save non-recurring development and engineering costs by reusing the support IT infrastructure and software, including dual-socket Opteron AMD service nodes, Seastar-2 high speed interconnects, fast Lustre storage cabinets and the associated Linux software stacks.

Changes were performed on the compute nodes by replacing the AMD Opterons with a custom designed multithreaded processor with a third generation MTA architecture that fits in XT3 motherboard processor sockets. Similar to the previous MTA incarnations, the Threadstorm processor schedules 128 fine-grained hardware streams to avoid pipeline stalls on a cycle-by-cycle basis. At runtime, a software thread is mapped to a stream comprised of a program counter, a status word, a target register and 32 general purpose registers. The MTA Long Instruction Word contains operations for the Memory functional unit, the Arithmetic unit and the control unit. The Arithmetic unit is capable of performing a multiply-add per cycle. In conjunction with the control unit doubling as arithmetic unit, a Threadstorm is capable of achieving 1.5 GFlop at a clock rate of 500MHz. As a reference, the previous MTA-2 processor ran at 220MHz. A 64KB, 4-way associative instruction cache helps in exploiting code locality.

Analogous to the Opteron memory subsystems, each Threadstorm is associated with a memory system that can accommodate up to 16GB of 128-bit wide DDR memory. Each memory module is complemented with a 128KB, 4-way associative data buffer to reduce access latencies. Memory is structured with full-empty-, pointer forwarding- and trap- bits to support fine grain thread synchronization with little overhead. As in the MTA-2, memory is hashed, albeit at a larger granularity. Continuous random accesses to memory will top memory bandwidth at around 100M requests per second. Up to 500M memory requests per second are delivered by the associated data buffers (Figure 1).

The Cray XMT uses the Seastar-2 network interconnect. Seastar-2 is a full system-on-chip design that integrates six high speed serial links, a 3-D router, with network interface functionality. The network interconnect includes an embedded PowerPC processor, in a single chip. The Seastar, initially, was designed to support message-passing applications on the Sandia Red Storm system and then Cray XT3/4. In the Seastar network, there are two DMA engines, one

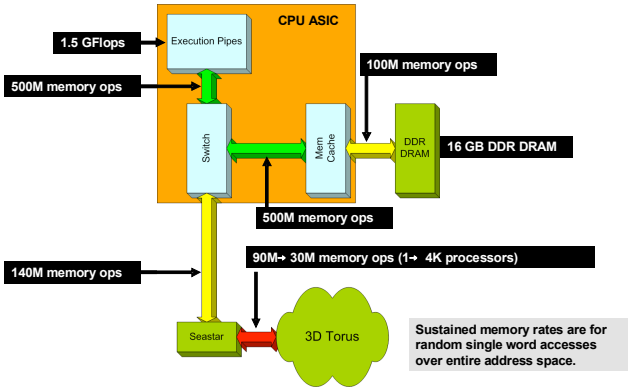

Figure 1. XMT Speeds

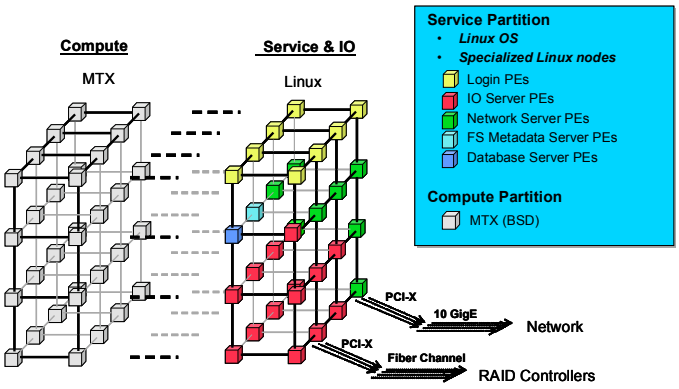

Figure 2. XMT System Architecture

for sending and the other for receiving, that interact with a router that supports a 3-D torus interconnect and the HyperTransport (HT) cave that provides an interface to the Cray Threadstorm processor and the host memory. The embedded processor is provided to program the DMA engines and assist with other network-level processing needs, in particular supporting the Portals message-passing layer of the Cray XT3/4. In the Cray XMT, the Seastar network adapter was modified to be able to support load/store operations through the MTX protocol. Bisection bandwidth between compute nodes ranges between 90M-30M memory requests per second for 3D topologies encompassing 1000-4000 processors. This contrasts sharply with the MTA-2 network that implements a modified Cayley graph with a bisection bandwidth that scales 3.5x with the number of processors. (Figure 2).

In the XMT, the Seastar network supports three communication interfaces: Portals for communication between service and storage nodes that are all based on the AMD Opteron processors, load/store operations between Threadstorm nodes, and the new Lightweight Communication Library (LUC) for communication between the Opteron and the Threadstorm nodes. On the Opteron side, LUC is implemented using Portals whereas on the Threadstorm side it uses Fast I/O API that is layered over the Seastar MTX protocol. LUC communication is based on the concept of endpoints. Endpoints facilitate connections between pairs of the service and compute nodes. From the software perspective, the LUC endpoints are implemented as objects that can be instantiated as server-only, client-only or both client and server objects which have their corresponding set of methods.

Another important element of the Cray XMT is the storage system. It is based on Lustre 1.6 version that has been also deployed in the Cray XT4. Lustre has been designed for scalability, supporting IO services to tens of thousands of concurrent clients. Lustre is a distributed parallel file system and presents a POSIX interface to its clients with parallel access capabilities to the shared objects. Lustre is composed of four main components: Metadata Server (MDS) providing metadata services; Object Storage Servers (OSSs) managing and providing access to underlying Object Storage Targets (OSTs); OSTs controlling and providing access to the actual physical block devices and data, and clients that access the data. The OSS/OST compartmentalization in Lustre provides increased scalability for large-scale computing platforms.

3 PDTree

The PDTree application originates in the cyber security domain, and involves large sets of network traffic data. Analysis is performed to detect anomalies in network traffic packet headers in order to locate and characterize network attacks, and to help predict and mitigate future attacks. This application is a special case of a more widely-applicable analysis method which uses ideas from conditional probability in conjunction with a novel data structure and algorithm to find relationships and patterns in the data [16].

When dealing with multivariate categorical data we can ask, for any combination of variables and instantiation of values for those variables, how many times this pattern has occurred. Because multiple variables are being considered simultaneously, the resulting count table, or contingency table, specifies a joint distribution. Contingency tables are a key component of a wide variety of analysis methods for discrete data. Efficient algorithms using such tables are critical for implementation of Bayesian networks, log-linear models, Markov random fields, various graph representa-

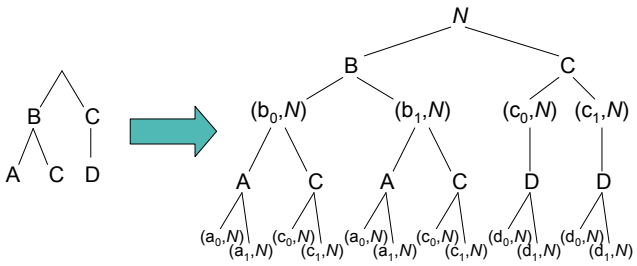

Figure 3. Guide tree and corresponding PDTree

tions, and novel algebraic-based approaches [15]. This is especially important when there are a large number of variables, a large number of observations, or when some variables take on many distinct values. All of these hold with regard to the massive data prevalent in cyber security analysis.

Moore and Lee's ADtree data structure, described in [12], makes count queries significantly more efficient. In this data structure, the root of the tree contains the total count. At each step down into the tree, another variable is instantiated with a particular value, with counts tracked at each level. ADtrees take advantage of the sparsity that typically results when many variables are simultaneously instantiated, and they allow fast queries: any count query can be answered in a number of steps proportional to the number of variables instantiated in the query. However, they can be quite expensive to populate; memory usage and computation time are both at best exponential, since each is at best linear in the total number of combinations of variables.

Fortunately, in many cases, we need not store counts for every such combination. This may be due to the availability of a statistical model for the data, or to requirements involving computation time or memory. In such cases, limiting the data structure to reflect this can result in a corresponding increase in performance. Because this differs from an ADtree primarily in that we no longer store *all dimensions*, but only *partial dimensions*, we refer to this as a *PDTree*. By specifying a priori which combinations of variables are to be stored, we reduce the number of steps required to traverse the PDTree each time a new record is inserted. The nested structure of the variables is specified in an auxiliary data structure called a *guide tree*.

As an example, suppose we have variables $\{A, B, C, D\}$ and that we wish to fit the Markov chain $A \rightarrow B \rightarrow C \rightarrow D$. This does not require storage of counts for every combination of variables, but only for the set $\{AB, B, BC, C, CD\}$. Figure 3 shows a guide tree that represents these combinations, and the corresponding PDTree. For simplicity, we have left off some subscripts: each "N" in this figure represents a count corresponding to the appropriate instantiation of variables.

The most time- and memory-intensive step of using a PDTree for a given data set is populating it. For this reason, we have based our study of PDTree performance entirely on the population step. To populate a PDTree for a given model and data, we follow these steps:

1. Determine which combinations of variables must be stored for the model.
2. Construct a guide tree representing the required combinations.
3. Begin with an empty PDTree (only the root node, with a zero count).
4. For each record, traverse the guide tree, which at each step specifies how the PDTree is to be traversed. Increment counts or insert new nodes as needed.

Note that once a PDTree is built, the guide tree is not needed for queries; all required information about the variables is contained within the PDTree itself. Thus the guide tree can be represented in any form that allows efficient traversal, such as a list-of-lists or similar structure, but we need not be concerned with random lookup efficiency of the guide tree.

3.1 Multithreaded Implementation

The keys to a highly scalable, efficient PDTree algorithm on the XMT are:

1. a scalable data structure that supports increasing numbers of insertions, and
2. an insertion operation that is safe, concurrent, and minimizes synchronization costs.

The XMT's shared memory and tolerance for highly irregular memory access patterns gives the programmer great latitude in choice of data structure. One can choose the data structure that best fits the problem rather than try to force the problem into a data structure that best fits the architecture.

On the XMT, the PDTree is a multiple type, recursive tree structure. A root node (one root node per column) is a collection of ValueNodes. Interior and leaf nodes are linked lists of ValueNodes. Since a root is just a histogram of column values, its size and contents are easy to set. Our original implementation used a linked list at the top level; but, as we explain below, it suffered from high synchronization costs and did not scale past eight processors. Inserting a record at the top level requires only that we increment the

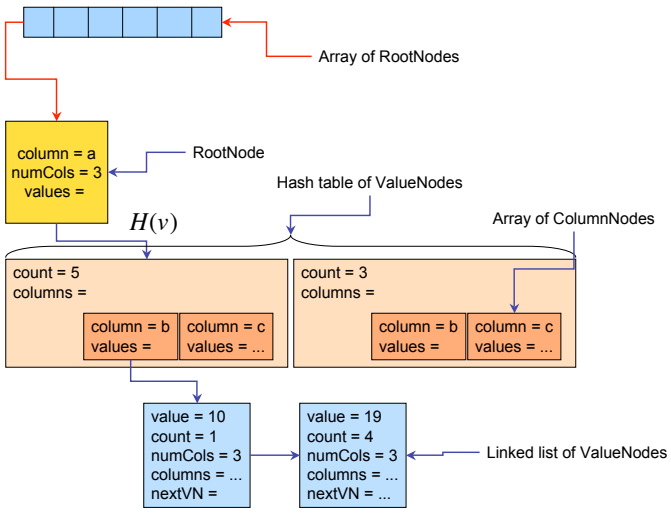

Figure 4. Dynamic PDTree implementation

counter of the right ValueNode. Inserting the record at other levels of the tree requires us to traverse a linked list to find the right ValueNode. If we find the node, we increment its counter. If we do not find the node, we add the node to the end of the list and set its counter to 1. We use the XMT's atomic `int_fetch_add` operation to increment counters. This operation is performed in memory and costs only one instruction cycle.

Inserting the record at other levels of the tree when no node is found is trickier. To insure safety, a thread must lock the list's end pointer before inserting a new node. Implementing a critical section is easy on the XMT using the synchronized read and write operations, `readfe` and `writeef`, respectively.

In our implementation, a critical section exists only at the end of a list. An insertion operation waits only if it tries to insert a node at the same time at the end of the same list as another insertion operation. Since the PDTree grows quickly into a massive, sparse structure, the probability of two threads colliding quickly approaches zero. Moreover, even if an insertion waits, the Threadstorm processor on which it executes does not wait; the processor merely switches to another thread and continues executing instructions. Note that the reason that we do not want to use a linked list at the top level is that all records are inserted in the same top level structure; thus, the probability of two insertions colliding is significant.

More details on the statistical method and theory behind the PDTree application and its parallel implementation can be found in [16] and [13].

3.2 Static & Dynamic Implementations

In some instances, it might be possible to have information ahead of time on the possible range of values for a column: i.e. IP addresses corresponding to certain subnetworks. In this case, it is efficient to use a dense array of ValueNodes at the RootNode level to store the value data corresponding to a column. The access time to any entry in the array has minimal constant cost, at the expense of possible wasted memory space if the values in the actual dataset are sparse with respect to their range.

In other cases, the range of values for a column can be very large or unknown at execution time, this is particularly true when performing analysis of streaming data coming from network sensors and other dynamic sources. In this case, it is necessary to use a more flexible data structure for the ValueNode collection at the RootNode level. We use a C++ template-based hash table implementation with a number of entries that we have found to be balanced between memory usage and collision likelihood for each column at the RootNode level.

Figure 4 illustrates the details of the concrete data structure implementation we use for the PDTree. The top level corresponds to an array of RootNode structures, one for each column described in the guide tree. Each individual RootNode contains a hash table of ValueNodes, on which individual value instances for that column will be stored. We use Wang's 64-bit mix hash function [18] ($H(v)$ in Figure 4), which is based on Knuth's multiplicative method [9]. For the dense array case, each RootNode will have a dense array of as many entries as different values are expected for each column, in place of the hash table.

We have implemented the PDTree application using both data structure styles. Section 4 describes in detail the experimental results obtained with both data structure approaches.

4 Experimental Setup and Results

As described in Section 2, the XMT supports many standard HPC capabilities present in mainstream supercomputers like the Cray XT3/4, including access to high-performance parallel filesystems such as Lustre. However, many of this mainstream HPC capabilities are not directly accessible from the Threadstorm compute nodes due to their unique custom hardware and available operating system.

Cray has provided an indirect mechanism to provide access to standard HPC capabilities from the Threadstorm compute nodes via the Lightweight User Communication Library (LUC) [7]. LUC offers an RPC-like mechanism that enables the Threadstorm processors to remotely execute procedures on service Opteron processors and vicevversa. The RPC mechanism enables data transfers between

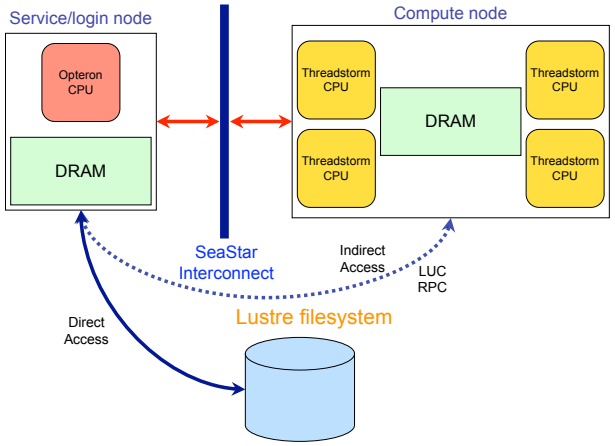

Figure 5. Indirect access to Lustre filesystem from Threadstorm processors (via LUC RPC)

the Opteron and Threadstorm nodes via function call parameters. These parameters can be modified (when passed via C-style pointers) and their modification will be reflected in the originating side of the RPC call. In this manner, it is possible to design a LUC-based protocol to enable indirect reading and/or writing (by the Threadstorm nodes) of files resident on a Lustre filesystem accessible only to the Opteron nodes. Figure 5 illustrates this indirect access concept. The operating system (MTK) on the Threadstorm compute nodes does provide an NFS filesystem client, which can directly access files on an NFS server. However, the performance of direct file access by the Threadstorm processors is not comparable to that of indirect access via LUC.

The PDTree application reads its input data from a file as a proxy for a streaming data source (i.e. network sensor or router/switch). The PDTree input data consists of variable-length ASCII records, separated by carriage returns. Each line consists of column data separated by space characters.

We have extended our original PDTree implementation to handle very large datasets on the Cray XMT by using its hybrid execution capabilities via a LUC-based RPC mechanism. We have developed a LUC server to execute on the Opteron service nodes, which have direct access to a large Lustre filesystem. The server opens a very large file, with PDTree data, resident on the Lustre filesystem. The server then proceeds to stream chunks of the file, in response to requests from a Threadstorm LUC client. The server reads the file in binary mode in fixed size chunks and sends the data to the Threadstorm client after discarding incomplete records that might appear at the end of the read chunk. Each one of the read chunks is then processed as a record set and inserted into its corresponding place in the PDTree. This process closely follows the expected use of a PDTree-like application for anomaly detection in a realistic networking environment with streaming data coming from network routers and sensors.

In order to handle this dynamic input data, we use the hash table-based PDTree implementation described in Section 3.2, which does not have assumptions with respect to the range of values expected in the input records. We also executed the static dense array-based PDTree implementation for a smaller dataset in order to have a baseline for the possible overhead of the dynamic hash table-based implementation.

4.1 Results

Our experiment uses a 4GB PDTree data input file with 64 million records, we use a template with 9 columns (guide tree) resident on the described Lustre filesystem. We stream five chunks of data using three different chunk sizes: 100 MB, 200 MB and 250 MB. The chunk sizes correspond to increasing numbers of ASCII records.

The ASCII data is then converted to binary records in a parallel preprocessing step. The binary records are then inserted into the PDTree using the general, hash-based dynamic scheme described in Section 3.2. We present results obtained for different numbers of processors on the XMT system. We include the time for the aggregated data transfer through LUC, the preprocessing time (conversion from ASCII to binary), the time to insert the data into the PDTree and the estimated speedup (for the insertion portion only) obtained with respect to execution on a single Threadstorm processor. For the dynamic streaming execution, due to time limitations, we were not able to execute the experiments on processor configurations below 8, for this reason we assume that execution on 8 processors has perfect speedup (8.0) and compute the other speedups relative to the 8 processor execution time (the static results on smaller number of processors lead us to believe that this is the case).

We use a hash table size of 128 K elements for each top-level column in the PDTree and our preliminary experiments have indicated that there are no hash collisions for our chosen dataset. In fact, a few hash collisions for a given dataset could be interpreted as not affecting the overall, aggregate properties of the PDTree. However, if hash collisions are very frequent then a particular column of the PDTree will have the aggregate data for the colliding values, leading to misleading results.

Table 1 presents the results of our static setup (dense arrays at the RootNode level) for a 1 million record input file, for comparison purposes. All times are in seconds. The PDTree data insertion time is scaling linearly with the number of Threadstorm processors up to 96, indicating that the original dense array-based approach for the top level of the

# of Procs.	XMT Insertion	XMT Speedup	MTA Insertion	MTA Speedup
1	239.26	1.00	200.17	1.0
2	116.36	2.06	98.25	2.04
4	56.48	4.24	48.07	4.16
8	27.53	8.69	23.29	8.59
16	13.97	17.13	11.61	17.24
32	7.13	33.56	5.81	34.45
64	3.68	65.02	N/A	N/A
96	2.60	92.02	N/A	N/A

Table 1. Performance results for an in-core, static 1,000,000 record execution

# of Procs.	LUC Transfer	Preprocessing	Insertion	Speedup
8	3.25	17.85	229.79	8.00
16	3.26	9.22	114.92	16.00
32	3.28	4.95	58.29	31.52
64	3.28	3.13	30.79	59.68
96	3.23	2.78	24.14	76.16

Table 2. Performance results for 100 MB ($\approx 1,750,000$ records, 500 MB transferred) dynamic streaming execution

# of Procs.	LUC Transfer	Preprocessing	Insertion	Speedup
8	6.35	35.46	449.88	8.00
16	6.35	18.16	226.56	15.89
32	6.41	9.56	115.01	31.29
64	6.35	5.57	60.72	59.27
96	6.38	4.53	46.53	77.35

Table 3. Performance results for 200 MB ($\approx 3,250,000$ records, 1 GB transferred) dynamic streaming execution

# of Procs.	LUC Transfer	Preprocessing	Insertion	Speedup
8	7.93	44.70	572.05	8.00
16	7.86	22.47	283.86	16.16
32	7.95	11.77	143.62	31.84
64	7.98	6.66	76.02	60.16
96	7.93	5.33	54.77	83.52

Table 4. Performance results for 250 MB ($\approx 4,100,000$ records, 1.25 GB transferred) dynamic streaming execution

PDTree is scalable and that there is a diminishing probability of collisions when creating the interior nodes and leaves of the tree. These experiments were executed using a preproduction configuration with the 4-way set associative memory buffer enabled and only half of the DIMM slots populated on each compute node, which limits memory bandwidth. We also have included absolute times and speedups for the same setup executing on the original MTA-2 system on up to 32 processors.

Tables 2, 3 and 4 present the results of our dynamic experiment for chunk sizes of 100 MB, 200 MB and 250 MB. All times are in seconds. The dynamic experiments were executed using a preproduction setup with the memory buffer using a 1-way set associative configuration, as well as only half the DIMM slots populated.

The LUC transfer times correspond to a single LUC server thread (single endpoint) reading the chunk from the Lustre file system accessible from an Opteron service node, then transferring the read data over the Portals interface to the Threadstorm compute nodes. This simple mechanism can be optimized significantly to provide much better throughput by using multiple concurrent LUC requests through as many LUC endpoints. However, the performance of this simple single-endpoint access scheme is superior to direct access from the Threadstorm processor to input files resident on NFS: for the static version of the application the time to read the 1 million record input file directly by the Threadstorm processors is approximately 350 seconds. The indirect access times through LUC are much smaller and represent only a small fraction of the total processing (insertion) time.

5 Related Work

There is a large body of work on the use of out-of-core methods for numerical problems [14, 10]. which focus on maintaining high numerical throughput in spite of datasets which do not fit into the main memory of the processors. Many papers also address the requirements for processing massive datasets for computer graphics and visualization [6, 4].

McMains, Hellerstein and Séquin [11] propose a technique to build high-level topological information from an unordered, out-of-core set of polygonal data for geometrical modeling. Their work is similar to our approach in that their technique is building a higher-level, abstract representation of the low-level data present in the out-of-core dataset.

6 Conclusions

We have presented some early results obtained with an out-of-core, data-intensive application on a preproduction Cray XMT system. Our experience indicates the value of

the XMT's hybrid architecture and its improved I/O capabilities over the predecessor system, the Cray MTA-2. The hybrid architecture provides user with ability to select what portions of the application should execute on the multithreaded processors versus the mainstream processors. We have used the new Cray LUC communication interface to transfer the out-of-core data resident on a high-performance Lustre filesystem to the multithreaded part of the machine, which does not directly support access to Lustre. Our implementation of the I/O operations based on LUC has not been yet optimized to take the full advantage of the high-performance and scalability Lustre offers. However, it enabled us to achieve good scaling for the out-of-core version of the PDTree application.

The experimental results indicate that the overall scalability of the system is very good, in the presence of complex, irregular, data-dependent access in-core access patterns and large out-of-core block transfers. We plan to optimize our out-of-core strategy by implementing multiple LUC endpoints to take advantage of the parallel transfer capabilities of the LUC protocol and the parallel Lustre filesystem.

References

[1] W. Anderson, P. Briggs, C. S. Hellberg, D. W. Hess, A. Khokhlov, M. Lanzagorta, and R. Rosenberg. Early Experience with Scientific Programs on the Cray MTA-2. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 46, Washington, DC, USA, 2003. IEEE Computer Society.

[2] P. J. Braam. Lustre: A Scalable, High Performance File System. `http://www.lustre.org/docs/whitepaper.pdf`, 2002.

[3] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson. Seastar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, 2006.

[4] K. Cai, Y. Liu, W. Wang, H. Sun, and E. Wu. Progressive out-of-core compression based on multi-level adaptive octree. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 83–89, New York, NY, USA, 2006. ACM.

[5] M. Cannataro, D. Talia, and P. K. Srimani. Parallel data intensive computing in scientific and commercial applications. *Parallel Comput.*, 28(5):673–704, 2002.

[6] Y.-J. Chiang, R. Farias, C. T. Silva, and B. Wei. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 59–66, Piscataway, NJ, USA, 2001. IEEE Press.

[7] Cray Inc. Cray XMT Lightweight User Communication Library (LUC) User's Guide. Draft version, August 2007.

[8] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM.

[9] D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, Reading, MA, Second edition, 1981.

[10] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Softw.*, 12(3):249–264, 1986.

[11] S. McMains, J. M. Hellerstein, and C. H. Séquin. Out-of-core build of a topological data structure from polygon soup. In *SMA '01: Proceedings of the sixth ACM symposium on Solid modeling and applications*, pages 171–182, New York, NY, USA, 2001. ACM.

[12] A. Moore and M. S. Lee. Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets. *Journal of Artificial Intelligence Research*, 8, 1998.

[13] J. Nieplocha, A. Márquez, J. Feo, D. Chavarría-Miranda, G. Chin, C. Scherrer, and N. Beagley. Evaluating the Potential of Multithreaded Platforms for Irregular Scientific Computations. In *CF '07: Proceedings of the 4th International Conference on Computing frontiers*, pages 47–58, New York, NY, USA, 2007. ACM Press.

[14] V. Rotkin and S. Toledo. The design and implementation of a new out-of-core sparse cholesky factorization method. *ACM Trans. Math. Softw.*, 30(1):19–46, 2004.

[15] C. Scherrer and N. Beagley. Conditional Independence Modeling for Categorical Anomaly Detection. In *Proc. Joint Annual Meeting of the Interface and Classification Society of North America*, 2005.

[16] C. Scherrer, N. Beagley, J. Nieplocha, A. Márquez, J. Feo, and D. Chavarría-Miranda. Probability convergence in a multithreaded counting application. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–5, 26-30 March 2007.

[17] R. B. Trammell. Portals 3.3 on the Sandia/Cray Red Storm System. In *Cray User Group*, 2005.

[18] T. Wang. 64-bit Mix Function. `http://www.concentric.net/~Twang/tech/inthash.htm`, 1997.

[19] W. Yu, S. Liang, and D. K. Panda. High performance support of parallel virtual file system (PVFS2) over Quadrics. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 323–331, New York, NY, USA, 2005. ACM.