Structure-Adaptive Parallel Solution of Sparse Triangular Linear Systems

Ehsan Totoni*, Michael T. Heath, Laxmikant V. Kale

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

Abstract

Solving sparse triangular systems of linear equations is a performance bottleneck in many methods for solving more general sparse systems. Both for direct methods and for many iterative preconditioners, it is used to solve the system or improve an approximate solution, often across many iterations. Solving triangular systems is notoriously resistant to parallelism, however, and existing parallel linear algebra packages appear to be ineffective in exploiting significant parallelism for this problem.

We develop 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.

Keywords: Triangular solver, Parallel algorithms, Sparse linear systems, Distributed memory computers

1. Introduction

Solving 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 improve an approximate solution iteratively [4]. It is also a fundamental kernel in many iterative

*Corresponding author: address: 201 N. Goodwin Avenue, Urbana, IL 61801; email: totoni2@illinois.edu; Tel: 1-217-4199843

methods (such as Gauss-Seidel method) and in many preconditioners for other iterative methods (such as Incomplete-Cholesky preconditioner for Conjugate Gradient) [5]. Unfortunately, the performance of parallel algorithms for triangular solution is notoriously poor, so they are performance bottlenecks for many of these methods.

As an 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. Thus, the triangular solves take about the same time as the Sparse Matrix Vector product (SpMV), accounting 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 is at most 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. Thus, the communication cost is 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 solves in parallel even though 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 [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]. Second, partitioning the matrix into sparse factors and inversion is the basis of another class of methods [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 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 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 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.

2. 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 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.

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

Figure 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. 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, 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 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, 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.

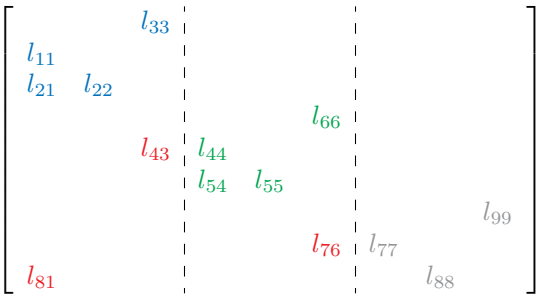


Figure 2: Reordering rows of sparse matrix example 1.

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.

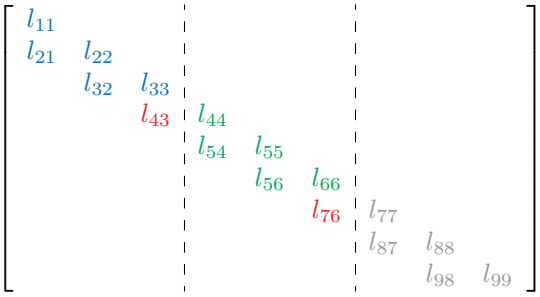


Figure 3: Sparse matrix example 2.

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. 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 overheads.

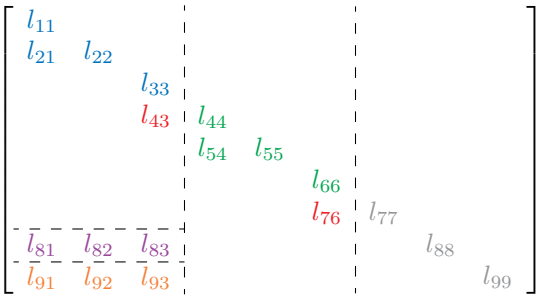


Figure 4: Sparse matrix example 3.

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

3. Parallel Algorithm

In this section we describe in greater detail our algorithm for solving sparse triangular systems.

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. Alternatively, one could use parallel decomposition by rows with Compressed Sparse Column (CSC) format. Furthermore, our algorithm can be adapted for other decomposition schemes and storage formats that allow low overhead tracking of inter-block dependencies. Investigation and evaluation of other decompositions and formats for our algorithm is left for future work.

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 it. 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. Even if this information is not present in the parallel data structures, it can be obtained easily before the analysis phase, as outlined in Algorithm 1. This algorithm is mostly distributed and does not take significant time compared to our triangular solve algorithm (either analysis or solve phases), which we confirmed experimentally in our implementation. The most important optimization required for this algorithm is message agglomeration (discussed later).

Algorithm: propagateDependencies
Input: Row *myRows*[]
initialize dependency message d
for *r in myRows* **do**
  if r *is not diagonal* **then**
    store r's index in d
  end
end
store this block's index as original sender in d
send d to next block (on right)
for *each dependency message* m *received* **do**
  for *each row* s *in* m **do**
    if s *has any nonzero in this block* **then**
      mark s as dependent
      send an acknowledgement message for s
    else
      forward dependency info of s to next block on right
    end
  end
end
for *each acknowledgement message* a *received* **do**
  `// store information to know destinations of partial sums of off-diagonal rows`
  store sender's index for row (or rows) acknowledged in a
end
`// terminate when all blocks have reached here and there is no message left in system`
terminate on quiescence
Result: Dependency information for each row in *myRows* is known

Algorithm 1: Propagate Row Dependency Information

Summary of algorithm. Algorithm 2 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 3.

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 2: Parallel Solution of Triangular Systems

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 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 5), possibly multiple times, as may be needed. More detailed explanations of these steps follow.

Algorithm 3 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, as described below.

```
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 3: Reorder Diagonal Block

The *copyRowRecusive* routine described in Algorithm 4 inspects all the nonzeros of a given row to make sure that the needed rows are already copied and it then copies the row. If a needed row is not copied, the routine calls itself recursively to copy it. It also marks the row as copied, so that it will not be copied again. The final loop copies the dependent rows in forward order, without regard for their inter-dependencies, since forward order copy satisfies them automatically. 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 without actual copies.

Algorithm 5 performs the local solve for each block. Initially, the messages

Algorithm: copyRowRecursive

Input: Row r
for *e in nonzeros of r (reverse order)* **do**
  `// (row number of s = column number of e)`
  row $s \leftarrow$ row containing x value for e
  if *s not already copied* **then**
    copyRowRecursive(s)
  end
end
copy r to new buffer
Result: r is copied in first possible position of new buffer

Algorithm 4: Copy Row to New Buffer

received are processed (as described in Algorithm 7) 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 of each block is possible when there are multiple blocks per physical processor (described in Section 4). The routine then processes the independent rows (described in Algorithm 6) and waits for messages until all the rows have been completed.

Algorithm: triangularSolve

Input: Row *myRows*[], Value *myRhs*[]
Output: Values x[]
while *any DataMessage msg arrived* **do**
  receiveDataMessage(*msg*)
end
for *each Row r in independent rows* **do**
  processRow(r, 0, *myRhs*)
end
while *there are pending rows* **do**
  wait for DataMessage *msg*
  receiveDataMessage(*msg*)
end

Algorithm 5: Local Triangular Solve

Algorithm 6 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 solution (x) values. For the diagonal rows (i.e., rows that have a diagonal element), x_i, which is the x entry corresponding to "r", is computed. If x_i is the last variable needed for the off-diagonal rows that are needed for the next block, and those rows are local to this block, they are processed. This accelerates the dependencies of the next block and

probably the critical path. If x_i is the last variable needed for all the off-diagonal rows and they are local, they are processed. If the off-diagonal rows are not local, the x sub-vector is multicast to the off-diagonal blocks that depend on it. These dependent off-diagonal blocks are the ones sent to the other processors by this block in the analysis phase according to our dense off-diagonal regions parallelism strategy (Algorithm 2). For the local off-diagonal rows, the updated partial sum value is sent to the block waiting for it. The destination is known as a result of the preprocessing step described in Algorithm 1.

Algorithm: processRow
Input: Row r, Value *val*, Value *myRhs*[]
update partial sum *val* using nonzeros of r and computed x[] values
if *r is diagonal* **then**
  compute $x_i = (myRhs_i - val)/l_{ii}$
  if *off-diagonal rows for next block are local in this block and x_i is last needed unknown for them* **then**
    call processRow() on all off-diagonal rows needed for next block that have their left dependencies satisfied
  end
  if *x_i is last needed variable for all off-diagonal rows of this column-block* **then**
    if *off-diagonals are local* **then**
      call processRow() on off-diagonal rows that have their left dependencies satisfied (partial sums arrived)
    else
      multicast the solution sub-vector (x values) computed to off-diagonal blocks dependent on this block
    end
  end
else
  `// If r is off-diagonal, send the partial sum value to the right. Destination is known from the preprocessing phase.`
  send *val* to depending block
end
Result: if diagonal row: x value computed, if off-diagonal row: data value sent to next block

Algorithm 6: Process Local Row

Algorithm 7 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 a 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 off-diagonal and the needed solution sub-vector is ready (x values are computed), it can be processed as well. However, if it is diagonal but not the current row, or off-diagonal but x is not ready, then *val* is stored for later use.

```
Algorithm: receiveDataMessage
Input: DataMessage msg
for each partial sum (Value val) in msg do
    Row r ← row corresponding to val
    if r is diagonal and is current pending row then
        processRow(r,val)
        while next row (Row s) is not outside dependent do
            processRow(s,0)
        end
    else if r is off-diagonal and x values ready (locally or received) then
        processRow(r,val)
    else
        store value for later
    end
end
Result: message used for computation or stored
```

Algorithm 7: Receive Data Message

4. Implementation in Charm++ and MPI

To test its effectiveness in practice, we have implemented our triangular solver algorithm using Charm++ [13].[1] Other 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

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

```
1  // if this chare has some diagonal part of matrix
2  if (onDiagonalChare) {
3    // schedule the independent computation with lower priority
4    serial {thisProxy[thisIndex].indepCompute(...)}
5    // "while" and "when" can happen in any order
6    overlap {
7      // while there are incomplete rows, receive data
8      while (!finished) {
9        when recvData(int len, double data[len], int rows[len])
10         serial {if(len>0) diagReceiveData(len, data, rows);}
11     }
12     // do serial independent computations scheduled above
13     when indepCompute(int a) serial {myIndepCompute();}
14   }
15   // if chare doesn't have diagonal part of matrix
16 } else {
17   // wait for x values
18   when getXvals(xValMsg* msg) serial {nondiag_compute();}
19   // while there are incomplete rows, receive data
20   while (!finished) {
21     when recvData(int len, double data[len], int rows[len])
22     serial {nondiagReceiveData(len, data, rows);}
23   }
24 }
```

Figure 5: Parallel flow of our triangular solver in Structured Dagger

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. Figure 5 shows the parallel flow of the algorithm using Structured Dagger language [16] (a coordination sublanguage of CHARM++), and roughly corresponds to Algorithm 5. 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.

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 (column 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 while 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 process. In our implementation, we use a constant virtualization ratio of four, which seemed to provide a good balance between communication overlap and concurrency for many matrices.

Message priority. We use message and computation 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 process the rows in large chunks without interruptions. For example, we keep track of the number 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 off-diagonal row may need to send its partial sum results 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 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. Other message aggregation approaches that minimize both the message overhead and the delay can also be used [17].

Implementation in MPI. In principle, any CHARM++ program can be implemented in MPI, since CHARM++ itself can be built on top of MPI, although more programming effort might be needed. We believe that our algorithm can be implemented in MPI directly, 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. 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, in which case some performance enhancements resulting from overlapping of computation and data dependencies would not be available. However, the major benefit of the algorithm, extracting parallelism by analysis and reordering, can still be realized.

Tuning parameters. As mentioned earlier, there are various parameters to choose in our algorithm, such as virtualization ratio, message priorities, and buffer size for message aggregation. We have not experimented with them extensively, but tuning these parameters carefully might result in performance improvements. Tuning methods specific to this algorithm and choosing values based on the structure of a particular matrix and machine is a subject for future research. In addition, automatic tuning approaches such as Control Points [18] in CHARM++ framework could be used. However, in our experience performance does not seem to be very sensitive to these parameters, so we set values manually for all the experiments discussed below.

5. 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, 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.

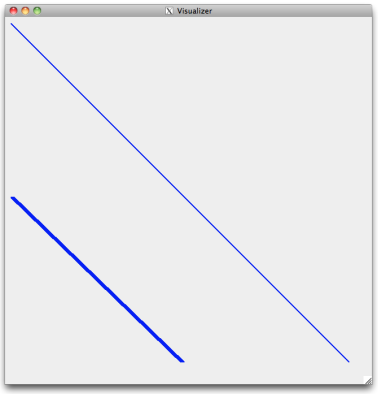
Test Problems. We first describe our test problems, which are drawn from several real application sparse matrices from the University of Florida Sparse Matrix Collection [14]. Table 1 lists these matrices and their properties. Those prefixed with "slu_" are obtained from a complete LU factorization using 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 [9]. 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 processed 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 processed in parallel.

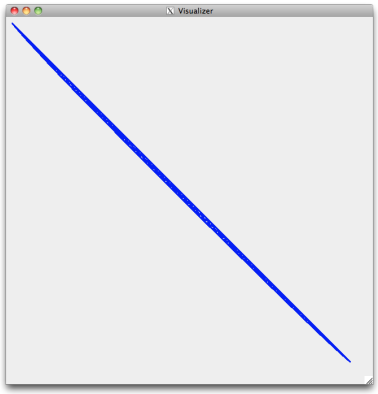
Figure 6 shows the nonzero structure of some of the matrices we use. We will use this figure, along with Table 1, to help understand the performance behavior for these matrices.

Table 1: Benchmark Matrices (sorted by the number of nonzeros)

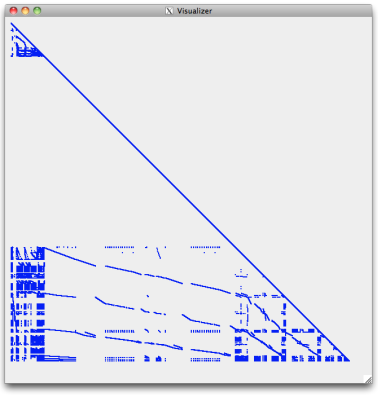
Name	Dimension	Indep. rows	Nonzeros	In nondiag. blocks	Application domain
slu_c-big	345,241	345,141	499,807	17,038	optimization
slu_helm2d03	392,257	373,796	648,305	23,380	2D/3D problem
slu_nlpkkt80	1,062,400	1,062,400	1,062,400	0	optimization
slu_hood	220,542	192,353	2,143,007	540,982	structural analysis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



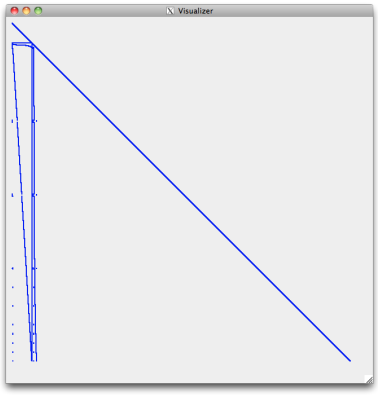
(a) nlpkkt120



(b) Geo_1438



(c) Freescale1



(d) circuit5M

Figure 6: Nonzero structure of various test matrices

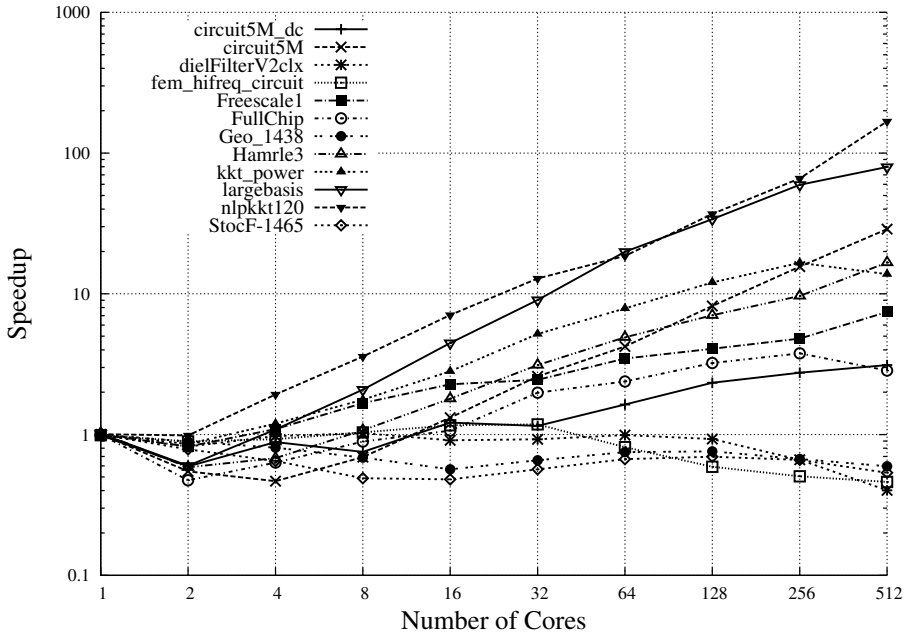


Figure 7: Scaling for ILU(0) matrices (higher is better).

Scaling for ILU(0) matrices. Figure 7 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 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 8 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 ILU(0) 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 strategy improves the triangular solution using our method as well. This characteristic needs to be investigated further in future work.

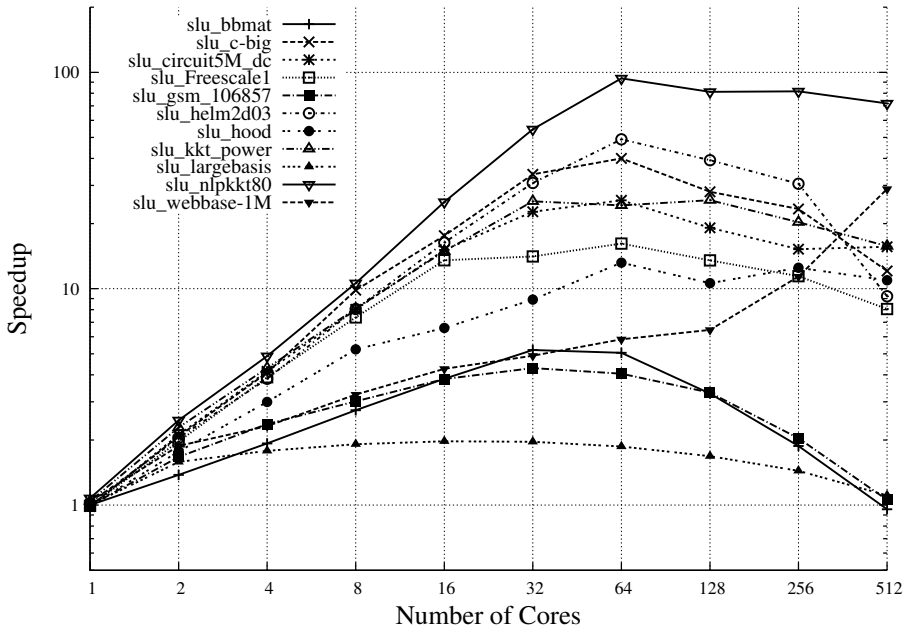


Figure 8: Scaling for complete LU matrices (higher is better).

Scaling for various matrix structures. Performance and scaling of our algorithm can vary with the matrix structure because the effectiveness of each parallelism strategy and optimization highly depends on the matrix structure. Table 1 and Figure 6 help in understanding the parallelism available in various matrices. For example, matrix *nlpkkt120* (Figure 6(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 work in parallel. Those blocks send their values to the right diagonal blocks to complete the computation. Essentially, finding independent rows strategy and parallelization of off-diagonal regions strategies work together in this case. Thus, there are three stages, and each stage has many parallel portions. Matrix *Freescale1* also has similar off-diagonal parallelism opportunities, but with a different structure (Figure 6(c)). For this matrix, since there are not as many independent rows, early send strategy is more important for critical path acceleration. Matrix *circuit5M* (Figure6(d)) shows another structure with good off-diagonal parallelism despite having relatively few independent rows. The top left diagonal blocks enable the computation of many off-diagonal blocks on the left, utilizing the off-diagonal parallelization strategy. Those will be processed in parallel and cause the other diagonals to complete in parallel. From these examples, one can conclude that if the nonzeros in the diagonal blocks have a favorable structure, with many independent rows, finding independent rows strategy helps the most. Otherwise, early send strategy becomes very important to accelerate the critical path. In addition, if there are regions in off-diagonal blocks that are dense enough to amortize the communication overheads, the off-diagonal parallelism strategy can be very effective. Note that often times multiple strategies are effective for a matrix structure simultaneously (e.g., in matrix *nlpkkt120* case explained above).

On the other hand, matrix *Geo_1438* shows poor scaling because it has little parallelism available. Most of its nonzeros are near the diagonal, and the rows are dependent on each other (Figure 6(b)). Most of the matrices with poor scaling have similar structures. 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.

Comparison with HYPRE. Figure 9 compares the performance of our method with that of HYPRE, which is a commonly used linear algebra package [12]. Table 2 also presents some of the solution times. 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.

Table 2: Solution times of triangular solver from HYPRE compared to our structure adaptive triangular solver (SA). Times are in milliseconds.

	1 Core		64 Cores		512 Cores	
Name	HYPRE	SA	HYPRE	SA	HYPRE	SA
largebasis	100	66	220	3.3	30	0.8
slu_Freescale1	390	418	430	25	470	52

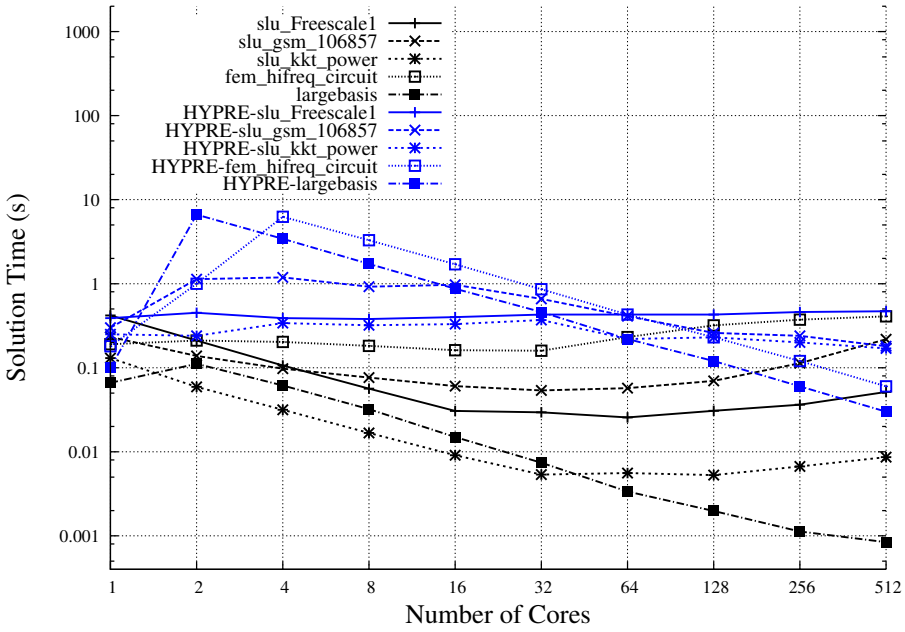


Figure 9: Comparison with triangular solver from HYPRE (lower is better). HYPRE is in blue, and our solver is in black.

Comparison with SuperLU_DIST. Figure 10 compares the performance of our triangular solver to the triangular solver from the SuperLU_DIST package [4]. Table 3 also presents some of the solution times. 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 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 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 subsequent refinement will be much faster, less accurate but faster factorizations may also become possible.

Table 3: Solution times of triangular solver from SuperLU_DIST (SLU_D) compared to our structure adaptive triangular solver (SA). Times are in milliseconds.

	1 Core		64 Cores		512 Cores	
Name	SLU_D	SA	SLU_D	SA	SLU_D	SA
slu_helm2d03	1540	31	570	0.6	240	3.3
slu_largebasis	1260	93	640	50	280	84

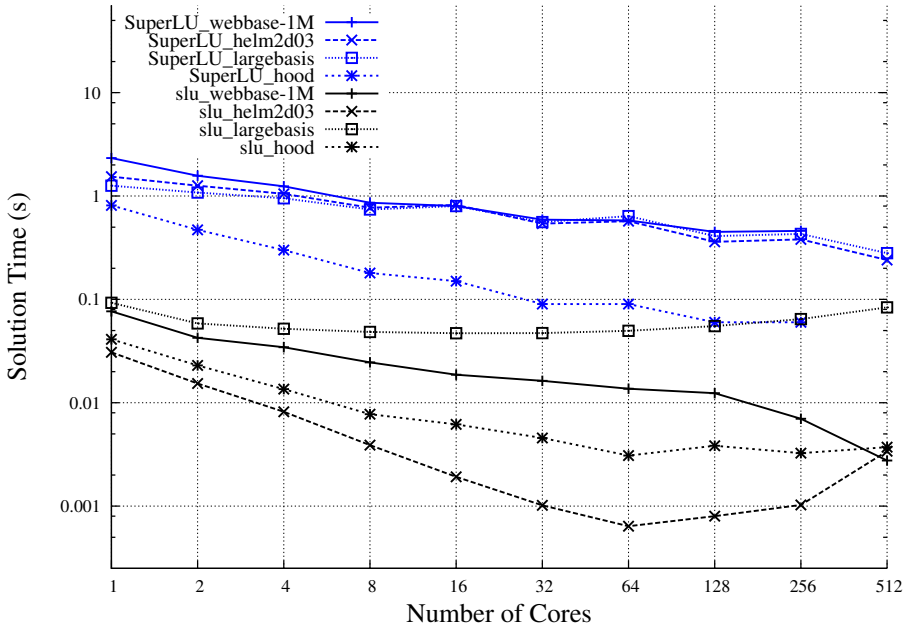


Figure 10: Comparison with triangular solver from SuperLU_DIST (lower is better). SuperLU_DIST is in blue, and our solver is in black.

Comparison with DAG-based 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 [9], so they cannot scale to many cores of a distributed-memory machine. In addition, there can be thousands of barriers for some small matrices [9], so the barriers are a bottleneck even for

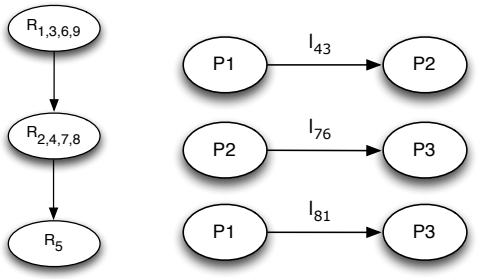
shared-memory machines.

To understand why our algorithm is much more scalable than DAG-based ones (e.g., Level-Set algorithm), we analyze the critical path lengths of the two approaches for a sample of matrices. By critical path length, we mean the longest chain of communication dependencies occurring in each algorithm for a given matrix and number of processors. We choose the longest chain of dependent communication steps as our performance metric in this analysis, because communication overheads (e.g., communication initiation) are very significant in modern large-scale systems. This is especially important for triangular solution since the computation per data element is very small.

The Level-Set algorithm analyzes the rows of the entire matrix, while our algorithm divides the matrix in blocks of columns, then analyzes the local rows and finds the dependencies among processors. Therefore, level-set algorithm follows a dependency DAG of rows, while our algorithm follows a dependency DAG of processors. The critical path length is the diameter of the DAG.

Note that our DAG diameter comparison is not perfect since the algorithms that block the unknowns such as ours often times have a shorter DAG diameter compared to the more fine-grained algorithms. On the other hand, more fine-grained algorithms, such as Level-Set, can potentially expose more parallelism. However, we argue that the DAG diameter factor is dominant in this case and our analysis below explains this bottleneck. Note that we have already established the relatively high degree of parallelism of our algorithm in previous discussions and results.

Figure 11(a) shows the sets of rows (Level-Sets) made by Level-set algorithm and their communication dependencies for example matrix of Figure 1. For comparison, Figure 11(b) shows the communication dependencies of processors for the same matrix in our algorithm (arrow labels indicate which nonzero caused the dependency). As can be seen, the communications can happen concurrently in our case and the diameter of the DAG is shorter. Note also that in some cases, the number of levels (hence, diameter of the DAG) for Level-set algorithm can be extremely high compared to our algorithm. As an extreme example, the bidiagonal matrix of Figure 3 will have critical path length of 8 in Level-set algorithm, while its critical path length will be only 2 in our algorithm.



(a) Level-Set (b) SA

Figure 11: Communication dependencies for example of Figure 1

Table 4 compares the critical path length for our algorithm (when running on 512 processors) to the level-set algorithm. As can be seen, our algorithm's critical path is much shorter in all the cases (and this is consistent for other matrices we examined not shown in the table). In addition, each dependency in our case is between two processors ($O(1)$ communication time), while each dependency implies a full barrier ($O(\log(p))$ communication time) in the level-set algorithm. Note again that the critical path length for each matrix, presented in Figure 7, is highly correlated with the scalability of the corresponding algorithm. Thus, our algorithm is more scalable than DAG-based ones due to a much shorter critical path.

Table 4: Critical path length comparison of our algorithm (SA) with Level-Set algorithm

Matrix	Level-Set	SA
circuit5M	18	2
kkt_power	17	3
Freescale1	216	18
Hamrle3	31083	25
Geo_1438	5823	87

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 whether 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* requires less than 300MB memory. On 512 BGP processors, each solve iteration takes about 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 hundreds or even thousands [9]) of iterations.

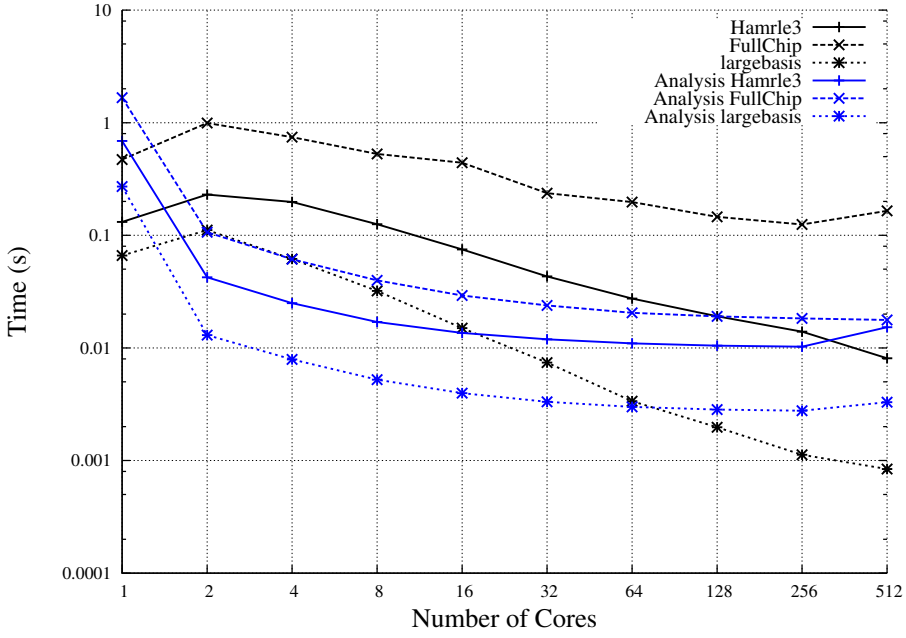


Figure 12: Comparison of analysis and reordering time to solution time (lower is better). Analysis time is in blue, and solution time is in black.

Analysis cost. 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, assuming that some parallel information is available as described in Section 3. 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 12 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 overall solver 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 did not attempt to accelerate it further in this study.

6. Conclusions and Future Work

Solving 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 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, 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 a variety of 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 the characteristics of the matrix and the machine may improve performance. Furthermore, techniques for aggregating messages that minimize communication overheads but do not cause delays for computation should be developed. Moreover, adaptation and evaluation of our algorithm with other decomposition and storage methods need to be investigated. For matrices that do not allow significant parallelism using our algorithm, novel numerical methods that eliminate some of the nonzeros for more parallelism seem to be potentially promising to develop and study.

References

[1] M. Heath, E. Ng, B. Peyton, Parallel algorithms for sparse linear systems, SIAM Review (1991) 420–460.

[2] J. I. Aliaga, M. Bollhfer, A. F. Martn, E. S. Quintana-Ort, Exploiting thread-level parallelism in the iterative solution of sparse linear systems, Parallel Computing 37 (3) (2011) 183 – 202. doi:10.1016/j.parco.2010.11.002.

[3] T. A. Davis, Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse QR factorization, ACM Trans. Math. Softw. 38 (1) (2011) 8:1–8:22. doi:10.1145/2049662.2049670.

[4] X. S. Li, J. W. Demmel, SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Trans. Math. Softw. 29 (2) (2003) 110–140. doi:10.1145/779359.779361.

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

[6] A. Baker, R. Falgout, T. Kolev, U. Yang, Multigrid smoothers for ultra-parallel computing, SIAM Journal on Scientific Computing 33 (5) (2011) 2864–2887.

[7] J. Saltz, Aggregation methods for solving sparse triangular systems on multiprocessors, SIAM J. Sci. Stat. Comput. 11 (1990) 123.

[8] J. Mayer, Parallel algorithms for solving linear systems with sparse triangular matrices, Computing 86 (2009) 291–312.

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

[10] N. J. Higham, A. Pothen, Stability of the partitioned inverse method for parallel solution of sparse triangular systems, SIAM J. Sci. Comput. 15 (1) (1994) 139–148. doi:10.1137/0915009.

[11] P. Raghavan, Efficient parallel sparse triangular solution using selective inversion, Parallel Processing Letters 8 (1) (1998) 29–40.

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

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

[14] T. A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1:1–1:25. doi:10.1145/2049662.2049663.

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

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

[17] 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.

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