Novel Graph Processor Architecture

William S. Song, Jeremy Kepner, Vitaliy Gleyzer,

Huy T. Nguyen, and Joshua I. Kramer

Graph algorithms are increasingly used in applications that exploit large databases. However, conventional processor architectures are hard-pressed to handle the throughput and memory requirements of graph computation. Lincoln Laboratory's graph-processor architecture represents a fundamental rethinking of architectures. It utilizes innovations that include high-bandwidth three-dimensional (3D) communication links, a sparse matrix-based graph instruction set, accelerator-based architecture, a systolic sorter, randomized communications, a cacheless memory system, and 3D packaging.

Many problems in computation and data analysis can be framed by graphs and solved with graph algorithms. A graph, which is defined as a set of vertices connected by edges, as shown on the left in Figure 1, adapts well to presenting data and relationships. Graphs take two forms: a directed graph has edges with orientation as shown in Figure 1, and an undirected graph has edges with no orientation. Graph algorithms perform operations on graphs to yield desired information. In general, graphs can also be represented as full standard and sparse matrices as shown in Figure 1 [1, 2]. The graph $G(V, E)$ with vertices V and edges E can be represented with the sparse matrix $\mathbf{A}$ where the matrix element $\mathbf{A}_{ij}$ represents the edge between the vertex i and vertex j. In this example, $\mathbf{A}_{ij}$ is set to 1 when there is an edge from vertex i to vertex j. If there is no edge between vertices i and j, then $\mathbf{A}_{ij}$ would be zero and thus would have no entry in the sparse matrix. In this example, the sparse matrix has reduced the required data points representing the graph from 16 to 7.

Increasingly, commercial and government applications are making use of graph algorithms [3]. These applications address a wide variety of tasks—finding the shortest or fastest routes on maps, routing robots, analyzing DNA, corporate scheduling, transaction processing, and analyzing social networks—as well as network optimizations for communication, transportation, water supply, electricity, and traffic.

Some of the graph algorithm applications involve analyzing very large databases. These large databases could contain consumer purchasing patterns, financial transactions, social networking patterns, financial market infor-

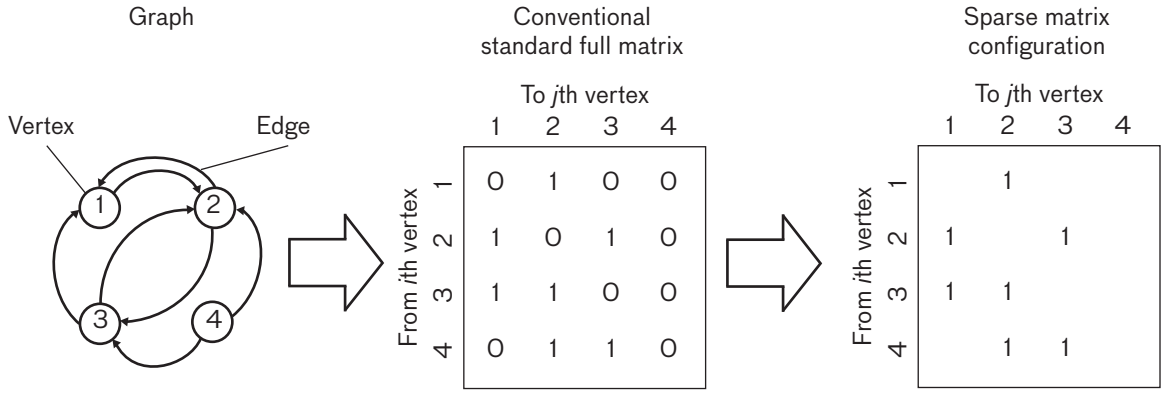


FIGURE 1. A sparse matrix representation of a graph reduces the amount of computation power necessary by representing the graph with a minimum (sparse) number of data points.

mation, Internet data, test data, biological data, cyber communication, or intelligence, surveillance, and reconnaissance (ISR) sensor data. For example, an analyst might be interested in spotting a cyber attack, locating a terrorist cell, or identifying a market niche. Some graphs may contain billions of vertices and edges requiring petabytes of memory for storage. For these large database applications, computation efficiency often falls off dramatically. Because conventional cache-based processor architectures are generally not well matched to the flow of the graph computation, it is impossible for computation hardware to keep up with computation throughput requirements. For example, most modern processors utilize cache-based memory in order to take advantage of highly localized memory access patterns. However, memory access patterns associated with graph processing are often random in nature and can result in high cache miss rates. In addition, graph algorithms require significant overhead computation for dealing with the indices of vertices and edges of graphs.

For benchmarking graph computation, we often use sparse matrix operations for estimating graph algorithm performance because sparse matrix arithmetic operations have computational flow and throughput very similar to the flow and throughput of graph processing. Once the graphs have been converted to the sparse matrix format, the sparse matrix operations can be used to implement most graph algorithms. Figure 2 shows an example of the computational throughput differences between conventional processing and graph processing [4]. Shown in blue is a conventional matrix multiply kernel running on PowerPC and Intel Zeon processors. In contrast, shown in red is a sparse matrix multiply kernel running on identical processors. As one can see, the graph computation throughput is approximately 1000 times lower; this result is consistent with typical application codes.

Recently, multiple processor cores have become available on a single processor die. Multicore processors can speed up graph computation somewhat, but are still limited by conventional architectures that are optimized essentially for dense matrix processing using cache-based memory.

Parallel processors have often been used to speed up large conventional computing tasks. A parallel processor generally consists of conventional multicore processors that are connected through a communication network so that different portions of the computations can be done on different processors. For many scientific computing applications, these processors provide significant speedup over a single processor.

However, large graph processing tasks often run inefficiently on conventional parallel processors. The speedup often levels off after only a small number of processors are utilized (Figure 3) because the computing patterns for graph algorithms require much more communication between processor nodes than conventional, highly localized processing requires. The limited communication bandwidth of conventional parallel processors generally cannot keep pace with the demands of graph algorithms.

In the past, numerous attempts have been made to speed up graph computations by optimizing processor architecture. Parallel processors such as Cray XMT and Thinking Machine's Connection Machine are example attempts to speed up large graph processing with spe-

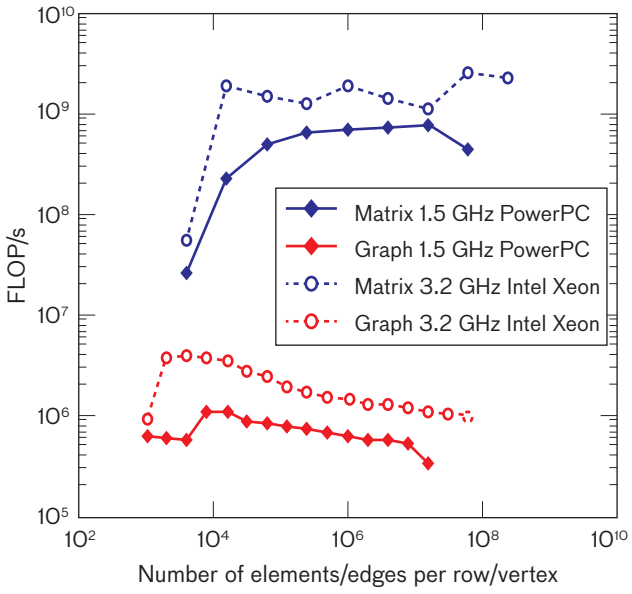


FIGURE 2. A comparison of computational throughput differences between conventional and graph processing shows that in conventional processors computational efficiency is significantly lower for graph processing compared to conventional processing.

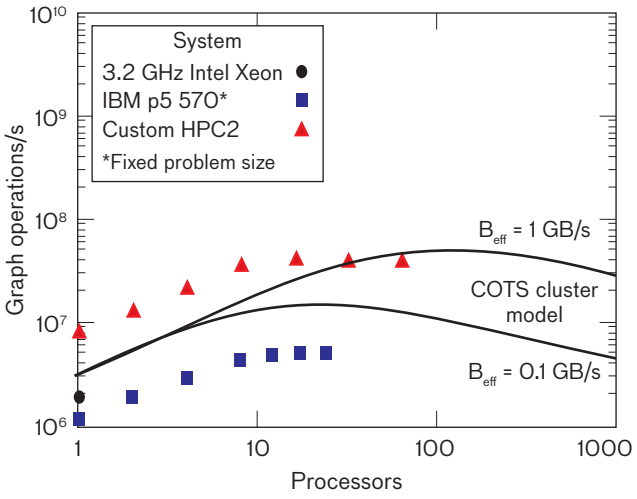


FIGURE 3. Graph processing computational throughput in networked multiprocessors levels off at the use of a relatively small number of processors.

cialized parallel architectures. However, inherent difficulties associated with graph processing, including distributed memory access, indices-related computation, and interprocessor communications, have limited the performance gains.

Lincoln Laboratory has been developing a promising new processor architecture that may deliver orders of magnitude higher computational throughput and power efficiency over the best commercial alternatives for large graph problems.

Graph Processor

The Laboratory's new graph processor architecture represents a fundamental rethinking of the computer architecture for optimizing graph processing. The instruction set is unique in that it is based on and optimized for sparse matrix operations. In addition, the instruction set is designed to operate on sparse matrix data distributed over multiple processors. The individual processor node—an architecture that is a great departure from the conventional von Neumann architecture—has local cacheless memory. All data computations, indices-related computations, and memory operations are handled by specialized accelerator modules rather than by the central processing unit (CPU). The processor nodes utilize new, efficient message-routing algorithms that are statistically optimized for communicating very small packets of data such as sparse matrix elements or partial products. The processor hardware design is also optimized for very-high-bandwidth three-dimensional (3D) communications. Detailed analysis and simulations have demonstrated an orders-of-magnitude increase in computational throughput and power efficiency for running complex graph algorithms on large distributed databases.

Parallel Graph Processor Architecture Based on a Sparse Matrix Algebra Instruction Set

Assume that the graph has been converted into sparse matrix format before being inputted into the processor. The sparse matrix opervations are then used to implement the graph algorithms. There are a number of advantages in implementing the graph algorithms as sparse matrix operations. One advantage is that the number of lines of code is significantly reduced in comparison to the amount of code required by traditional software that directly implements graph algorithms using conventional instruction sets. However, while this advantage can increase software development efficiency, it does not necessarily result in higher computational throughput in conventional processors.

Perhaps a more important advantage of embedding graph algorithms in sparse matrix operations is that it is much easier to design a parallel processor that computes

sparse matrix operations rather than general graph algorithms. The instruction set can be vastly simplified because implementing sparse matrix–based graph algorithms requires surprisingly few base instructions. Another reason sparse matrix operations facilitate the designing of a processor architecture is that it is much easier to visualize the parallel computation and data movement of sparse matrix operations running on parallel processors than it is on conventional machines. This advantage enables developers to come up with highly efficient architectures and hardware designs with much less effort.

Lincoln Laboratory's new graph processor is a highly specialized parallel processor optimized for distributed sparse matrix operations. The processor is targeted for implementing graph algorithms (converted to sparse matrix format) for analyzing large databases. Because large matrices do not fit into a single processor's memory and require more throughput than the single processor can provide, the approach is to distribute the large matrices over many processor nodes. Figure 4 shows the high-level architecture of the parallel processor. It consists of an array of specialized sparse matrix processors called node processors. The node processors are attached to the global communication network, and they are also attached to the global control processor through the global control bus.

Although the generic high-level architecture in Figure 4 appears quite similar to that of a conventional multiprocessor system, how it is implemented is significantly different from how a conventional parallel architecture is implemented. One of the main differences is that the processor's instruction set is based on sparse matrix algebra operations [2] rather than on conventional instruction sets. Important instruction kernels include sparse matrix multiply, addition, subtraction, and division operations shown in Table 1. Individual element-level operators within these matrix operations, such as multiply and accumulate operators in the matrix-multiply operation, often need to be replaced with other arithmetic or logical operators, such as maximum, minimum, AND, OR, XOR, etc., in order to implement general graph algorithms. Numerous graph algorithms have already been converted to sparse matrix algorithms [2, 4].

The other main differentiating feature of the new architecture is the high-bandwidth, low-power communication network that is tailored for communicating small

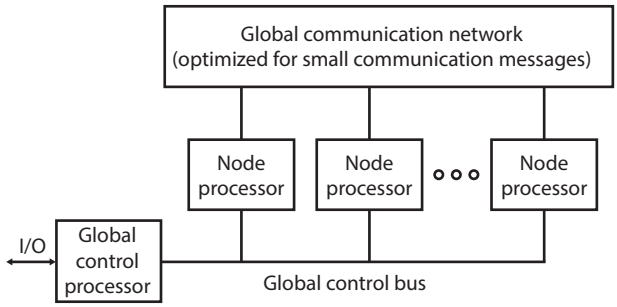


FIGURE 4. The illustration of the high-level architecture for Lincoln Laboratory's parallel graph processor shows the connection between the specialized sparse matrix processors (node processors) and the global components.

Table 1: Sparse Matrix Algebra-Based Processor Instruction Set

OPERATION	COMMENTS
C = A +.* B	Matrix multiply operation is the throughput driver for many important benchmark graph algorithms. Processor architecture is highly optimized for this operation.
C = A .± B **C = A .* B** **C = A ./ B**	Dot operations are performed within local memory.
B = op(k, A)	Operation with matrix and constant. This operation can also be used to redistribute matrix and sum columns or rows.

messages. A typical message contains one matrix element or one partial product, which consists of the data value, row index, and column index. In contrast, a conventional communication network tries to maximize the message sizes in order to minimize the overhead associated with moving the data. A newly developed statistical routing algorithm with small message sizes greatly improves the communication-bandwidth availability for graph processing. In addition, the bandwidth of the network hardware itself is very large compared to the bandwidth of conventional parallel processors; this large bandwidth is needed to handle the demands of graph processing.

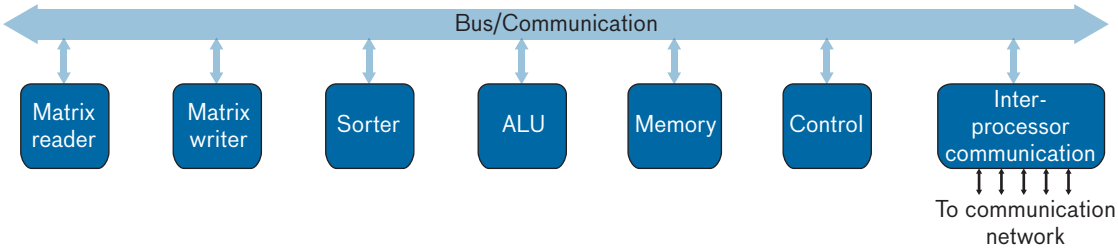


FIGURE 5. The new node processor architecture uses specialized modules to speed up sparse matrix processing. (ALU stands for arithmetic logic unit.)

Accelerator-Based Node Processor Architecture

The architecture of the Laboratory's individual node processor is also a great departure from conventional cache-based von Neumann machines, which perform all computations in the CPU. This new architecture consists of a number of specialized modules, including matrix reader, matrix writer, sorter, arithmetic logic unit (ALU), and communication modules, as shown in Figure 5 [4, 5]. The CPU is mainly used to provide the control and timing for the sparse matrix instructions. Most of the computation, communication, and memory operations are performed by the specialized modules that are designed to optimally perform the given tasks. There is no cache because the cache misses tend to slow down graph processing. In general, multiple modules are utilized simultaneously in performing sparse matrix computations.

The architecture based on the specialized accelerator module provides much higher computational throughput than the conventional von Neumann processor architecture by enabling highly parallel pipelined computations. In a conventional processor, the microprocessor is used to compute all the processing tasks, such as memory access, communication-related processing, arithmetic and logical operations, and control. These processing tasks are often done serially and take many clock cycles to perform, lowering the overall computational throughput. In the new architecture, these tasks are performed in parallel by separate specialized accelerator modules. These accelerator modules are designed for fast throughput using highly customized architectures. Ideally, they would be designed to keep up with the fastest data rate possible, which is processing one matrix element or one partial product within a single clock cycle in effective throughput. Further speedup may be gained by having multiple parallel versions of these modules to process multiple matrix elements or partial products per clock cycle.

The matrix reader and writer modules are designed to efficiently read and write the matrix data from the memory. The example formats include compressed sparse row (CSR), compressed sparse column (CSC), and coordinate (also called tuple) format (Figure 6). In the CSR format, the element data and column index are stored as pairs in an array format. An additional array stores the row start address for each column so that these pointers can be used to look up the memory locations in which the rows are stored. In the CSC format, the element data and row index are stored as pairs in an array format. An additional CSC array stores the column start address for each row. The coordinate format stores matrix element–related data, including element data, row index, and column index together in array format. The coordinate format is also convenient in storing randomly ordered matrix elements or partial products. The matrix reader and writer modules are designed so that all the overhead operations—such as formatting matrix element data and indices for writing, generating pointer arrays for CSC and CSR for writing, and generating matrix element indices for reading—are performed automatically without requiring additional instructions. In this way, complexity associated with sparse matrix read and write operations is minimized, and memory interface operations are accelerated significantly.

The ALU module is designed to operate on the stream of sparse matrix elements or partial products instead of operating with a register file as in conventional processor architectures. The streaming method eliminates register load operations and increases the computational throughput. It generally performs designated arithmetic or logical operations on the data stream, depending on the indices. For example, the ALU

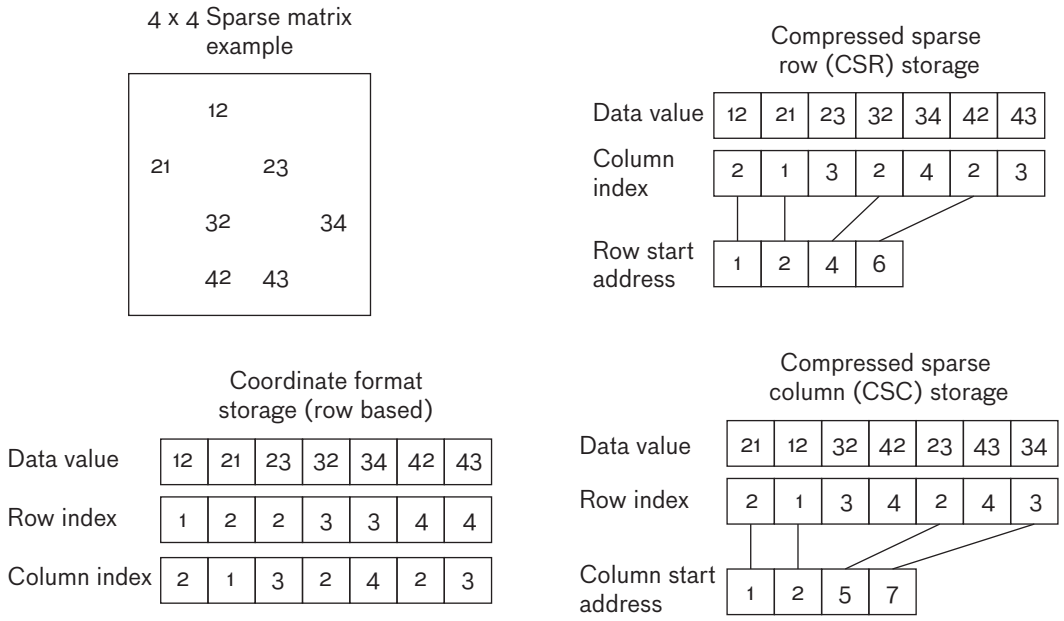


FIGURE 6. Three formats for sparse matrix storage are shown for the 4 × 4 sparse matrix example in the figure.

module may accumulate successive matrix elements only if the element indices match exactly. Because these matrix operations perform computations only when the indices match, this feature is useful for sparse-matrix add and multiply operations.

The communications module handles the communication between processor nodes. It takes the matrix element or partial product and makes a communication message that includes the matrix element in coordinate format and a header that contains the destination processor address. The header may also contain error detection and correction bits and other relevant information, such as the priority of the message. The communication messages are then sent to the global communication network and are forwarded to the destination nodes. The communications module also decodes the received messages, performs error correction, and outputs the matrix element or partial product into the node in coordinate format.

The memory for the node processor can be implemented with various types of memory including static random-access memory (SRAM), dynamic RAM, and synchronous DRAM. Nonvolatile memory such as Flash memory may be used for long-term storage and for instances when the storage requirement is high. There is no cache in the memory system since cache miss rates tend to be very high in graph processing.

The node controller module is responsible for setting up and coordinating the sparse matrix operations. Before a sparse matrix operation, the controller module loads the control variables into the control registers and control memory of the accelerator modules by using the local control bus. The control variables include types of sparse matrix operations to be performed, matrix memory storage locations, matrix distribution mapping, and other relevant information. The controller module also performs timing and control. The node controller module can be implemented with a conventional general-purpose microprocessor. This particular microprocessor may also have a cache since the processing is mostly conventional processing. The node controller can also perform other processing tasks that are not well supported by the accelerator modules, such as creating an identity matrix and checking to see if a matrix is empty across all processor nodes. The controller module is tied to the global control bus, which is used to load the data and programs to and from the nodes, and to perform the global computation process control.

Systolic Merge Sorter

The systolic merge sorter module is used for sorting the matrix element indices for storage and for finding matching element indices during matrix operations. It is one of the most critical modules in graph processing

because more than 95% of computational throughput can be associated with the sorting of indices. The sparse matrix and graph operations consist mainly of figuring out which element or partial product should be operated on. In contrast, relatively few actual element-level operations get performed. In order to meet the computational throughput requirement, the systolic k-way merge sorter architecture [4] was developed to provide significantly higher throughput than the conventional merge sorters.

The conventional merge sorter sorts long sequences of numbers by using a recursive "divide and conquer" approach. It divides the sequence into two sequences that have equal, or as equal as possible, lengths. The two shorter sequences are then sorted independently and merged to produce the sorted result. The sorting of two shorter sequences can also be divided into even shorter sequences and sorted recursively by using the same merge sort algorithm. This process is recursively repeated until the divided sequence length reaches 1. Figure 7 illustrates an example of the 2-way merge sorting in which 16 items are sorted in four steps. The merge sort algorithm requires order of $n\log_2 n$ processor cycles and order of $2n$ locations in memory, where n is the length of the sequence. The merge sort algorithm can be readily implemented with a conventional general-purpose processor or a digital signal processor working with random-access memory.

The k-way merge sorter can perform the sorting task faster than a 2-way merge sorter when k is larger than 2. The k-way merge sorting is identical to 2-way merge sorting, except k sequences are merged in each step as shown in Figure 8 with a 4-way merge sort example. Order $n\log_k n$ memory cycles require order of $2n$ locations in memory to sort a sequence of length n. This is $\log_2 k$ times faster than the 2-way merge sort process. For example, when $k = 32$, the k-way merge sorter has five times greater sorter throughput than the 2-way merge sorter.

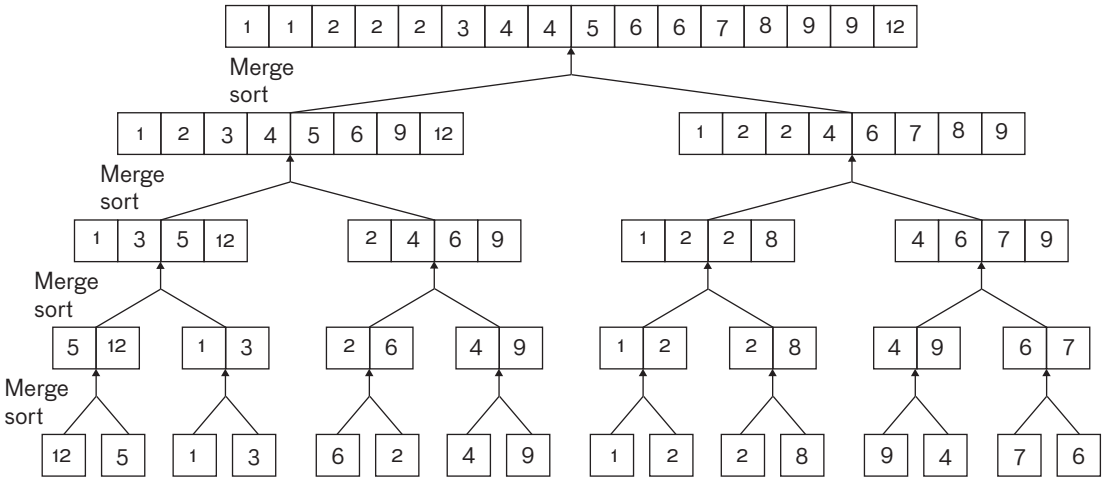


FIGURE 7. In this conventional 2-way merge sort, 16 items are sorted in four steps.

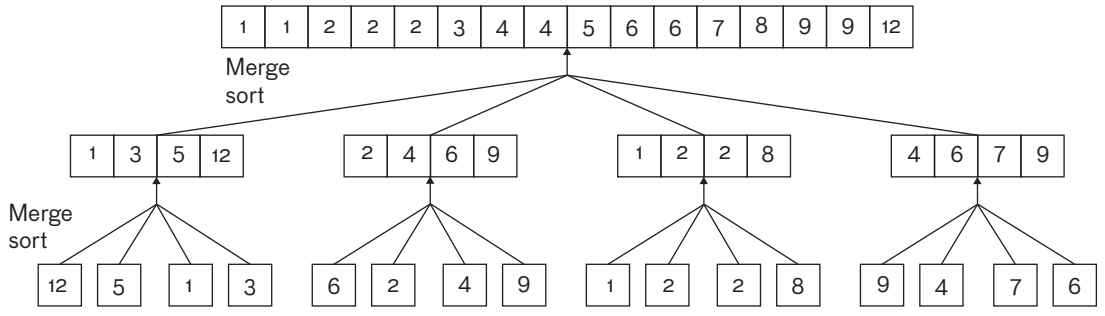


FIGURE 8. In the 4-way merge sort, the 16 items are sorted in half the steps of the sort in Figure 7.

The main difficulty with implementing a k-way merge sorter in a conventional processor is that it takes many clock cycles to figure out the smallest (or largest) value among k entries during each step of the merge sorting process. Ideally, the smallest value of k should be computed within one processor clock cycle for the maximum sorter throughput. The systolic merge sorter array shown in Figure 9 and Figure 10 can achieve such maximum sorter throughput during merge sorting of k presorted lists. Figure 9 shows an example of a 7-way merge sorter array. The array consists of three cells with the cell operations shown in Figure 10. Each cell has two registers. The top register R_S (for smaller) contains the smaller of the two values and the bottom register R_B (for bigger) contains the larger value. Each clock cycle, the cell passes the smaller value to the left if it is smaller than the larger value on the left. For example, the middle cell in Figure 9 at time 0 decides to pass the value 3 to the left because 3 is smaller than 7, which is the larger value on the left cell. As the result, the value 3 ends up in the left cell at time 1. Similarly, the larger value is passed to the right if it is bigger than the smaller value on the right adjacent cell. For example, the left cell in Figure 9 at time 0 decides to pass the value 7 to the right because 7 is larger than 3, which is the smaller value on the right adjacent cell. As the result, the value 7 ends up in the middle cell at time 1. In the continuous

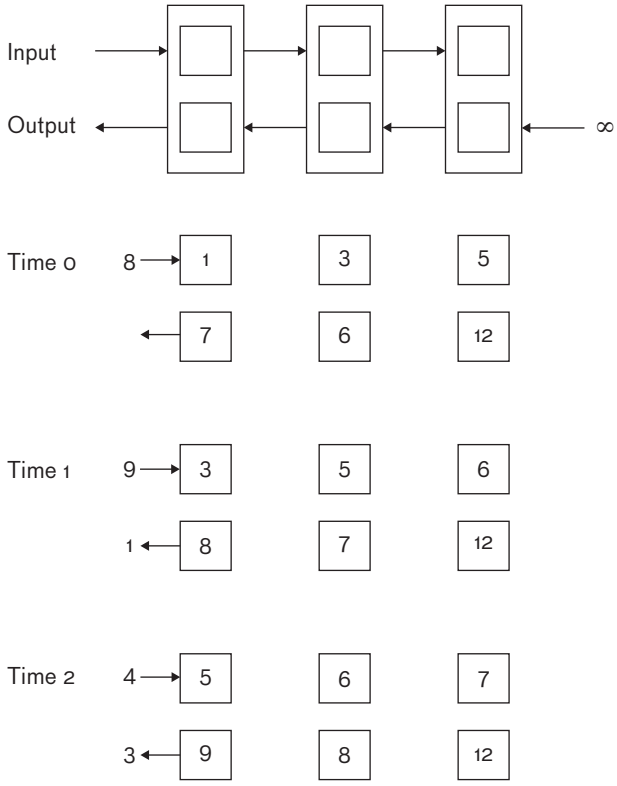
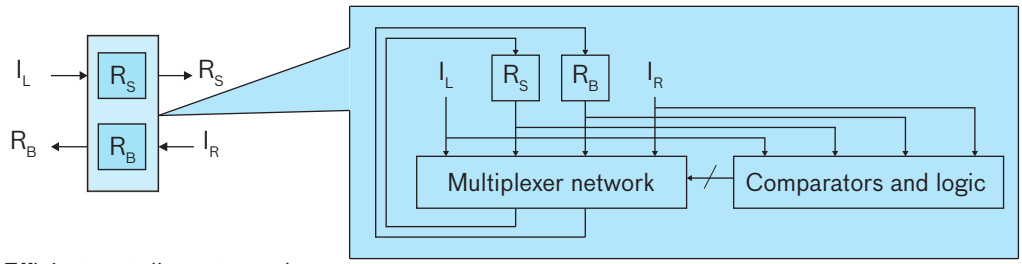


FIGURE 9. The illustration shows the systolic 7-way merge sorter array in operation.



Efficient systolic sorter node register update process

Condition	New R_S	New R_B
$I_L \le R_S \le R_B \le I_R$	R_S	R_B
$R_S < I_L \le R_B \le I_R$	I_L	R_B
$R_B < I_L$, $R_B \le I_R$	R_B	I_L
$I_L \le R_S$, $R_S \le I_R < R_B$	R_S	I_R
$I_L \le R_S$, $I_R < R_S$	I_R	R_S
$R_S < I_L \le I_R < R_B$	I_L	I_R
$R_S \le I_R < I_L$, $I_R < R_B$	I_R	I_L

R_S Smaller value register

R_B Bigger value register

Register values are updated depending on comparator results.

FIGURE 10. The systolic merge sorter array cell operation depends on the relationships between internal variables (R) and external inputs (I).

operation mode, the smallest value in the array is always at the register R_S of the left-most cell. As a new input is presented to the array, the smaller values between the input and the R_S of the left-most cell are outputted from the array in the very next clock cycle. At the same time, smaller values in the array march toward the left, and larger values march toward the right with each clock cycle. Because the array always outputs the smallest of the k values in the very next clock cycle, the k-way merge sort process can be carried out at the maximum possible throughput rate.

The systolic array works as follows. Before time 0, the array is preloaded with six values. Such loading can be achieved by preloading the registers with -∞ or a very small value. As six input values are loaded before time 0 from the left, the -∞'s are outputted to the left. At the same time, larger values march toward the right and smaller values march toward the left, guaranteeing that the smallest value in the array will always be at the R_S register of the left-most cell (correspondingly, the largest value in the array will be at the R_B register of the right-most cell). Toward the end of the k-way merge sort process when no more inputs are available, the systolic array can be emptied in sorted order by injecting ∞'s or very large values as inputs.

The systolic array implementation of the k-way merge sorter has numerous implementation advantages over conventional processor architectures. Because all the sorter cells are identical, developers can spend significant effort in optimizing the cell for high speed, low power consumption, and low circuit area, and can replicate the design to achieve very high performance with relatively little overall design effort. In addition, all the communication happens between neighboring cells, eliminating long communication paths, making high-speed, low-power communication between cells feasible. Because of these implementation advantages as well as inherently higher throughput of the k -way merge sort over a 2-way merge sort, the custom systolic sorter module can provide up to two orders of magnitude higher sorter throughput than prevailing microprocessor-based sorting.

3D Communication Network and Randomized Message Routing

The new graph processor architecture is a parallel processor interconnected in a 3D toroidal configuration using very high bandwidth links [5], as shown in Figure 11. The 3D toroid provides much higher communication performance than a two-dimensional (2D) toroid because of higher bisection bandwidth. Bisection bandwidth represents the worst-case communication bandwidth between two parts of a network partitioned to contain an equal number of nodes. Bisection bandwidth is often used to gauge the communication performance of the processor network for communication-intensive tasks.

In order to minimize communication link lengths, the 2D toroidal cluster is placed on a circuit board, and multiple circuit boards are stacked on top of each other to form the 3D toroid. The links between the circuit boards are enabled by an array of electromagnetic coupling connectors [6] that can communicate at high data rates without requiring physical conductor connections.

Custom-designed, high-speed input/output circuitries provide high-bit-rate, low-power communication for 2D links within the board and for 3D links between the boards. Multiple links share the delay lock loop (DLL) circuitry, as shown in Figure 12, because the DLL is the highest power consumption circuitry for the communica-

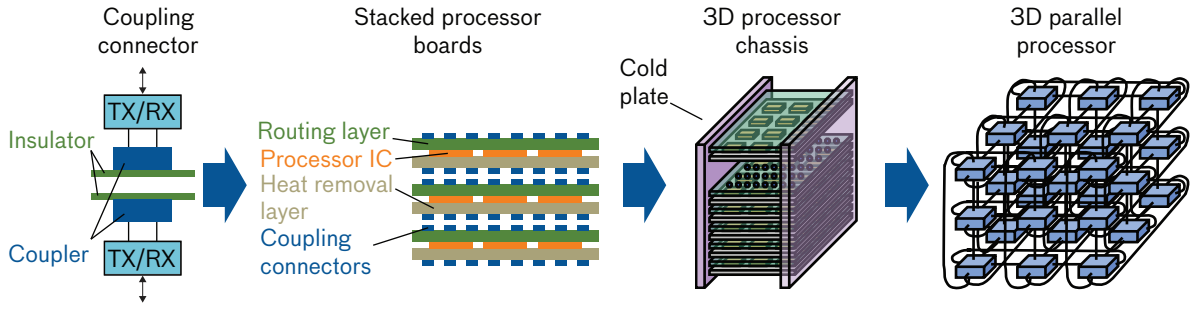


FIGURE 11. The illustration shows the structure of a 3D graph processor with electromagnetic coupling communications between processor boards. TX and RX are the transmitter and receiver.

tion links. Because all communication paths are relatively short with well-controlled lengths and impedances, such sharing is possible while maintaining the high bit rate. Each node processor is designed to be capable of a communication rate of more than one trillion bits per second to keep up with the communication demands of graph algorithms. The test chips have been designed to verify performance and power efficiency.

The 3D toroidal communication network is designed as a packet-routing network optimized to support small packet sizes that are as small as a single sparse matrix element. The network scheduling and protocol are designed so that successive communication packets from a node would have randomized destinations in order to minimize network congestion [6]. This design is a great contrast to typical conventional multiprocessor message-routing schemes that are based on the much larger message sizes and globally arbitrated routing that are used in order to minimize the message-routing overhead. However, large message-based communications are often difficult to route and can have a relatively high message contention rate caused by the long time periods during which the involved communication links are tied up. The small message sizes, along with randomized destination routing, minimize message contentions and improve the overall network communication throughput. Figure 13 shows the 512-node ($8 \times 8 \times 8$) 3D toroidal network simulation based on randomized destination communication versus unique destination communication. Even though both routing methods are based on small message sizes, the unique destination routing has a message contention rate that is closer to the contention rate of the conventional routing that is based on large message sizes. The randomized destination routing achieved approximately six times higher data rate and network utilization efficiency in the simulation using an identical network. The relative difference between the network utilization efficiencies is the important parameter because the absolute network utilization efficiency depends on exact communication links and routing algorithms.

Simulation and Performance Projection

A detailed simulation of the architecture was performed to verify the design and to estimate the performance. Bit-level accurate simulation models were used to simulate the entire 1024-node processor running the graph algorithm kernels. The performance projection was achieved

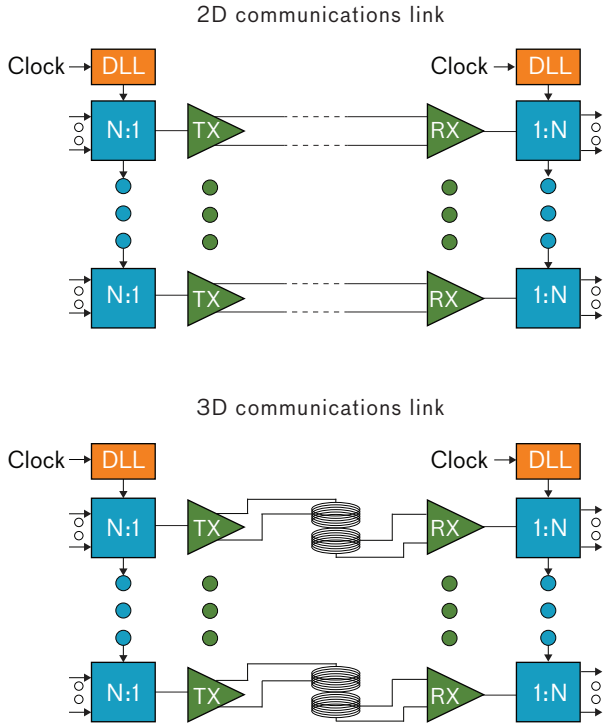


FIGURE 12. In the 2D and 3D parallel high-speed communication link circuitry, multiple links share the delay lock loop (DLL) circuitry.

by extrapolating the existing computation circuits to the target fabrication processes at 45 nm to 65 nm. The new custom communication circuitry was developed to provide 3D interconnection based on coupling connectors. Figure 14 shows the computational throughput projections versus number of processor nodes, assuming that the database size scales with the number of processors. We projected that the processor would provide several orders of magnitude higher graph computational throughput compared to the commercial alternatives. For the planned initial prototype with 1024 processor nodes, the projection for computation throughput is approximately three orders of magnitude higher than the best commercial alternatives.

The power-efficiency projection obtained with the same method is shown in Figure 15. The power efficiency was also projected to be several orders of magnitude higher. For the prototype with 1024 processor nodes, the projected power efficiency is up to four or five orders of magnitude higher than the best commercial alternatives. In many cases, the power efficiency is even more important than the computational throughput because

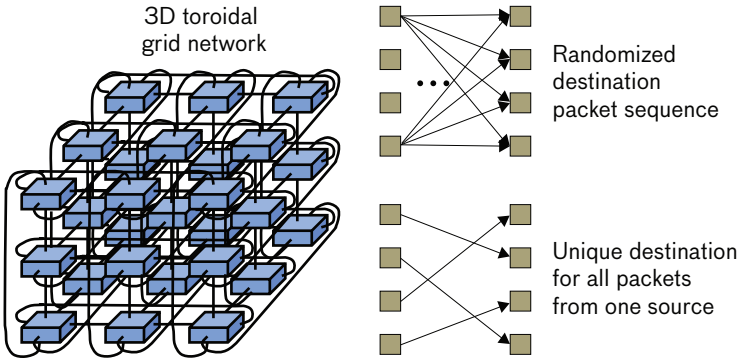


FIGURE 13. This illustration of the 3D graph processor with electromagnetic coupling communications between processor boards shows the contrast between a randomized destination and a unique destination in a simulation using a 512-node 3D toroidal network. In the randomized destination case, 87% full network efficiency is achieved. In the unique destination case, only 15% full network efficiency is achieved.

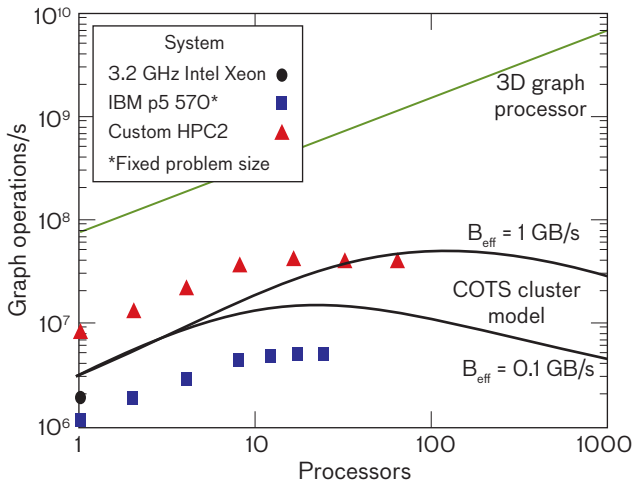


FIGURE 14. The computational throughput performance projections were made based on simulations. The 3D graph processor achieved orders of magnitude better computation performance than the commercial systems.

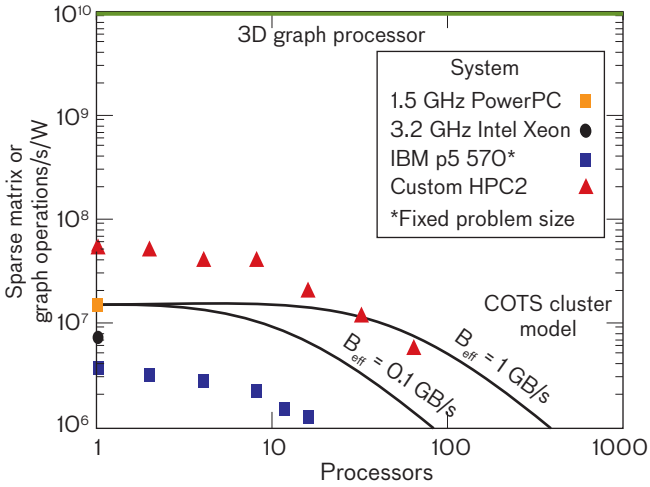


FIGURE 15. Power efficiency projections versus number of parallel processors were made based on simulations. The power-efficiency difference compared to commercial processors is expected to be even larger than the computational throughput difference as the number of processors grows because the computational throughput does not level off as in commercial processors.

the computational throughput of large database processing centers often tends to be limited by the availability of power and heat dissipation that can be provided.

The speedup of graph computation over multiple processors depends closely on how well the computational load is balanced between the processors. Advanced process mapping algorithms have been developed to optimize allocation of sparse matrix data and computations to achieve robust balancing of processing load and memory usage. A sparse matrix compiler could also be developed to enable a simplified user interface with MATLAB-like matrix high-level instructions.

Future Directions

Graph processing is of great interest to the cyber, Department of Defense (DoD), and intelligence communities. However, conventional computers are notoriously slow when running graph algorithms. This poor performance is mainly due to the inherent mismatches between the graph processing flows and conventional processor architectures. Graph algorithms can run faster in parallel processors, but performance gains quickly level off after a relatively small number of compute nodes because of the enormous interprocessor communication bandwidth requirements driven by the data flow patterns. Therefore, the sizes of

the problems to which graph algorithms could be applied have been severely limited, and many DoD and intelligence needs have gone unmet by conventional processors. To address these challenges, Lincoln Laboratory developed an entirely new 3D graph processor architecture—a significant departure from the variations of the von Neumann architecture that have dominated the computing world since the inception of the architecture.

Our detailed performance projections based on simulations point to orders of magnitude improvements in computational throughput and power efficiency over the best commercial alternatives. On the basis of these improvements, there will likely be numerous system insertion opportunities in the future for cyber and intelligence applications as well as for providing post-detection knowledge extraction and decision support processing in various ISR sensor platforms.

In addition, we are currently working to extend the instruction set to include instructions that are optimized for text-based data processing. Such enhancement is expected to significantly improve the analysis of "semantic" databases. We are also currently investigating the development and integration of a high-bit-rate, low-power optical communication network into the architecture. Such communication network technology would help the 3D graph processor architecture grow from thousands of nodes up to millions of nodes and beyond to handle very large databases in the future.

Acknowledgments

We would like to acknowledge the late Dennis Healy at the Defense Advanced Research Projects Agency Microsystems Technology Office (DARPA MTO), who made this work possible. We would also like to acknowledge the following research team members: Robert A. Bond, Nadya T. Bliss, Albert H. Horst, James R. Mann, Sanjeev Mohindra, Julie Mullen, Larry L. Retherford, and Eric I. Robinson. ■

References

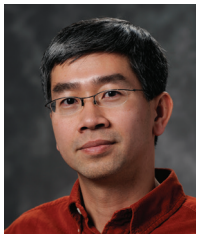
1. J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra.* Philadelphia: SIAM Press, 2011.
2. S.H. Roosta, *Parallel Processing and Parallel Algorithms, Theory and Computation.* New York: Springer-Verlag, 2000.
3. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms.* Cambridge, Mass.: The MIT Press, 2001.
4. W.S. Song, J. Kepner, H.T. Nguyen, J.I. Kramer, V. Gleyzer, J.R. Mann, A.H. Horst, L.L. Retherford, R.A. Bond, N.T. Bliss, E.I. Robinson, S. Mohindra, and J. Mullen, "3-D Graph Processor," Workshop on High Performance Embedded Computing, September 2010, available at http://www.ll.mit.edu/HPEC/agendas/proc10/agenda.html.
5. W.S. Song, "Processor for Large Graph Algorithm Computations and Matrix Operations," U.S. Patent pending, no. 13153490, June 6, 2011.
6. W. S. Song, "Systolic Merge Sorter," U.S. Patent no. 8,190,943, May 29, 2012.
7. W.S. Song, "Multiprocessor Communication Networks," U.S. Patent pending, no. 12703938, February 11, 2010.
8 W.S. Song, "Electromagnetic Coupling Connector for Three-Dimensional Electronic Circuits," U.S. Patent no. 6,891,447, May 10, 2005.

About the Authors

William S. Song is a senior staff member in the Embedded and Open Systems Group. Since his arrival at Lincoln Laboratory in 1990, he has been working on high-performance sensor and VLSI signal processor technologies for adaptive sensor array applications. He has developed numerous advanced signal processing algorithms, architectures, real-time embedded processors, and sensor array systems. He holds 11 U.S. patents and has 4 others pending, has submitted 23 invention disclosures, and has authored or co-authored 26 journal and conference publications. He received a 2005 MIT Lincoln Laboratory Technical Excellence Award and is an IEEE Senior Member. He received bachelor's, master's, and doctoral degrees from the Massachusetts Institute of Technology in 1982, 1984, and 1989, respectively.

Jeremy Kepner is a senior staff member in the Computing and Analytics Group. He earned a bachelor's degree with distinction in astrophysics from Pomona College. After receiving a Department of Energy Computational Science Graduate Fellowship in 1994, he obtained his doctoral degree from the Department of Astrophysics at Princeton University in 1998 and then joined MIT. His research is focused on the development of advanced libraries for the application of massively parallel computing to a variety of data-intensive signal processing problems. He has published two books and numerous articles on this research. He is also the co-inventor of parallel Matlab, Parallel Vector Tile Optimizing Library (PVTOL), Dynamic Distributed Dimensional Data Model (D4M), and the Massachusetts Green High Performance Computing Center (MGHPCC).

Vitaliy Gleyzer has been a staff member in the Embedded and Open Systems Group at Lincoln Laboratory for four years. Prior to joining the Laboratory, he received his master's degree in electrical and computer engineering from Carnegie Mellon University, with a research concentration on network architecture and network modeling. His current work and research interests are primarily focused on high-performance computing systems and embedded systems engineering.



Huy T. Nguyen is a staff member in the Embedded and Open Systems Group. He has been involved with architecting and leading the development of several high-performance power-efficient signal processor systems, specialized accelerators for nontraditional signal processing. He has published 20 papers, co-authored two book chapters, and consulted on high-performance computing technologies. He received his bachelor's degree at the University of Delaware in 1989 and joined Lincoln Laboratory in 1998 after earning his doctoral degree from the Georgia Institute of Technology in the area of low-power, very-large-scale integration for digital signal processing applications. Prior to pursuing his doctorate, he worked on real-time radar software at the Georgia Tech Research Institute.

Joshua I. Kramer is a technical staff member in the Cyber Systems and Technology Group. He has worked on low-power, high-performance circuit design and embedded computing architecture since joining Lincoln Laboratory in 2007. His current research interests are in anti-tamper, secure, and trusted computing architectures, and hardware accelerated cryptography. He holds a doctorate in electrical engineering and a bachelor's degree in computer engineering from the University of Delaware.