Structure-Aware Parallel Algorithm for Solution of Sparse Triangular Linear Systems

Ehsan Totoni, Michael T. Heath and Laxmikant V. Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
{totoni2, heath, kale}@illinois.edu

ABSTRACT

Solution of sparse triangular systems of linear equations is a performance bottleneck in many methods for solving more general sparse systems. In both direct methods and iterative preconditioners, it is used to solve the system or refine the solution, often across many iterations. Triangular solution is notoriously resistant to parallelism, however, and existing parallel linear algebra packages appear to be ineffective in exploiting much parallelism for this problem. We develop a novel parallel algorithm based on various heuristics that adapts to the structure of the matrix and extracts parallelism that is unexploited by conventional methods. By analysis and reordering operations, our algorithm can extract parallelism of many different sparse matrix structures.

Keywords

Triangular solver, Parallel algorithms, Sparse linear systems, Distributed memory machines

1. INTRODUCTION

Solution of sparse triangular linear systems is an important kernel for many numerical linear algebra problems, such as linear systems and least squares problems, that arise in many science and engineering simulations. It is used extensively in direct methods, as well as many iterative methods (such as Gauss-Seidel method) or in many preconditioners for other iterative methods. Unfortunately, the parallel performance of triangular solution is notoriously poor, so it is a performance bottleneck for many of these methods.

For example, a Preconditioned Conjugate Gradient (PCG) method with Incomplete-Cholesky as the preconditioner will have a triangular solve with the preconditioner matrix in every step. This operation costs in the order of the number of non-zeros of the preconditioner matrix, which has the same number of non-zeros as the coefficient matrix or more. Thus, it takes about the same time as the Sparse Matrix Vector product (SpMV) or more, accounting for 50% or more of the floating point operations (assuming enough non-zeros so that the vector operations are negligible). Thus, if the triangular solve does not scale (which is the case in most standard packages), the parallel speed up of the PCG method is at most 2 according to Amdahl's Law, no matter how many processors are used in a parallel machine. Therefore, scalability of triangular solve is crucial.

Here, we devise an algorithm that uses various heuristics to adapt to the structure of the sparse matrix, with the goal of exploiting as much parallelism as possible. Our data distribution is in blocks of columns, which is natural for distributed-memory computers. Our analysis phase is essentially a simple local scan of the rows and nonzeros and is done fully in parallel, with limited information from other blocks. The algorithm reorders the rows so that independent rows are extracted for better concurrency. It also tries to compute the rows that are needed for other blocks (probably on the critical path) sooner and send the required data.

We implement our algorithm in Charm++, since many features of Charm++, such as virtualization, make the implementation easier and enhance performance [1].

SC '13 November 17-21, 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2503210.2503228

2. NEW TRIANGULAR PARALLELISM APPROACH

In this section, we use examples to illustrate various opportunities for parallelism that we exploit in our algorithm for computation of the solution vector x to an $n \times n$ lower triangular system $Lx = b$ using forward substitution. Figure 1 shows a sparse lower triangular matrix (of a linear system), for which the computation of x_8 (the left-hand side variable corresponding to the 8th row) depends only on x_1, so x_8 can be computed as soon as x_1 has been computed, without having to await the availability of $x_2, \ldots, x_7$. Similarly, computation of x_3, x_6, and x_9 can be done immediately and concurrently, as they depend on no previous components.

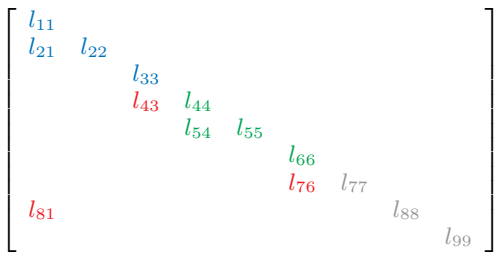

Figure 1: Sparse matrix example 1.

Assume that the columns of L are divided among three processors (P1, P2, P3) in blocks, as shown by the color coded diagonal blocks (blue, green, gray) in Figure 1. Nonzeros below the diagonal blocks are colored red. If each processor waits for all the required data, all work is done sequentially among processors and there is no overlap.

However, there are some sources of parallelism in this example. Row 3 is independent, since it has no nonzeros in the first two columns. Thus, x_3 can be computed immediately by P1 and sent to P2 earlier than x_2. P1 can then process l_{43} and send the result to P2. In this way, P1 and P2 can do most of their computations in parallel. The same idea can be applied to processing of l_{76} and l_{81}, and more concurrency is created. To exploit independent rows, they could be permuted to the top within their block, and then all rows are processed in order. Thus, in our example rows 3, 6, and 9 can be completed concurrently. P1 then processes l_{43}, sends the result to P2, processes row 1 (in the original row order), sends the result from l_{81} to P3, and finally completes row 2. Similarly, P2 first processes row 6, sends the result from l_{76} to P3, receives necessary data from P1, and then processes its remaining rows. P3 can process row 9 immediately, but must await data from P1 and P2 before processing its other rows.

Another common case that may provide opportunities for parallelism is having some denser regions below the diagonal block (e.g. Figure 2). If we divide that region among two additional processors (P4 and P5), they can work on their data as soon as they receive the required solution components.

$$\begin{bmatrix} l_{11} & & & & & & & & & \\ l_{21} & l_{22} & & & & & & & & \\ & & l_{33} & & & & & & & \\ & & l_{43} & l_{44} & & & & & & \\ & & & l_{54} & l_{55} & & & & & \\ & & & & & l_{66} & & & & \\ & & & & & l_{76} & l_{77} & & & \\ l_{81} & l_{82} & l_{83} & & & & & l_{88} & & \\ l_{91} & l_{92} & l_{93} & & & & & & l_{99} & \end{bmatrix}$$

Figure 2: Sparse matrix example 3.

These three strategies — sending data earlier to achieve greater overlap, identifying independent rows, and parallel processing of dense offdiagonal regions — are the bases for our algorithm.

3. TEST RESULTS

Figure 3 shows the scaling of our implementation for up to 512 cores of BlueGene/P using triangular matrices from incomplete LU factorization with no fill (compared to the best sequential algorithm). Since the matrices are small relative to the number of cores used, the results represent strong scaling of this approach. Furthermore, in our experiments, the analysis time was comparable to the time of one solve iteration, which is negligible.

Figure 4 compares the performance of our method with that of HYPRE, which is a commonly used linear algebra package. As shown, our method can exploit parallelism on many matrices, whereas HYPRE's scaling is poor in all cases.

Note that some other algorithms, e.g. the DAG approaches, have limited concurrency and thousands of barriers (even

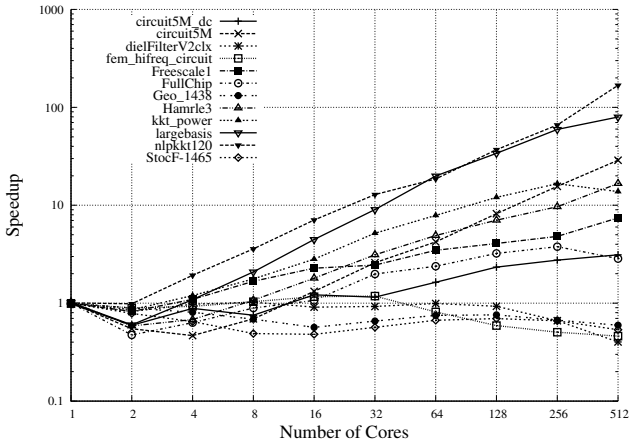

Figure 3: Scaling for no-fill incomplete-LU matrices.

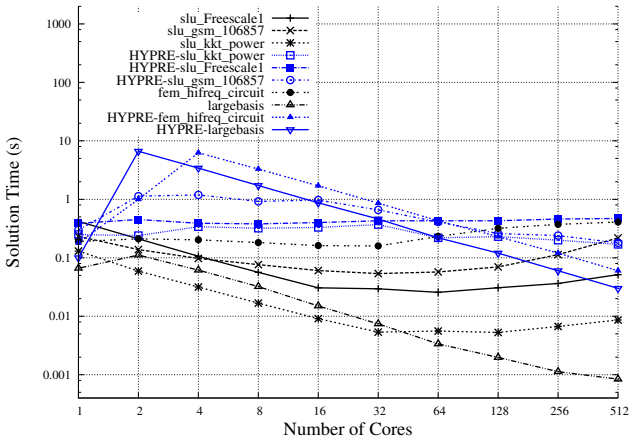

Figure 4: Comparison with triangular solver from HYPRE.

for small matrices sometimes), so they cannot scale to many cores of a distributed-memory machine. To understand why our algorithm is much more scalable than DAG-based ones (e.g. Level-set algorithm), we analyze the critical path (longest communication chain) of the two approaches for a sample of matrices (Table 1).

Matrix	Our algorithm	level-set algorithm
circuit5M	2	18
kkt_power	3	17
Freescale1	18	216
Hamrle3	25	31083
Geo_1438	87	5823

Table 1: Critical path length comparison

4. REFERENCES

[1] E. Totoni, M. T. Heath, and L. V. Kale, "Structure-adaptive parallel solution of sparse triangular linear systems," Parallel Programming Laboratory, Tech. Rep. 12-42, 2012.