# Analysis and Mapping of Sparse Matrix Computations

Nadya Travinin Bliss, Sanjeev Mohindra
Varun Aggarwal, Una-May O'Reilly
{nt, smohindra}@ll.mit.edu
MIT Lincoln Laboratory, Lexington, MA 02420
varun_ag@mit.edu, unamay@csail.mit.edu
MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA 02139

## Abstract

In the past, knowledge processing (anomaly detection, target identification, social network analysis) of sensor data did not require real-time processing speeds. However, the rapid growth in the size of the data and the complexity of data analysis are driving the need for applications that provide real-time signal and knowledge processing at the front end. Many knowledge processing algorithms, such as Bayesian networks, social networks, and neural networks, are based on graph algorithms. Graph algorithms are difficult to parallelize and thus cannot take advantage of multi-core architectures. Many graph operations can be cast as sparse linear algebra operations. While this increases the ease of programming, parallel sparse algorithms are still inefficient. This talk presents an automatic mapping and routing framework for sparse operations. This work extends the pMapper automatic mapping framework to handle the finer granularity of the sparse matrix problem. New machine modeling and program analysis techniques are discussed, followed by details of the mapping algorithm and preliminary results.

## Introduction and Motivation

MIT Lincoln Laboratory has been developing techniques for automatic mapping of signal processing applications. The pMapper [1, 2] automatic mapping approach has been demonstrated to be both feasible and effective. However, modeling, program analysis, and mapping techniques have been limited to dense matrix computations. As more and more post-processing algorithms move to the front end, efficient parallel implementations of those algorithms are becoming necessary.

Post-processing algorithms are typically represented in graphical form, but can also be formulated using sparse matrix notation. The efficiency of the graph algorithms is often only a small fraction (< 0.01) of the peak throughput of a conventional architecture, and the efficiency deteriorates as the problem size increases. Since graph algorithms are not well suited for parallelization, the implementation efficiency is likely to reduce even further as multi-core processors become prevalent. Sparse matrix formulations expose intrinsic algorithm parallelism; however, efficient mappings are still needed to exploit the parallelism and deliver respectable efficiency, motivating the research reported here.

Let us consider an example where we need to perform anomaly detection on a network of about 10,000 entities. Given a disturbance in the environment, we need to make decisions in a second or less. The data can be represented by a 10K × 10K sparse matrix. A typical processing step could involve performing 20K operations per entity, requiring $2\times10^8$ operations per second. These operations have irregular data access patters, distinctly different from standard signal processing computations, making the mapping problem increasingly challenging. The low efficiency of these operations presents a computation challenge in front-end sensor form-factors. Additionally, the problem size is expected to grow significantly in the near future to allow for persistent surveillance.

All of these factors motivate the need for increasing parallel efficiency of sparse matrix computations. This efficiency is tightly coupled to the underlying network architecture requiring fine-grained analysis of communication patterns.

## Machine Model

To provide detailed analysis of the communication operations, a detailed model of the underlying architecture is necessary. This model has to provide accurate representation of machine topology and allow for heterogeneous networks. To address this need, we have chosen to represent the machine model as a graph structure, with adjacency matrices representing latency and bandwidth between any two nodes. Specifically, $L$ and $B$ are matrices for latency and bandwidth respectively, where $L_{ij}$ represents latency between nodes $i$ and $j$; and $B_{ij}$ represents bandwidth between nodes $i$ and $j$. Additional processor information is stored in a linear array with each entry corresponding to each processing element.

## Program Analysis

In order to efficiently map sparse arrays, access to individual communication operations is desirable. Figure 1 illustrates an example where the computation is addition of two matrices, followed by a redistribution operation. 1a is the coarse-grained signal flow graph, or parse tree, of the operation. 1b is a sample mapping. Here, different colors indicate different processors. Specifically, B and C are mapped onto processors 2 (green) and 3 (light green) and A is mapped onto processors 0 (blue) and 1 (light blue). 1c is the list of operations that need to occur. For example, consider operation 11 which is a communication operation between processor 2 and 0. The time for this operation depends on the underlying network topology and the route chosen. Finally, 1d is the fine grained signal flow graph of the addition, with each node corresponding to an operation in 1c.

*This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

All nodes in the fine-grained signal flow graph can be classified as memory operations (memop), computation operations (cpuop), or communication operations (netop). For each operation, the program analysis framework provides information regarding what processor(s) is (are) involved and the amount of data that is being operated on. Sparse support has been implemented by storing both index ranges and number of non-zeros in each node.

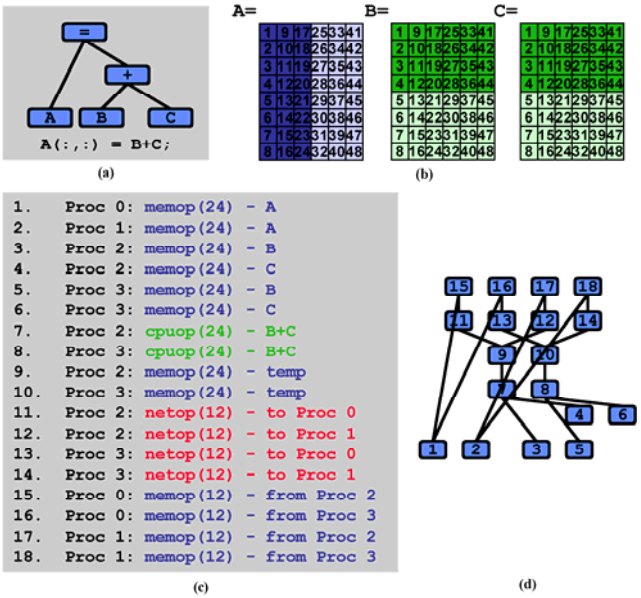


**Figure 1: Extracting individual operations (c) and fine-grained signal flow graph (d) from coarse-grained signal flow graph (a) and maps (b).**

## Mapping and Results

A nested genetic algorithm (GA) has been implemented to handle mapping of sparse arrays onto detailed machine model specifications. Two nested GAs (Figure 2) complementarily search for a map (outer GA) and then search for the best route among the routing options for the map (inner GA). The nesting of the two algorithms yields a <map, route> candidate solution. The performance of this candidate can be estimated on the fine-grained topologically defined machine model that has routing dependencies. For example, communication operations, such as operation 11 in Figure 1c cannot be determined without first determining the maps for the arrays. The combinatorial nature of both the mapping and routing problems makes the approach well addressed by the GAs.

Due to the nature of the sparse matrix computation, the mapper is allowed to consider irregular mappings, or mappings that are not purely block-cyclic in nature. To evaluate the quality of our solutions, the results are compared with the standard mapping used for sparse computations, which is 2D cyclic. The results in Table 1 are for a matrix multiply operation [3]. The computation was mapped onto an 8-processor ring and an 8-processor cube. A matrix multiply operation was chosen because it is a key kernel in post-processing algorithms. In the talk, results will be presented for larger arrays, however the preliminary results already indicate that access to sparsity information and support for irregular mappings provide an order of magnitude performance improvement over 2D cyclic distribution. We expect results to further improve as we fine tune the GA. Figure 3 illustrates a sample mapping produced by the mapping framework. Different colors indicate different processors used.

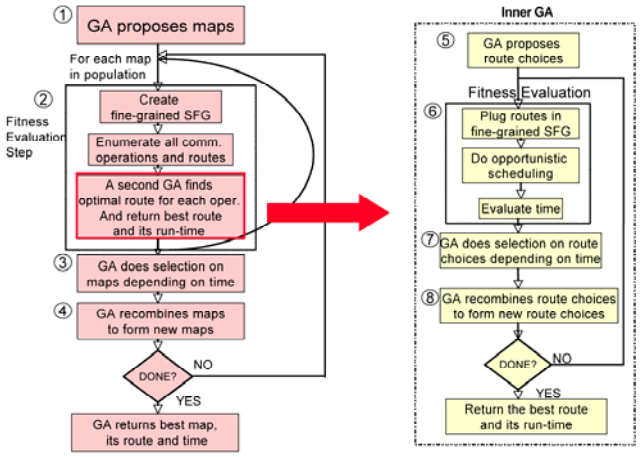


**Figure 2: Genetic Algorithm Mapper.**

**Table 1: Operations per second using different mappings.**

| | 2D Cyclic Map | GA Mapper |
|---|---|---|
| **8 proc ring** | $5.65\times 10^{6}$ | $5.65\times 10^{7}$ |
| **8 proc cube** | $7.80\times 10^{6}$ | $1.41\times 10^{8}$ |

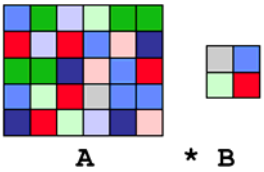


**Figure 3: Sample mappings produced by the mapper.**

## Summary

For this talk, pMapper automatic mapping framework is extended to provide support for mapping sparse arrays and computations on those arrays. Automatic mapping of sparse arrays is a significantly different problem from mapping of dense arrays. Finer grained analysis is necessary and access to the sparsity pattern is valuable. The talk will present results for larger problems. Additionally, many sparse computations have similar patterns. Once a solution is evolved, it is possible to keep re-using it within the same family of matrices. The results for various sparse matrix families will also be presented.

## References

[1] N. Travinin, H. Hoffmann, R. Bond, H. Chan, J. Kepner, E. Wong, "pMapper: Automatic Mapping of Parallel Matlab Programs," *HPEC 2005 Workshop*, Lexington, MA, September 2005.

[2] N. T. Bliss, J. Dahlstrom, D. Jennings, S. Mohindra, "Automatic Mapping of the HPEC Challenge Benchmarks," *HPEC Workshop 200*6, Lexington, MA, September 2006.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations, 3rd Edition*. John Hopkins University Press, Baltimore, Maryland, 1996.