Structure-Adaptive Parallel 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
E-mail: {totoni2, heath, kale}@illinois.edu

***Abstract*—**
Solving sparse triangular linear systems is crucial for many numerical methods used in applications. It is used in both direct and iterative methods (as well as their preconditioners) to solve the system or improve an approximate solution (often across many iterations). However, solving triangular systems is notoriously resistant to parallelism. Thus, it often acts as a performance bottleneck for solving general sparse systems. Furthermore, existing parallel linear algebra packages appear to be ineffective in exploiting significant parallelism for this problem.
We present a novel parallel algorithm based on various heuristics that adapt to the structure of the matrix and extract parallelism that is unexploited by conventional methods. By analyzing and reordering operations, our algorithm can often extract parallelism even for cases where most of the nonzero matrix entries are near the diagonal. Our main parallelism strategies are: (1) identify independent rows, (2) send data earlier to achieve greater overlap, and (3) process dense off-diagonal regions in parallel. We describe the implementation of our algorithm in Charm++ and MPI and present promising experimental results on up to 512 cores of BlueGene/P, using numerous sparse matrices from real applications[1].

I. Introduction

Solving sparse triangular linear systems is an important kernel for many numerical linear algebra problems in science and engineering simulations, i.e. linear systems and least square problems [1], [2], [3]. It is used extensively in direct methods following a triangular factorization to solve the system (possibly with many right-hand sides), or to improve an approximate solution iteratively [4]. It is also a fundamental kernel in many iterative methods (such as Gauss-Seidel method) and in many preconditioners for other iterative methods (such as Incomplete-Cholesky preconditioner for Conjugate Gradient) [5]. Unfortunately, performance of parallel algorithms for triangular solution is notoriously poor, resulting in performance bottlenecks for many of these methods.

For example, a Preconditioned Conjugate Gradient (PCG) method with Incomplete-Cholesky as the preconditioner has two triangular solves with the preconditioner matrix at every step. The preconditioner matrix has at least as many nonzeros as the coefficient matrix. Therefore, the number of the floating point operations for the triangular solves is proportional to the number of nonzeros of the coefficient matrix. As a result, the triangular solves take about the same time as the Sparse Matrix Vector product (SpMV). This accounts for for 50% of the floating point operations (assuming sufficiently many nonzeros that the vector operations are negligible). Thus, if the triangular solves do not scale (which is the case in most standard packages [6]), then according to Amdahl's Law, the parallel speedup of the PCG method cannot exceed two, no matter how many processors are used. Therefore, improving the scalability of solving triangular systems is crucial.

Solving sparse triangular systems is particularly resistant to efficient use of parallel machines because there is little concurrency in the nature of the computation and the work per data entry is small. The lack of concurrency is due to structural dependencies that must be satisfied for the computation of each solution entry. By the nature of the successive substitution algorithm, computation of each solution component potentially must await the computation of all previous entries. Once these dependencies have been satisfied, computation of the next solution component requires only one multiply-add and one division. Consequently, the communication cost is high compared with the computation. This is especially true on distributed-memory parallel computers.

Despite the apparent lack of parallelism and relatively high communication overhead, sparse triangular systems are usually solved in parallel. The reasons for this are memory scalability and because the matrix is typically already distributed across processors from a previous computation (e.g., factorization). This may explain why some standard packages implement triangular solves in parallel even though parallel performance may be much slower than sequential computation.

It is therefore essential to achieve as much efficiency as possible in parallel triangular solution, especially considering the many (often required) iterations that can dominate the overall solution time. What is needed is a new parallel algorithm for solving triangular systems that can provide high performance for a range of practical matrices.

Previous work on this problem focused on two main directions. First, various techniques, such as dependency analysis and partitioning, have been employed to exploit sparsity and identify parallelism [7], [8], [9]. For example, a level-set triangular solver constructs a directed acyclic graph (DAG) capturing the dependencies among rows of the matrix. Then it processes each level of the DAG in parallel and synchronizes before moving on to the next. Since data redistribution and many global synchronizations are usually required, these methods are most suitable for shared memory machines, and most recent studies have considered only shared memory architectures [8], [9]. The second class of methods is based on partitioning the matrix into sparse factors and inversion [10], [11]. However, the cost of preprocessing and data redistribution may be high, and the benefits seem to be limited. In addition, numerical stability may be questionable for these nonsubstitution methods. Nevertheless, after years of development, these methods have not found their way into standard linear algebra packages, such as HYPRE [12], because of their

[1]The full version of this paper can be found following this link: http://charm.cs.uiuc.edu/~ehsan/stsolver_totoni.pdf

limited performance.

Because of this lack of scalable parallel algorithms for solving triangular systems, many packages such as HYPRE avoid algorithms that require it, often thereby incurring numerical issues [6]. An example is hybrid smoothers that use Gauss-Seidel within each processor but use Jacobi method between processors. In addition, incomplete factorization schemes often drop interprocessor connections to be able to utilize faster algorithms. However, these methods introduce new numerical issues, which our algorithm is likely to mitigate.

In this study, we present 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 a simple local scan of the rows and nonzeros, is done fully in parallel, and with limited information from other blocks. The algorithm reorders the rows so that independent rows are extracted for better concurrency. It also tries to process the rows that are needed for other blocks (probably on the critical path) sooner and send the required data. Another positive property of the algorithm is that it allows various efficient node-level sequential kernels to be used (although not evaluated here).

We describe our implementation in CHARM++[13] and discuss the possible implementation in MPI. We believe that many features of CHARM++, such as virtualization, make the implementation easier and enhance performance. We use several matrices from real applications (University of Florida Sparse Matrix Collection [14]) to evaluate our implementation on up to 512 cores of BlueGene/P. The matrices are fairly small relative to the number of processors used, so they illustrate the strong scaling of our algorithm. We compare our results with triangular solvers in the HYPRE [12] and SuperLU_DIST [4] packages to demonstrate the superiority of our algorithm relative to current standards.

II. Parallelism in Solving Sparse Triangular Systems

In this section we use examples to illustrate various opportunities for parallelism that we exploit in our algorithm. Computation of the solution vector x for an $n \times n$ lower triangular system $Lx = b$ using forward substitution can be expressed by the recurrence

$$x_i = (b_i - \sum_{j=1}^{i-1} l_{ij}\, x_j)/l_{ii}, \quad i = 1, \ldots, n.$$

For a dense matrix, computation of each solution component x_i depends on all previous components x_j, $j < i$. For a sparse matrix, however, most of the matrix entries are zero, so that computation of x_i may depend on only a few previous components, and it may not be necessary to compute the solution components in strict sequential order. For example, Figure 1 illustrates a sparse lower triangular system for which the computation of x_8 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 $x_2, \ldots, x_7$. In addition, computation of x_3, x_6, and x_9 can be done immediately and concurrently, as they depend on no previous components. These dependencies are conveniently described in terms of matrix rows: we say that row i depends on row j for $j < i$ if $l_{ij} \neq 0$. Similarly, we say that row i is independent if $l_{ij} = 0$ for all $j < i$. We can also conveniently describe the progress of the algorithm in terms of operations involving the nonzero entries of L, since each is touched exactly once.

$$\left[\begin{array}{ccc:ccc:ccc}
l_{11} & & & & & & & & \\
l_{21} & l_{22} & & & & & & & \\
 & & l_{33} & & & & & & \\
 & & l_{43} & l_{44} & & & & & \\
 & & & l_{54} & l_{55} & & & & \\
 & & & & & l_{66} & & & \\
 & & & & & l_{76} & l_{77} & & \\
l_{81} & & & & & & & l_{88} & \\
 & & & & & & & & l_{99}
\end{array}\right]
\left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ \hdashline x_4 \\ x_5 \\ x_6 \\ \hdashline x_7 \\ x_8 \\ x_9 \end{array}\right]
=
\left[\begin{array}{c} b_1 \\ b_2 \\ b_3 \\ \hdashline b_4 \\ b_5 \\ b_6 \\ \hdashline b_7 \\ b_8 \\ b_9 \end{array}\right]$$

Fig. 1. Sparse matrix example 1.

Continuing with our example, assume that the columns of L are divided among three processors (P1, P2, P3) in blocks, as shown by the dashed lines and 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, processes its rows in increasing order and sends the resulting data afterwards, then we have the following scenario. P2 and P3 wait while P1 processes all its rows in order, then sends the result from l_{43} to P2 and the result from l_{81} to P3. P2 can now process its rows while P3 still waits. After P2 finishes, P3 now has all the required data and performs its computation. Thus, all work is done sequentially among processors and there is no overlap.

However, there is another source 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, before computing x_1 and x_2. P1 can then process l_{43} and send the resulting partial sum 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, as illustrated in Figure 2, and then all rows are processed in order, or the row processing could be reordered without explicit permutation of the matrix. In either case, rows 3, 6, and 9 of our example can be processed 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.

$$\left[\begin{array}{ccc:ccc:ccc}
 & & l_{33} & & & & & & \\
l_{11} & & & & & & & & \\
l_{21} & l_{22} & & & & & & & \\
 & & & & & l_{66} & & & \\
 & & l_{43} & l_{44} & & & & & \\
 & & & l_{54} & l_{55} & & & & \\
 & & & & & & & & l_{99} \\
 & & & & & l_{76} & l_{77} & & \\
l_{81} & & & & & & & l_{88} &
\end{array}\right]$$

Fig. 2. Reordering rows of sparse matrix example 1.

This idea applies to many practical cases, but may not provide any benefit for others. For example, a triangular matrix with its diagonal and subdiagonal full of nonzeros has a chain of dependencies between rows, and the computation is essentially sequential. Matrices from various applications have a wider variety of nonzero structures and properties.

Another common case that may provide opportunities for parallelism is having some denser regions below the diagonal block. Figure 3 presents an example with a dense region in the lower left corner. 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. In this approach, P1 broadcasts the vector $x_{(1..3)}$ to P4 and P5 after it is calculated. P4 and P5 then complete their computations and send the results for rows 8 and 9 to P3. For good efficiency, there should be sufficiently many entries in the region to justify the communication and other overheads.

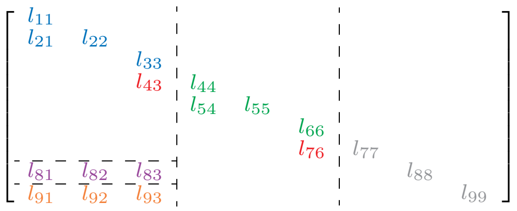

Fig. 3. Sparse matrix example 3.

These three strategies — sending data earlier to achieve greater overlap, identifying independent rows, and parallel processing of dense off-diagonal regions — form the basis of our algorithm.

III. Parallel Algorithm

Data decomposition and format: We assume that the basic units of parallelism are blocks of columns, which are distributed in round-robin fashion among processors for better load balance. We also assume that each block is stored in a format that allows easy access to the rows, such as compressed sparse row (CSR) format. This mixture of column and row views of the matrix results in manageable pieces of data on each processor (a local row in this case), enabling the algorithm to keep track of inter-block dependencies with low overhead.

Global matrix information: The global information (e.g. parallel communication data structures) about the matrix needed for our algorithm on each unit of parallelism (holding a block of columns) is comparatively small. For each local row, the algorithm needs to know whether it depends on any block to the left of this block. In addition, for each off-diagonal local row, it needs to know which blocks to the right depend on that row. This information is usually known or can be obtained from the previous stages of the application, such as symbolic factorization phase of an incomplete factorization algorithm.

Summary of algorithm: Algorithm 1 gives a high-level view of our method. In summary, the diagonal rows of each column-block are first reordered for better parallelism by identifying independent rows, as described in Algorithm 2. Next, the nonzeros below the diagonal block are inspected for various structural properties. If there are "many" nonzeros below the diagonal region, then the off-diagonal rows are divided and packed into new blocks. These off-diagonal blocks are essentially tiles of the matrix and act as independent units of parallelism similar to the regular column-blocks. However, they depend on their "master" block to send them the required solution subvector before they can process any of their elements, even if the potential data dependencies from the blocks to their left are resolved. These new blocks are sent to processors in round-robin fashion to create more parallelism.

```
Algorithm: ParallelSolve
// Blocks of matrix columns distributed to
   processors
Input: Row myRows[], Value myRhs[]
Output: x[], solution of sparse triangular system
// We know which rows depend on other blocks
reorderDiagonalBlock(myRows)
inspectOffDiagonalRows(myRows)
if many nonzeros in off-diagonal rows then
    create new blocks and send them to other processors
end
while more iterations needed do
    triangularSolve(myRows, myRhs)
end
```

Algorithm 1: Parallel Solution of Triangular Systems

Algorithm 2 describes the reordering step, in which independent rows are identified so that they can be processed without any data required from other blocks. Independent row in the diagonal block means that it has no nonzero to the left of the block, and it does not depend on any dependent row. For instance, row 6 of Figure 1 is independent because it has no nonzero to the left. On the other hand, row 5 is dependent; it has no nonzero to the left of the block, but it depends on the fourth row through l_{54}. The first loop finds and marks any dependent rows. The second loop copies the independent rows to a new buffer in backward order. We reverse the order of independent rows in the hope of computing the dependencies of subsequent blocks sooner. This heuristic has enhanced performance significantly in our test results. Note that since the rows are copied in backward order, in many cases we need to copy some previous rows to satisfy each row's dependencies in a recursive routine.

```
Algorithm: reorderDiagonalBlock
Input: Row myRows[]
for r in myRows (forward order) do
    r.depend ← false
    // nonzeros in left blocks
    if r depends on other blocks then
        r.depend ← true
    end
    for each nonzero e in r do
        // (row number of s = column number of e)
        if row s corresponding to e is dependent then
            r.depend ← true
        end
    end
end
for r in myRows (backward order) do
    // recursion needed in backward order to
       maintain dependencies
    if r.depend = false and r not already copied then
        copyRowRecursive(r)
    end
end
for r in myRows (forward order) do
    // no recursion required in forward order
    if r.depend = true then
        copy r to new buffer
    end
end
Result: myRows reordered for more parallelism
```

Algorithm 2: Reorder Diagonal Block

IV. Implementation in Charm++ and MPI

To test its effectiveness in practice, we implemented our triangular solver algorithm using Charm++ [13].[2] Other

[2] Our benchmark code can be downloaded from *https://charm.cs.illinois.edu/benchmarks/triangularsolver.git* repository.

```
// if this chare has some diagonal part of matrix
if (onDiagonalChare) {
  // schedule the independent computation with lower priority
  serial {thisProxy[thisIndex].indepCompute(...)}
  // "while" and "when" can happen in any order
  overlap {
    // while there are incomplete rows, receive data
    while (!finished) {
      when recvData(int len, double data[len], int rows[len])
        serial {if(len>0) diagReceiveData(len, data, rows);}
    }
    // do serial independent computations scheduled above
    when indepCompute(int a) serial {myIndepCompute();}
  }
  // if chare doesn't have diagonal part of matrix
} else {
  // wait for x values
  when getXvals(xValMsg* msg) serial {nondiag_compute();}
  // while there are incomplete rows, receive data
  while (!finished) {
    when recvData(int len, double data[len], int rows[len])
      serial {nondiagReceiveData(len, data, rows);}
  }
}
```

Fig. 4. Parallel flow of our triangular solver in Structured Dagger

distributed-memory parallel paradigms, such as MPI can be used as well (discussed later). The resulting code consists of 692 Source Lines Of Code (SLOCs), which compares favorably in complexity with SuperLU_DIST's 879 SLOCs triangular solver. Note that by using the interoperability feature of CHARM++, our code can be integrated into MPI packages, such as the PETSc toolkit for scientific computation [15].

In our implementation, blocks of columns are stored in compressed sparse row (CSR) format and assigned to *Chare* parallel objects (of a *Chare array*), which are the basic units of parallelism in CHARM++. There can be many more Chares than the number of physical processors, and the runtime system places them according to a specified mapping. We specify the built-in round-robin mapping in CHARM++ for better load balance.

Each Chare analyzes its block, receives the required data, processes its rows and sends the results to the dependent Chares. Figure 4 presents the parallel flow of the algorithm using Structured Dagger language [16] (a coordination sub-language of CHARM++). This code is most of the parallel part of the actual implementation. In short, *serial* marks the serial pieces of code (written in C/C++). *When* waits for the required incoming message before executing its statement. *Overlap* contains multiple Structured Dagger statements that can execute in any order. Overall, this language simplifies the implementation of the parallel control flow of our algorithm significantly.

The analysis phase needs to know only which of its rows are dependent on the left Chare, which can be determined by a parallel communication step or prior knowledge (e.g., symbolic factorization phase) as mentioned before. In our implementation, the analysis phase determines whether there is a dense off-diagonal region that can be broken into blocks for more parallelism, in which case new Chares are created and assigned to processors by the runtime system according to the specified mapping.

We use over-decomposition (multiple Chares per processor), message priorities, and message aggregation features of CHARM++ to improve the performance.

Implementation in MPI: Our algorithm can be implemented in MPI, perhaps with somewhat greater effort than for the CHARM++ version. The major difficulty is mapping and managing multiple blocks of columns per processor and creating the effect of virtualization. Allocating blocks dynamically also seems harder. In addition, the priority of data messages over computation can be implemented using *MPI_Iprobe()*. When the computation of a block is completed, *MPI_Iprobe()* could be used to determine whether any data message is available. If a message is available, then it is processed before moving to the next computation. *Wildcards* may also be needed for specifying the source. Other parts of the implementation (e.g., message aggregation) are straightforward, and there would be little difference.

V. TEST RESULTS

In this section, we evaluate our implementation for up to 512 cores on BlueGene/P. We benchmark the time for one solution iteration with one right-hand side.

Our sequential algorithm is just the standard nested loops, without any significant overhead. Thus, our speedups are measured against the best sequential case.

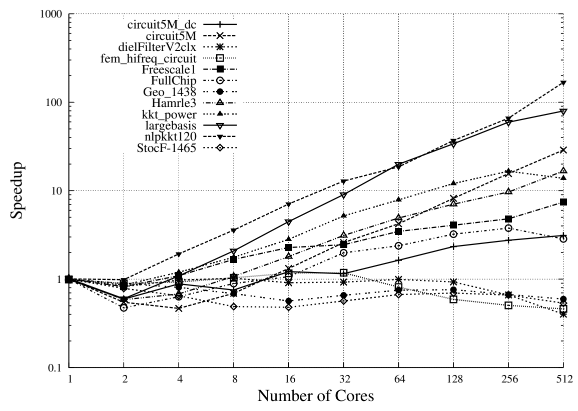

Fig. 5. Scaling for ILU(0) matrices (higher is better).

Scaling for ILU(0) matrices: Figure 5 illustrates the scaling of our implementation for up to 512 cores of BlueGene/P using triangular matrices from incomplete LU factorization with no fill. Since the matrices are small relative to the number of cores used, the results represent strong scaling of this approach. Matrix *nlpkkt120* shows the best scaling and achieves speedup of 166 on 512 cores. This is because its structure allows parallel and pipelined execution and it is larger than the other matrices (about 50 million nonzeros). Matrix *largebasis* also scales to 512 cores with a speedup of more than 78. Some matrices, such as *Hamrle3* and *kkt_power*, show good parallelism initially, but the speedup declines for larger numbers of cores. The reason is that the parallelism and matrix size are insufficient to exploit the processing power, so parallel overhead become relatively more costly. Some other matrices, such as *FullChip* and *circuit5M_dc*, show limited parallelism and need much larger matrix sizes to show good speedup. A few matrices, such as *Geo_1438* and *StocF-1465*, do not show any parallelism, and the execution time increases with more cores. However, their execution time is worse than sequential by only a small constant (roughly two), which shows the low overhead of the algorithm in the worst case. For these matrices and their application domains, new methods are needed.

Scaling for complete LU matrices: Figure 6 presents the scaling of our method for up to 512 cores of BlueGene/P using triangular matrices from complete LU factorization. There are cases with superlinear speedup because of cache effects.

For example, matrix *slu_nlpkkt80* achieves speedup of 87 on 64 cores. Many matrices scale well up to 32 or 64 cores, but performance decreases beyond that point. This is mostly because the matrices are small relative to the number of cores. For instance, matrix *slu_c-big* has only 500k nonzeros that occupy only about 5MB of memory in total. However, it achieves speedup of more than 40 on 64 cores. By reordering, this matrix is mostly parallel, with few dependencies. Thus, the parallel overheads are relatively high in this case and should be alleviated in production implementations.

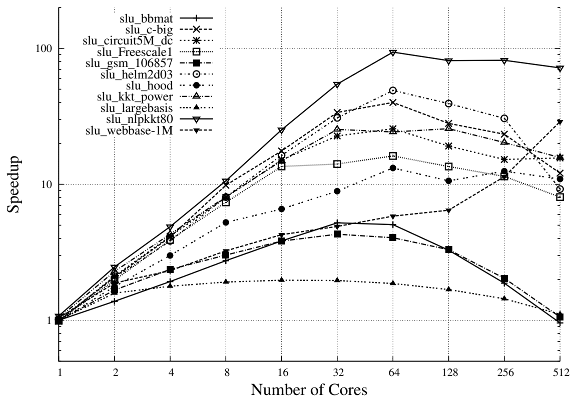

Fig. 6. Scaling for complete LU matrices (higher is better).

Comparison with HYPRE: Figure 7 compares the performance of our method with that of HYPRE, which is a commonly used linear algebra package [12]. As shown, our method can exploit parallelism on many matrices, whereas HYPRE's performance is nearly sequential in all cases. Although the triangular solver in HYPRE includes some minor optimizations, it works essentially sequentially among the processors. Each processor performs its computations and sends the results to the next one, so the processors form a chain. The choice of this method for the such a widely used package illustrates the ineffectiveness of previous parallel approaches for this problem. The performance of HYPRE is worse than sequential in many cases because of parallel overhead, although there is some improvement for large numbers of processors, probably due to cache effects. Overall, our method is a significant improvement over this existing code and will reduce the solution time for many problems.

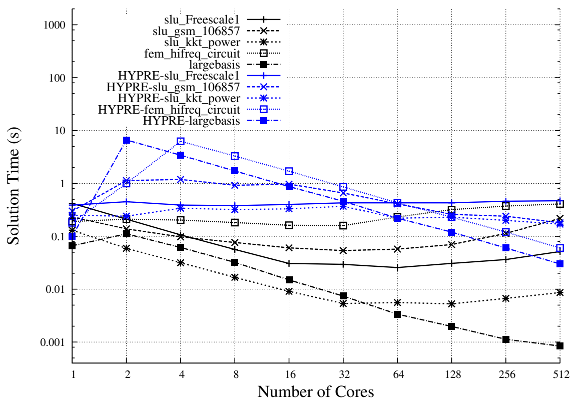

Fig. 7. Comparison with triangular solver from HYPRE (lower is better). HYPRE is in blue, and our solver is in black.

Comparison with SuperLU_DIST: Figure 8 compares the performance of our triangular solver to the triangular solver from the SuperLU_DIST package [4]. As shown, it does not exploit sufficient parallelism and the scaling is not very good, even though it has some very limited scaling with respect to its own sequential performance (e.g., 6.4 times self-speedup on 512 cores for matrix *helm2d03*). In fact, it is worse than the best serial performance for most cases. For example, SuperLU_DIST is about 18.5 times slower than the best serial performance on 64 cores for matrix *slu_helm2d03*, whereas our solver achieves a speedup of more than 48. SuperLU_DIST uses a simple 2D decomposition approach for parallelism, which is inefficient. Our method significantly improves triangular solution and refinement after complete LU.

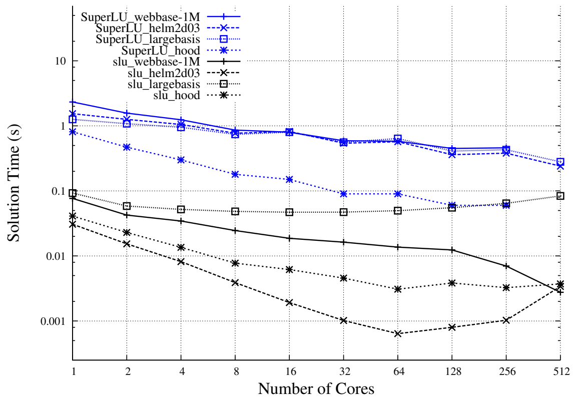

Fig. 8. Comparison with triangular solver from SuperLU_DIST (lower is better). SuperLU_DIST is in blue, and our solver is in black.

VI. Conclusions and Future Work

Solving sparse triangular linear systems is an important kernel for many numerical methods used in applications. It is not easy to implement efficiently in parallel, however, because of its dependencies and small amount of work per data. We presented a novel algorithm based on heuristics that strives to extract most of the parallelism available in the matrix. It uses low-cost analysis and row reordering to prepare for efficient execution. As opposed to previous methods, our algorithm does not rely on repeated data redistributions and many global synchronizations. We presented promising performance results on up to 512 cores of BlueGene/P for numerous sparse matrices from real applications.

For future studies, mapping, virtualization ratio, and message aggregation aspects of our algorithm need to be studied further. For matrices that do not allow significant parallelism using our algorithm, novel numerical methods that eliminate some of the nonzeros for more parallelism are needed.

References

[1] M. Heath, E. Ng, and B. Peyton, “Parallel algorithms for sparse linear systems,” *SIAM Review*, pp. 420–460, 1991.

[2] J. I. Aliaga, M. Bollhfer, A. F. Martn, and E. S. Quintana-Ort, “Exploiting thread-level parallelism in the iterative solution of sparse linear systems,” *Parallel Computing*, vol. 37, no. 3, pp. 183 – 202, 2011.

[3] T. A. Davis, “Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse QR factorization,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 8:1–8:22, Dec. 2011.

[4] X. S. Li and J. W. Demmel, “SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems,” *ACM Trans. Math. Softw.*, vol. 29, no. 2, pp. 110–140, Jun. 2003.

[5] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[6] A. Baker, R. Falgout, T. Kolev, and U. Yang, "Multigrid smoothers for ultraparallel computing," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2864–2887, 2011.

[7] J. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM J. Sci. Stat. Comput.*, vol. 11, p. 123, 1990.

[8] J. Mayer, "Parallel algorithms for solving linear systems with sparse triangular matrices," *Computing*, vol. 86, pp. 291–312, 2009.

[9] M. Wolf, M. Heroux, and E. Boman, "Factors impacting performance of multithreaded sparse triangular solve," in *High Performance Computing for Computational Science VECPAR 2010*, ser. Lecture Notes in Computer Science, J. Palma, M. Dayd, O. Marques, and J. Lopes, Eds. Springer Berlin / Heidelberg, 2011, vol. 6449, pp. 32–44.

[10] N. J. Higham and A. Pothen, "Stability of the partitioned inverse method for parallel solution of sparse triangular systems," *SIAM J. Sci. Comput.*, vol. 15, no. 1, pp. 139–148, 1994.

[11] P. Raghavan, "Efficient parallel sparse triangular solution using selective inversion," *Parallel Processing Letters*, vol. 8, no. 1, pp. 29–40, 1998.

[12] R. Falgout, J. Jones, and U. Yang, "The design and implementation of Hypre, a library of parallel high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, ser. Lecture Notes in Computational Science and Engineering, A. M. Bruaset and A. Tveito, Eds. Springer Berlin Heidelberg, 2006, vol. 51, pp. 267–294.

[13] L. V. Kale and G. Zheng, "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects," in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[14] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[15] S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang, "PETSc Web page," 2001, http://www.mcs.anl.gov/petsc.

[16] L. V. Kale and M. Bhandarkar, "Structured Dagger: A Coordination Language for Message-Driven Programming," in *Proceedings of Second International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 1123-1124, September 1996, pp. 646–653.