

Lecture 15: March 17

*Lecturer: Dr. Horst Simon**Scribes: Hong Kim*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

15.1 Introduction to SpMV (Sparse Matrix-Vector Multiply)

Sparse Linear Algebra is one of the Seven Dwarves or Motifs covered in the beginning of the course. Sparse Matrix-Vector Multiply is a subset of Sparse Linear Algebra. SpMV is key to solving many ODE problems when using explicit or implicit numerical algorithms. While there are two types of implicit solvers, direct and iterative, this lecture considers iterative solvers only. Lecture 16 covers direct solvers, i.e. Gaussian elimination. Two crucial issues for parallel SpMV are **locality** and **load balance**.

While there are many ways to store the non-zero terms of a sparse matrix, the most common way is the Compressed Sparse Row (CSR) format. CSR uses three arrays, one for the row indices, another for the column indices pointing to the non-zero entries, and one for the non-zero values.

15.2 Tuning of SpMV for Single Processor

Historically, SpMV performance has been around 10% or less of peak performance as shown on slide 8 of the lecture notes. The performance depends on the machine, the kernel, and the matrix. Since the matrix is not known until run-time, tuning SpMV is difficult.

15.2.1 Sparse Matrix Blocking

One way to better performance is by taking advantage of the inherent block structure of a sparse matrix. Most of the performance bottleneck occurs when acquiring the matrix from memory since there are only two flops per every non-zero element in a sparse matrix for SpMV. The time to get the matrix from memory can be decreased by not storing each individual non-zero element with an index but by storing each non-zero (row X column) block of with an index. In addition, optimizing the block size speeds up the computation. The optimal block size depends heavily on the architecture as seen in Table 15.1.

Table 15.1: Optimal Block Size Search in a 12 x 12 for Various Architectures

Architecture	Row Size	Column Size
Itanium 1	4	1
Itanium 2	4	2
Ultra 2i	6	8
Ultra 3	12	12
Pentium III	2	10
Pentium III-M	10	12
Power3	4	4
Power4	8	8

15.2.2 Complicated Non-Zero Structure

Blocking the matrix become challenging when the non-zero structure becomes complicated. Refer to slide 18 for an example. Selecting a certain block size requires many explicit zero fill-ins. Therefore selecting the correct block size is crucial. Automatic register blocking uses:

- Offline Benchmark - pre-compute **flops(row,col)** using a dense matrix for each row and col. Only once per machine.
- Run-Time Search - Sample sparse matrix to find **fill(row,col)** for each row and col.
- Run-Time Heuristic Model - choose row and col to minimize **fill(r,c)/flops(r,c)**

15.3 Distributed Memory SpMV

The main issue that arises when running SpMV in parallel on a distributed machines is partitioning. Which processor stores what part of the vector or matrix? Which processors are responsible for the computation?

A sparse matrix is partitioned by non-zero entry counts and not by rows and columns. There are two types of partitions that are generally used, 1D and 2D. 1D partition is the most popular, but these scale with $\log(\#ofprocessors)$ when doing a reduction on y . 2D partition scales with $\log(\sqrt{\#ofprocessors})$ on reduction but loading balancing becomes an issue.

15.3.1 Load Balance Partitioning

The simplest approach is to 2D partition a matrix using equal number of rows and columns for each processor. As seen on slide 30 of this lecture, this may lead to unbalanced loads. To create a better load-balanced partition:

1. Block into Rows - keeping the same number of non-zero entries fro each blocked row.
2. Block within Each Blocked Row - block each row using an even distribution of non-zeros. All blocks with the entire matrix should have equal number of non-zero entries.
3. Prune unneeded rows and columns - those that contain all zero entries.
4. Re-encode Column Indices to be Relative to each Thread Block.

15.3.2 Reordering and Partitioning

An ideal matrix structure for matrix-vector multiply is a blocked-diagonal matrix where the number of blocks equal the number of processors. If there are no non-zero entries outside the blocks, no communication is required between the processors. It may be possible to reorder a matrix into a blocked-diagonal structure using graphs. The reordering should balance load and storage, minimize inter-processor communication, and improve register and cache re-use.

There exists a relationship between a matrix and a graph. The edges in a graph represent the non-zero entries of a sparse matrix, and the nodes are the row or column indices. A graph should be partitioned so that there are equal number of nodes for each partition to balance storage and load. The number of edge-crossings between partitions should also be minimized to limit communication. The rows and columns should be reordered by grouping all the nodes in one partition next to each other.

15.4 Applications

This application example is from the SLAC accelerator cavity design which uses a sparse matrix [Ko]. Ko is a 50000x50000 sparse matrix with 2287944 non-zero entries.

15.4.1 Optimization on Itanium 2 - T3P

80% of the time was spent in SpMV. Symmetric storage and register blocking was used for optimization. There was an 1.68 times speed up resulting in 532 Mflops and an 4.4 times speedup with multiple 8 vectors resulting (1380 Mflops) for a single processor.

15.4.2 Optimization on Power 4 - Omega3P

In this application, SpMV was not dominant. Optimizations implemented were symmetric storage, register blocking, and reordering. Reverse Cuthill-McKee ordering was used to reduce the bandwidth. Traveling Salesman Problem-based ordering was used to create blocks. The optimization resulted in a 2.1 times speedup.