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, Urbana, IL 61801, USA
E-mail: {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 even extract parallelism in some cases where most of the nonzero matrix entries are near the diagonal. We describe the implementation of our algorithm in Charm++ and MPI and present promising results on up to 512 cores of BlueGene/P, using numerous sparse matrices from real applications.

I. 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 [1], [2], [3], that arise in many science and engineering simulations. It is used extensively in direct methods, following a triangular factorization, to solve the system, possibly with many right-hand sides, or to refine an approxmiate solution iteratively [4]. It is also a fundamental kernel in many iterative methods (such as Gauss-Seidel method) or in many preconditioners for other iterative methods (such as Incomplete-Cholesky before Conjugate Gradient) [5]. Unfortunately, the parallel performance of triangular solution is notoriously poor, so it is a performance bottleneck for many of these methods.

Solution of sparse triangular systems is particularly resistant to efficient use of parallelism 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. Thus, the communication cost are high compared with the computation, especially on distributed-memory parallel computers.

Despite the apparent lack of parallelism and relatively high communication overhead, sparse triangular systems are nevertheless usually solved in parallel, both for memory scalability and because the matrix is typically already distributed across processors from a previous computation (e.g., factorization). This is probably why some standard packages implement triangular solution in parallel even though its parallel performance may be much slower than sequential computation, as we will observe later. Thus, it is desirable to achieve as much efficiency as possible in parallel triangular solution, especially in view of the many iterations often required that can dominate the overall solution time. Our algorithm improves the performance of parallel triangular solution and provides good speedups for many matrices, even with strong (i.e., fixed problem size) scaling.

Previous work on this problem has focused on two main directions. First, various techniques, such as dependency analysis and partitioning, have been employed to exploit sparsity and identify parallelism [6], [7], [8]. For example, a level-set triangular solver constructs a directed acyclic graph (DAG) capturing the dependencies among rows of the matrix. Then it computes 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 [7], [8]. Second, partitioning the matrix into sparse factors and inversion is the basis of another class of methods [9], [10]. 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 [11], because of their limited performance.

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. Another good 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++[12] 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 [13]) 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 [11] and SuperLU_DIST [4] packages to show the superiority of our algorithm to current standards.

II. Parallelism in Sparse Triangular Solution

In this section, we use examples to illustrate various opportunities for parallelism that we exploit in our algorithm. Computation of the solution vector x to 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 shows 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 $x_2, \ldots, x_7$. Similarly, 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.

Continuing with our example, 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, then 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. Some overlap could be achieved by having P1 send the result from l_{43} before processing row eight, so that P2 can start its computation earlier. But sending data as they become available allows only limited 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 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, as shown 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, 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.

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

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

This idea applies to some practical cases, but may not provide any benefit for others. For example, Figure 3 shows a matrix with its diagonal and subdiagonal full of nonzeros, which implies a chain of dependencies between rows, and the computation is essentially sequential.

$$\left[\begin{array}{ccccccccc}
l_{11} & & & & & & & & \\
l_{21} & l_{22} & & & & & & & \\
 & l_{32} & l_{33} & & & & & & \\
 & & l_{43} & l_{44} & & & & & \\
 & & & l_{54} & l_{55} & & & & \\
 & & & & l_{56} & l_{66} & & & \\
 & & & & & l_{76} & l_{77} & & \\
 & & & & & & l_{87} & l_{88} & \\
 & & & & & & & l_{98} & l_{99}
\end{array}\right]$$

Fig. 3. Sparse matrix example 2.

Our previous example matrices had most of their nonzeros on or near the diagonal. Matrices from various applications have a wider variety of nonzero structures and properties.

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

Fig. 1. Sparse matrix example 1.

Another common case that may provide opportunities for parallelism is having some denser regions below the diagonal block. Figure 4 shows 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 overhead.

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

Fig. 4. 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.

III. Parallel Triangular Solution Algorithm

We now describe our sparse triangular solution algorithm in greater detail. Algorithm 1 gives a high-level view of our method. 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 format. The rows of each diagonal 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, they are packed into new blocks and sent to other processors. Here, "many" means that the communication and other overheads are justified by the computation in the block, and this presents a parameter to tune. In our implementation, if the nonzeros are more than some constant (20 seems to work well) times the size of the solution subvector, we send the block to another processor. After this precomputation is done, we can start solving the system (described in Algorithm 4), possibly multiple times, as may be needed.

```
Algorithm: ParallelSolve
// Matrix columns distributed to
   processors round-robin in blocks
Input: Row myRows[]
Output: x[], solution of sparse triangular system
// We know which rows depend on other
   blocks
reorderDiagonalBlock(myRows)
inspectBelowDiagonalBlock(myRows)
if many nonzeros below diagonal block then
    create new blocks and send to other processors
end
while more iterations needed do
    triangularSolve(myRows)
end
```

Algorithm 1: Parallel Solution of Triangular System

Algorithm 2 describes the reordering step, in which independent rows are identified so that they can be computed 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, l_{66} 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 places the independent rows on 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.

The PlaceRow routine described in Algorithm 3 inspects all the nonzeros of a given row to make sure that the needed rows are already placed and it then places the row. If a needed row is not placed, the routine calls itself recursively to place it. It also marks the row as placed, so that it will not be placed again. For simplicity, we assume that there is another buffer to contain the rows, but if memory is constrained, then the rows can be interchanged to reorder them. The final loop places the dependent rows in forward order, without regard for their inter-dependencies.

```
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 in r do
        if corresponding row is dependent then
            r.depend ← true
        end
    end
end
for r in myRows (backward order) do
    if r.depend = false and r not already placed then
        placeRow(r)
    end
end
for r in myRows (forward order) do
    if r.depend = true then
        add r to new buffer
    end
end
Result: myRows reordered for more parallelism
```

Algorithm 2: Reorder Diagonal Block

```
Algorithm: placeRow
Input: Row r
for e in nonzeros of r (reverse order) do
    row e_r ← row containing $x$ value for $e$
    if e_r not already placed then
        placeRow(e_r)
    end
end
add r to new buffer
Result: r is placed in first possible position of new buffer
```

Algorithm 3: Place Row on New Buffer

Algorithm 4 performs the local solution for each block. Initially, the messages received are processed as described in Algorithm 6, before starting the computation, in the hope of performing work on the critical path and sending the results sooner. Receiving messages before starting the computation is possible when there are multiple blocks per physical processor. The routine then processes the independent rows (described in Algorithm 5) and waits for messages until all the rows have been completed.

```
Algorithm: triangularSolve
Input: Row myRows[]
Output: Values x[]
if any DataMessage msg arrived then
    receiveDataMessage(msg)
end
for each Row r in independent rows do
    computeRow(r,0)
end
while there are pending rows do
    wait for DataMessage msg
    receiveDataMessage(msg)
end
```

Algorithm 4: Local Triangular Solve

Algorithm 5 describes the computation for each row. The input value (“val”) is the partial sum from the left of the block and is updated using the nonzeros of the row and the needed x values. For the diagonal rows (rows in diagonal blocks), x_i, which is the x entry corresponding to “*r*”, is computed. If x_i is the last variable needed for the subdiagonal rows that are needed for the next block, they are computed (if they were local to the block). This accelerates the dependencies of the next block and probably the critical path. If x_i is the last variable needed for the subdiagonal rows and they are local, they are processed. If they are not local, the x subvector is broadcast to the subdiagonal blocks. For offdiagonal rows, the updated value is sent to the block waiting for it.

Algorithm 6 describes the processing of data messages. For each value in the message, the local row that is waiting for it is determined. This may require a mapping of global row number to local row number (such as hash table), or this information can be communicated once during the analysis stage. If the row is diagonal and it is the current row (the first incomplete row), it is processed. In this case, all the following rows that are not still depending on outside data are processed as well until a depending one is discovered. Those rows can be processed since all of their dependencies are satisfied by processing them in order. If the row is offdiagonal and the needed x values are ready, it can be computed as well. However, if it is diagonal but not the current row, or offdiagonal but x is not ready, then *val* is stored for later use.

IV. Implementation in Charm++ and MPI

To test its effectiveness in practice, we have implemented our triangular solution algorithm using Charm++[12]. 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. There can be many

```
Algorithm: computeRow
Input: Row r, Value val
update val using nonzeros of r and computed x[] values
if r is diagonal then
    compute $x_i = (b-val)/l_{ii}$
    if below diagonal rows for next block are local and x is last needed variable for next block then
        compute below diagonal rows for next block and send data values
    end
    if $x_i$ is last needed variable for other blocks then
        if below diagonals are local then
            compute below diagonal rows for next block and send data values
        end
        else
            broadcast all x values so far to below diagonal blocks
        end
    end
end
else
    send val to depending block
end
Result: if diagonal row: x value computed, if offdiagonal row: data value sent to next block
```

Algorithm 5: Process Local Row

```
Algorithm: receiveDataMessage
Input: DataMessage msg
for each data Value val in msg do
    Row r ← row corresponding to value
    if r is diagonal and is current pending row then
        computeRow(r,val)
        while next Row n_r is not outside dependent do
        computeRow(n_r,0)
    else if r is offdiagonal and x variables ready (locally or received) then
        computeRow(r,val)
    else store value
end
Result: message used for computation or stored
```

Algorithm 6: Receive Data Message

more Chares than the number of physical processors, and the runtime system places them according 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. The analysis phase needs to know only which of its rows are dependent on the left Chare, which can be determined by communication or prior knowledge (e.g., symbolic factorization phase). In our implementation, the analysis phase determines whether there is a dense offdiagonal region that can be broken into blocks for more parallelism. It then *creates* new Chares, which will be placed on the processors by the runtime system according to the mapping.

Creating new parallelism units, independent of the fixed number of physical processors, is an abstraction that is useful for simplicity of the implementation of our structure-adaptive algorithm. However, the number of Chares (blocks) can be determined in advance based on knowledge of the matrix structure. For example, it can be determined in the symbolic factorization phase of LU or Cholesky factorization.

Virtualization ratio: The ratio of the number of Chares to the number of physical processors (*virtualization ratio*) is an important parameter since it presents a tradeoff between communication overlap and concurrency. If the virtualization ratio is large, then when some blocks are waiting for data to be received, others can still make progress in computation. On the other hand, if the matrix is divided too finely into small blocks, many nonzeros of diagonal blocks may fall in other blocks and create many dependent rows. Thus, the concurrency might be compromised because each block has fewer independent rows to compute. In our implementation, we use a virtualization ratio of four, which seemed to provide a good balance between communication overlap and concurrency.

Message priority: We use message priorities of CHARM++ to make more rapid progress on the critical path of the computation. Potentially, there may be a chain of dependencies along the diagonal of the matrix. Therefore, we give higher priority to data messages than the computation of other Chares. This means that when the computation of a Chare is completed, the runtime system tries to choose data messages over computation of other Chares. This may provide data for some critical computation that will send enabling data messages to other Chares. More sophisticated message priority approaches that use more information from the structure of the matrix are the subject of future work, but may be subject to diminishing returns.

Sequential kernels: Efficient sequential kernels are highly important for this problem, since the computation is very small relative to the amount of data. Therefore, most overheads, especially cache inefficiencies, are intolerable. Furthermore, using high-performance sequential and shared-memory node kernels inside the distributed-memory code can improve its performance significantly. For these reasons, in our implementation, we compute the rows in large chunks without interruptions. For example, we keep track of the number of of rows that are required before sending the next block's data in the analysis phase. In the computation, we process all of those rows as a chunk and then send the data afterwards. In this way no checking ("if" statement) is required after each row, though it was presented that way in the algorithm statement in the previous section for simplicity. The only change required in the sequential kernel is adding the result sum from the left of the row, which is easy to include. Thus, efficient sequential and shared-memory kernels can be used readily.

Aggregating data messages: Depending on the nonzero pattern of the matrix, an offdiagonal row may need to send its data to any block to its right. However, due to message startup and receive overheads, it is inefficient to send one data element at a time (one floating-point value in this case). Therefore, these data elements are aggregated into larger messages before sending over the network. This is possible since the neighboring rows have "locality" and are likely to send to the same blocks. Thus, we allocate buffers for different destinations and gather the data in them. We flush a buffer and send data when it is full, or when all buffers are allocated and we need to allocate a new one. However, gathering the data into buffers presents a tradeoff since delaying sending a message might delay progress on the critical path. Thus, we flush the buffers and send the data out at various stages of the algorithm to ensure faster progress. Message aggregation approaches that minimize both the message overhead and the delay can also be used [14].

Implementation in MPI: In principle, every CHARM++ program can be implemented in MPI, since CHARM++ itself can be built on top of MPI, although the programming effort might make it impractical in some cases. We believe that 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, which can make the code error prone. Allocating blocks dynamically also seems challenging.

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. If minimizing programming effort is the goal, these two major changes can be ignored and the algorithm can be implemented without multiple blocks per processor. Thus, some performance enhancements resulting from the overlap of computation and data dependencies will not be available. However, the major benefit of the algorithm, the analysis and reordering, can still be realized.

Tuning parameters: As mentioned, there are various parameters to choose in our algorithm, such as virtualization ratio, message priorities and buffer size for message aggregation. Tuning these parameters carefully might result in performance improvements. Thus, tuning methods specific to this algorithm and choosing the values based on the structure of the matrix and the machine is a subject for future research. In addition, automatic tuning approaches such as Control Points [15] in CHARM++ framework can be used. However, these parameters do not seem to be very sensitive for performance so we set some values manually for all the experiments discussed here.

V. Test Results

In this section, we evaluate our implementation for up to 512 cores on BlueGene/P. We benchmark the time of one solution iteration with one right-hand side, without the cost of benchmarking barriers. In some cases, barriers might be necessary for the application, but their cost is insignificant if the matrix is sufficiently large. Furthermore, the application might be able to overlap different iterations and fill processor idle times with useful work to attain higher performance.

Our sequential algorithm is just the standard nested loops, without any significant overhead. Note again that on only one processor core our algorithm boils down to this efficient sequential algorithm. Thus, our speedups are measured against the best sequential case. Note also that BlueGene/P's processors are low power by design, so they are somewhat slower than some other mainstream processors.

Test Problems: We first describe our test problems, which are drawn from several real application sparse matrices from the University of Florida Sparse Matrix Collection [13]. Table I lists these matrices and their properties. Those prefixed with "slu_" are obtained from a complete LU factorization using SuperLU. Note that the matrices used are fairly small relative to the number of processors used. Some of the matrices are even smaller than those in a recent study of shared-memory codes [8]. Thus, our results provide a reasonable indication of the strong scaling ability of our triangular solution algorithm.

There are two measures that can help in understanding the parallelism available in each matrix: (1) after the reordering and analysis phases of our algorithm, the total number of rows (across all processors) that can be computed independently in parallel, and (2) the number of nonzeros that are in nondiagonal blocks. The first metric is a direct measure of parallelism, while the second may or may not indicate better parallelism. If the nonzeros are close to the diagonal blocks and they are spread apart, it is more difficult to have parallelism. However, if they are in dense regions and far from the diagonal, they probably can be computed in parallel.

Figure 5 shows the nonzero structure of some of the matrices we use. We will use this figure, along with Table I, to help understand the performance behavior for these matrices.

Scaling for no-fill LU matrices: Figure 6 shows 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

TABLE I
BENCHMARK MATRICES

Name	Dimension	Independent rows	Nonzeros	In nondiagonal blocks	Application domain
circuit5M_dc	3,523,317	674,311	10,631,719	4,110,848	circuit simulation
circuit5M	5,558,326	333,841	32,542,244	26,616,437	circuit simulation
dielFilterV2clx	607,232	4,965	12,958,252	7,824,540	electromagnetics
fem_hifreq_circuit	491,100	8,744	10,365,173	7,321,726	electromagnetics
Freescale1	3,428,755	2,153,121	11,901,587	5,963,982	circuit simulation
FullChip	2,987,012	12,982	14,804,570	8,126,422	circuit simulation
Geo_1438	1,437,960	5,617	32,297,325	17,912,293	structural analysis
Hamrle3	1,447,360	746,720	3,032,733	1,582,170	circuit simulation
kkt_power	2,063,494	811,213	8,545,814	5,549,454	optimization
largebasis	440,020	200,010	3,000,060	2,560,040	optimization
nlpkkt120	3,542,400	1,814,400	50,194,096	46,651,696	optimization
StocF-1465	1,465,137	34,822	11,235,263	5,609,744	fluid dynamics
slu_bbmat	38,744	6,735	17,819,183	15,762,657	fluid dynamics
slu_c-big	345,241	345,141	499,807	17,038	optimization
slu_circuit5M_dc	3,523,317	3,429,272	8,027,174	332,376	circuit simulation
slu_Freescale1	3,428,755	3,329,165	12,624,349	1,079,503	circuit simulation
slu_gsm_106857	589,446	312,454	12,107,540	3,654,630	electromagnetics
slu_helm2d03	392,257	373,796	648,305	23,380	2D/3D problem
slu_hood	220,542	192,353	2,143,007	540,982	structural analysis
slu_kkt_power	2,063,494	2,043,810	3,298,181	287,311	optimization
slu_largebasis	440,020	280,483	5,095,186	1,991,169	optimization
slu_nlpkkt80	1,062,400	1,062,400	1,062,400	0	optimization
slu_webbase-1M	1,000,005	986,863	3,345,311	512,433	weighted directed graph

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.

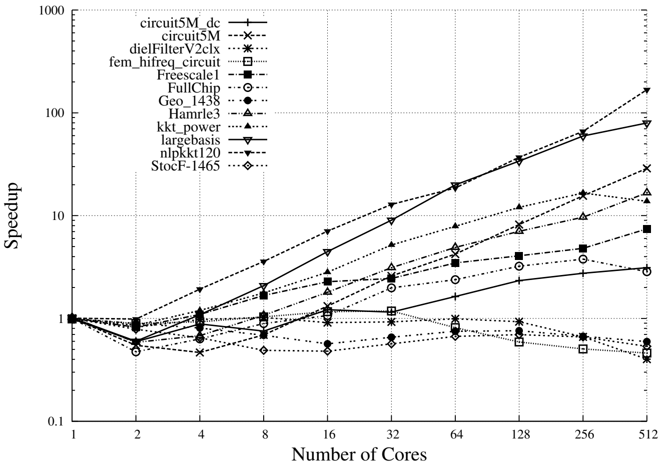
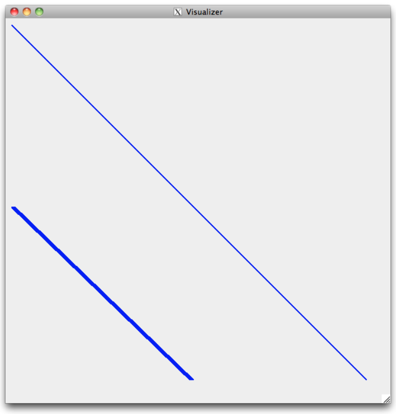


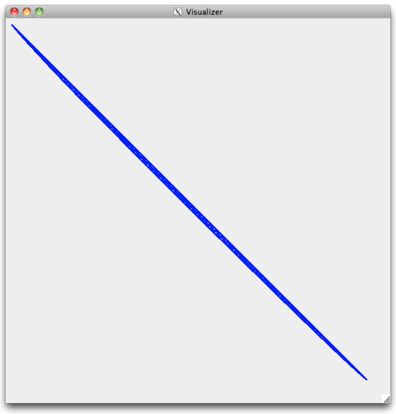
Fig. 6. Scaling for no-fill incomplete-LU matrices.

Scaling for complete LU matrices: Figure 7 shows 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. This includes better implementation of broadcast and reduction (synchronization) using the collective network of BlueGene/P, if synchronization is required for the application (for example, iterative refinement of the solution with error estimation). If synchronization is not required (for example, a fixed number of refinements), much better performance can be obtained with small changes to the implementation. In addition, communication latency is critical for solution of sparse triangular systems, because of structural dependencies and limited computation.

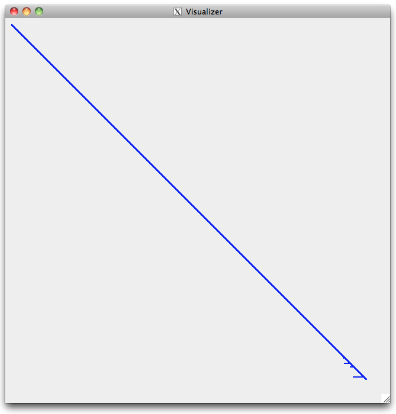
The figure also shows that matrices resulting from complete LU have a different (better on average) structure for parallelism than no-fill incomplete LU matrices. For example, slu_circuit5M_dc is much more parallel than circuit5M_dc. The reason is that SuperLU reorders the rows and elements for better factorization, so the resulting lower triangular and upper triangular matrices will have different structures. This



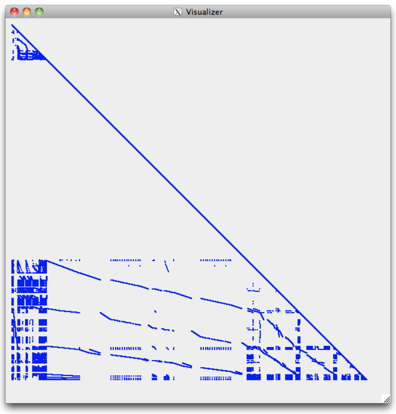
(a) nlpkkt120



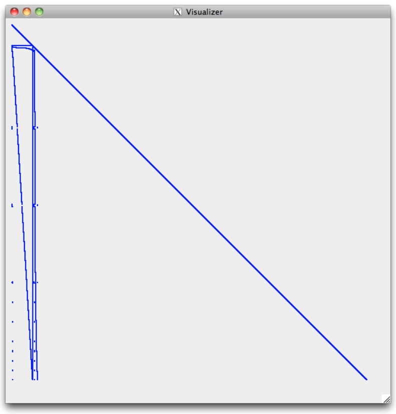
(b) Geo_1438



(c) slu_c-big



(d) Freescale1



(e) circuit5M

Fig. 5. Nonzero structure of various test matrices

strategy improves the triangular solution using our method as well.

Scaling for various matrix structures: Performance and scaling of our algorithm can vary with matrix structure. Table I and Figure 5 help in understanding the parallelism available in various matrices. For example, matrix nlpkkt120 (Figure 5(a)) enjoys the best performance on 512 cores. The reason is that its upper left portion consists mainly of independent rows. They begin computing in parallel, then they send their solution values to the nonzeros on the bottom (which form a slanted line) to compute in parallel. Those blocks send their values to the right diagonal blocks to complete the computation. Thus, there are three stages, and each stage has many parallel portions. Matrix Freescale1 also has similar parallelism opportunities, but with a different structure (Figure 5(d)). Matrix circuit5M (Figure5(e)) shows another structure with good parallelism despite having relatively few independent rows. The top left diagonal blocks enable the computation of many offdiagonal blocks on the left. Those will be processed in parallel and cause the other diagonals to complete in parallel.

On the other hand, matrix Geo_1438 shows poor scaling because it has little parallelism available. Most of its nonzeros are near the diagonal, but the rows are dependent on each other (Figure 5(b)). Most of the matrices with poor scaling have similar structure. Creating parallelism by numerical methods (such as dropping some nonzeros) is the subject of future study. Note that having the nonzeros near the diagonal does not necessarily result in limited parallelism. For instance, matrix slu_c-big has similar structure but shows good scaling, since many of its rows are independent after reordering.

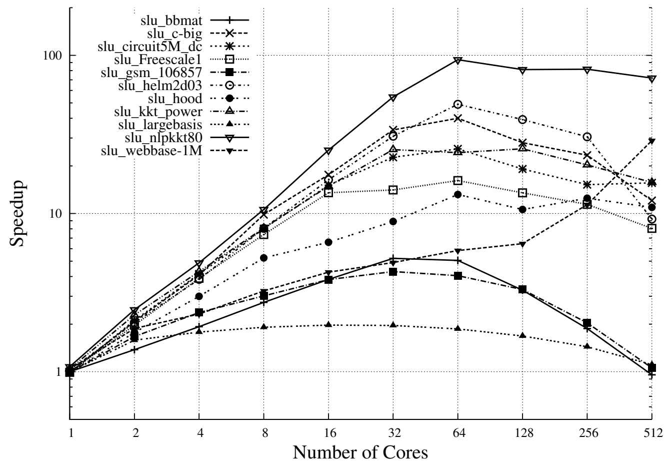


Fig. 7. Scaling for complete LU matrices.

Comparison with HYPRE: Figure 8 compares the performance of our method with that of HYPRE, which is a commonly used linear algebra package [11]. As shown, our method can exploit parallelism on many matrices, whereas HYPRE's performance is nearly sequential in all cases. The triangular solution in HYPRE 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 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.

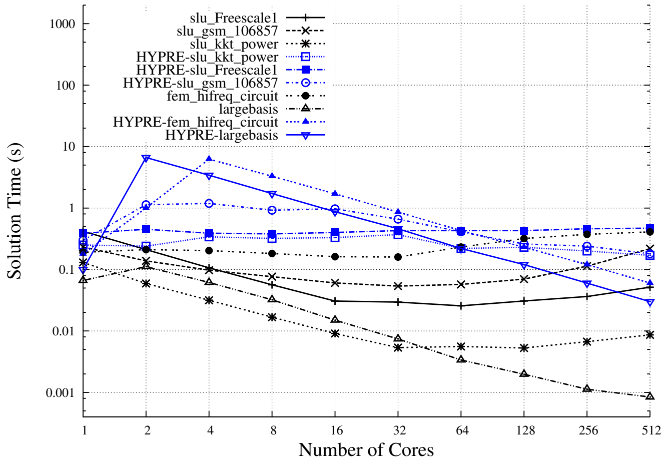


Fig. 8. Comparison with triangular solver from HYPRE.

Comparison with SuperLU_DIST: Figure 9 compares the performance of our triangular solver to the triangular solver from the SuperLU_DIST package [4]. This solver is called after factorization of the matrix, sometimes several times to refine the result or for other purposes. As shown, however, it does not exploit enough 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 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. Because refinement will be much faster, less accurate but faster factorizations may also become possible.

Comparison with other approaches: There are other algorithms for triangular solution, mostly for shared memory machines. However, it does not seem to be practical to adapt them for distributed memory machines. For example, the DAG approaches have limited concurrency [8], so they cannot scale to many cores of a distributed-memory machine. In addition, there can be thousands of barriers for some small matrices [8], so the barriers are a bottleneck even for shared-memory machines. Thus, we do not attempt to adapt and compare with them here.

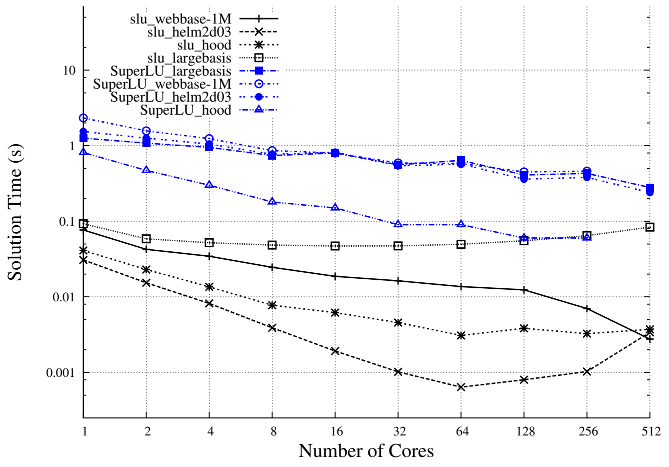


Fig. 9. Comparison with triangular solver from SuperLU_DIST.

Memory scaling: Memory scaling is important for many numerical algorithms. For our algorithm, there are only few additional scalable data structures per processor, other than the matrix itself. The largest is a data structure that has only a few entries for each local row, which contains information such as if the row is dependent. Importantly, there is no overhead per matrix element. Thus, the memory consumption is scalable and very large problem sizes are possible to solve.

Data redistribution: Since triangular solution is usually used in the context of other algorithms such as factorization, data redistribution is required when the data layouts do not match. However, it is negligible in most cases and this can be seen by some simple back-of-the-envelope calculations. For instance, matrix circuit5M is less than 300MB in memory. On 512 BGP processors, each solve iteration takes around 28ms. If each processor has to send 1MB of data, data redistribution will take less than 15ms (in parallel). Data redistribution happens only once, and this cost is amortized over many (often 100 or more [8]) iterations.

Analysis performance: Analysis time is an overhead that must be paid for many approaches to sparse triangular solution. It is negligible if it is performed only once, followed by sufficiently many iterations. In our algorithm, analysis is performed fully in parallel and independently on different processors. Thus, the analysis also scales with the number of processors. In addition, analysis reorders only the rows, based on a simple scan of rows and nonzeros, which is relatively inexpensive. Figure 10 compares the analysis time with the solution time for a sample of matrices using various numbers of processors. As can be seen, the analysis time is comparable to the solution time, and it is less than that in most instances. Thus, analysis time is negligible for applications with multiple solution iterations. Even for applications with only one solution iteration, our algorithm (with the analysis time added) performs much better than the packages we compared with here. In this case, the solution can be thought of as a constant times slower, with the constant usually less than two. Thus, analysis time is not a problem for the performance of our algorithm, so we do not strive to accelerate it further for now.

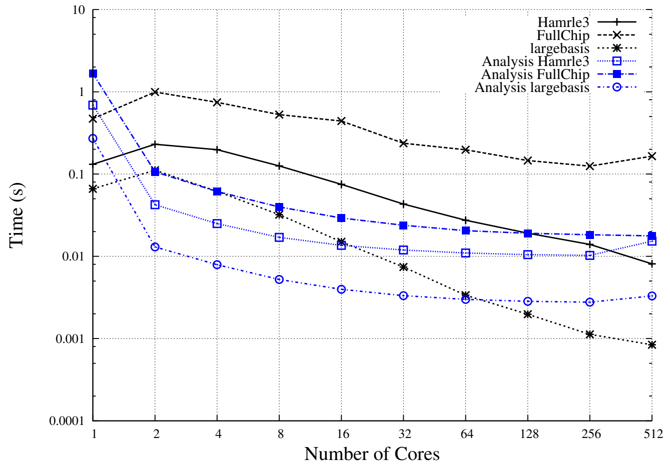


Fig. 10. Comparison of analysis and reordering time to solution time.

VI. Conclusions and Future Work

Parallel solution of sparse triangular linear systems is an important kernel for many numerical methods used in applications. For example, it is often used repeatedly in preconditioners for iterative methods. It is not easy to implement efficiently in parallel, however, especially on modern distributed-memory computers, because of its dependencies and small amount of work per data.

We presented a novel algorithm based on heuristics that strive to extract all 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, so it is suitable for large-scale distributed-memory machines. We implemented our algorithm in Charm++ and discussed its potential implementation using MPI. We saw that Charm++ provides some features that simplify the implementation.

We presented promising performance results on up to 512 cores of BlueGene/P for numerous sparse matrices from real applications. The performance depends on the parallelism available in the structure of the matrix, and we analyzed the parallelism using different metrics.

For future studies, more sophisticated methods for mapping blocks to processors and for determining priorities for processing blocks seem most important. In addition, novel techniques to determine the best virtualization ratio depending on characteristics of the matrix and the machine may improve performance. Furthermore, techniques for aggregation of messages that minimize communication overhead but do not cause delay for computation should be developed. For matrices that do not allow parallelism using our algorithm, numerical methods that eliminate some of the non-zeros for more parallelism seems to be a potentially promising approach.

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] C. Pham, “Comparison of message aggregation strategies for parallel simulations on a high performance cluster,” in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, 2000, pp. 358 –365.

[15] I. Dooley, “Intelligent runtime tuning of parallel applications with control points,” Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2010, http://charm.cs.uiuc.edu/papers/DooleyPhDThesis10.shtml.