A Bit-Compatible Parallelization for ILU(k) Preconditioning

Xin Dong* and Gene Cooperman*

College of Computer Science, Northeastern University
Boston, MA 02115, USA
{xindong,gene}@ccs.neu.edu

Abstract. ILU(k) is a commonly used preconditioner for iterative linear solvers for sparse, non-symmetric systems. It is often preferred for the sake of its stability. We present TPILU(k), the first efficiently parallelized ILU(k) preconditioner that maintains this important stability property. Even better, TPILU(k) preconditioning produces an answer that is bit-compatible with the sequential ILU(k) preconditioning. In terms of performance, the TPILU(k) preconditioning is shown to run faster whenever more cores are made available to it — while continuing to be as stable as sequential ILU(k). This is in contrast to some competing methods that may become unstable if the degree of thread parallelism is raised too far. Where Block Jacobi ILU(k) fails in an application, it can be replaced by TPILU(k) in order to maintain good performance, while also achieving full stability. As a further optimization, TPILU(k) offers an optional *level-based incomplete inverse method* as a fast approximation for the original ILU(k) preconditioned matrix. Although this enhancement is not bit-compatible with classical ILU(k), it is bit-compatible with the output from the single-threaded version of the same algorithm. In experiments on a 16-core computer, the enhanced TPILU(k)-based iterative linear solver performed up to 9 times faster. As we approach an era of many-core computing, the ability to efficiently take advantage of many cores will become ever more important.

1 Introduction

This work introduces a parallel preconditioner, TPILU(k), with good stability and performance across a range of sparse, non-symmetric linear systems. For a large sparse linear system $Ax = b$, parallel iterative solvers based on ILU(k) [1,2] often suffer from instability or performance degradation. In particular, most of today's commonly used algorithms are domain decomposition preconditioners, which become slow or unstable with greater parallelism. This happens as they attempt to approximate a linear system by more and smaller subdomains to provide the parallel work for an increasing number of threads. The restriction to subdomains of ever smaller dimension must either ignore more of the off-diagonal

* This work was partially supported by the National Science Foundation under Grant CCF 09-16133.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 66–77, 2011.
© Springer-Verlag Berlin Heidelberg 2011

matrix elements, or must raise the complexity by including off-diagonals into the computation for an optimal decomposition. The former tends to create instability for large numbers of threads (i.e., for small subdomains), and the latter is slow.

Consider the parallel preconditioner PILU [3,4] as an example. PILU would experience performance degradation unless the matrix A is *well-partitionable* into subdomains. This condition is violated by linear systems generating many fill-ins (as occurs with higher initial density or higher level k) or by linear solvers employing many threads. Another parallel preconditioner BJILU [5] (Block Jacobi ILU(k)), would fail to converge as the number of threads w grows. This is especially true for linear systems that are not diagonally dominant, in which the solver might become invalid by ignoring significant off-diagonal entries. This kind of performance degradation or instability is inconsistent with the widespread acceptance of parallel ILU(k) for varying k to provide efficient preconditioners.

In contrast, TPILU(k) is as stable as sequential ILU(k) and its performance increases with the number of cores. TPILU(k) can capture both properties simultaneously — precisely because it is not based on domain decomposition. In the rest of this paper, we will simply write that *TPILU(k) is stable* as a shortened version of the statement that TPILU(k) is stable for any number of threads whenever sequential ILU(k) is stable.

TPILU(k) uses a task-oriented parallel ILU(k) preconditioner for the base algorithm. However, it optionally first tries a different, level-based incomplete inverse submethod (*TPIILU(k)*). The term *level-based incomplete inverse* is used to distinguish it from previous methods such as "threshold-based" incomplete inverses [6]. The level-based submethod either succeeds or else it fails to converge. If it doesn't converge fast, TPILU(k) quickly reverts to the stable, base task-oriented parallel ILU(k) algorithm.

A central point of novelty of this work concerns bit-compatibility. The base task-oriented parallel component of TPILU(k) is bit-compatible with classical sequential ILU(k), and the level-based optimization produces a new algorithm that is also bit-compatible with the single-threaded version of that same algorithm. Few numerical parallel implementations can guarantee this stringent standard. The order of operations is precisely maintained so that the low order bits due to round-off do not change under parallelization. Further, the output remains bit-compatible as the number of threads increases — thus eliminating worries whether scaling a computation will bring increased round-off error.

In practice, bit-compatible algorithms are well-received in the workplace. A new bit-compatible version of code may be substituted with little discussion. In contrast, new versions of code that result in output with modified low-order bits must be validated by a numerical analyst. New versions of code that claim to produce more accurate output must be validated by a domain expert.

A prerequisite for an efficient implementation in this work was the use of thread-private memory allocation arenas. The implementation derives from [7], where we first noted the issue. The essence of the issue is that any implementation of POSIX-standard "malloc" libraries must be prepared for the case that a second thread frees memory originally allocated by a first thread. This requires

a centralized data structure, which is slow in many-core architectures. Where it is known that memory allocated by a thread will be freed by that same thread, one can use a thread-private (per-thread) memory allocation arena. The issue arises in the memory allocations for "fill-ins" for symbolic factorization. In LU-factorization based algorithms, the issue is still more serious than incomplete LU, since symbolic factorization is a relatively larger part of the overall algorithm.

The rest of this paper is organized as follows. Section 2 reviews LU factorization and sequential ILU(k) algorithm. Section 3 presents task-oriented parallel TPILU(k), including the base algorithm (Sections 3.1 through 3.2) and the level-based incomplete inverse submethod (Section 3.3). Section 4 analyzes the experimental results. We review related work in Section 5.

2 Review of the Sequential ILU(k) Algorithm

A brief sketch is provided. See [8] for a detailed review of ILU(k). LU factorization decomposes a matrix A into the product of a lower triangular matrix L and an upper triangular matrix U. From L and U, one efficiently computes A^{-1} as $U^{-1}L^{-1}$. While computation of L and U requires $O(n^3)$ steps, once done, the computation of the inverse of the triangular matrices proceeds in $O(n^2)$ steps.

For sparse matrices, one contents oneself with solving x in $Ax = b$ for vectors x and b, since A^{-1}, L and U would all be hopelessly dense. Iterative solvers are often used for this purpose. An ILU(k) algorithm finds sparse approximations, $\widetilde{L} \approx L$ and $\widetilde{U} \approx U$. The preconditioned iterative solver then implicitly solves $A\widetilde{U}^{-1}\widetilde{L}^{-1}$, which is close to the identity. For this purpose, triangular solve operations are integrated into each iteration to obtain a solution y such that

$$\widetilde{L}\widetilde{U}y = p \tag{1}$$

where p varies for each iteration. This has faster convergence and better numerical stability. Here, the *level limit* k controls how many elements should be computed in the process of incomplete LU factorization. A level limit of $k = \infty$ yields full LU-factorization.

Similarly to LU factorization, ILU(k) factorization can be implemented by the same procedure as Gaussian elimination. Moreover, it also records the elements of a lower triangular matrix $\widetilde{L}$. Because the diagonal elements of $\widetilde{L}$ are defined to be 1, we do not need to store them. Therefore, a single *filled matrix* F is sufficient to store both $\widetilde{L}$ and $\widetilde{U}$.

2.1 Terminology for ILU(k)

For a huge sparse matrix, a standard dense format would be wasteful. Instead, we just store the position and the value of non-zero elements. Similarly, incomplete LU factorization does not insert all elements that are generated in the process of factorization. Instead, it employs some mechanisms to control how many elements are stored. ILU(k) [1,2] uses the level limit k as the parameter to implement a more flexible mechanism. We next review some definitions.

Definition 2.1. A fill entry*, or* entry *for short, is an element stored in memory. (Elements that are not stored are called zero elements.)*

Definition 2.2. Fill-in*: Consider Figure 1a. If there exists h such that $i, j > h$ and both f_{ih} and f_{hj} are fill entries, then the ILU(k) factorization algorithm may fill in a non-zero value when considering rows i and j. Hence, this element f_{ij} is called a* fill-in*; i.e., an entry candidate. We say the fill-in f_{ij} is* caused *by the existence of the two entries f_{ih} and f_{hj}. The entries f_{ih} and f_{hj} are the* causative entries *of f_{ij}. The causality will be made clearer in the next subsection.*

Definition 2.3. Level*: Each entry f_{ij} is associated with a level, denoted as level (i, j) and defined recursively by*

$$level\ (i,j) = \begin{cases} 0, & if\ a_{ij} \neq 0 \\ \min_{1 \leq h < \min(i,j)} level\ (i,h) + level\ (h,j) + 1, & otherwise \end{cases}$$

The *level limit* k is used to control how many fill-ins should be inserted into the filled matrix during ILU(k) factorization. Those fill-ins with a level smaller than or equal to k are inserted into the filled matrix F. Other fill-ins are ignored. By limiting fill-ins to level k or less, ILU(k) maintains a sparse filled matrix.

2.2 ILU(k) Algorithm and Its Parallelization

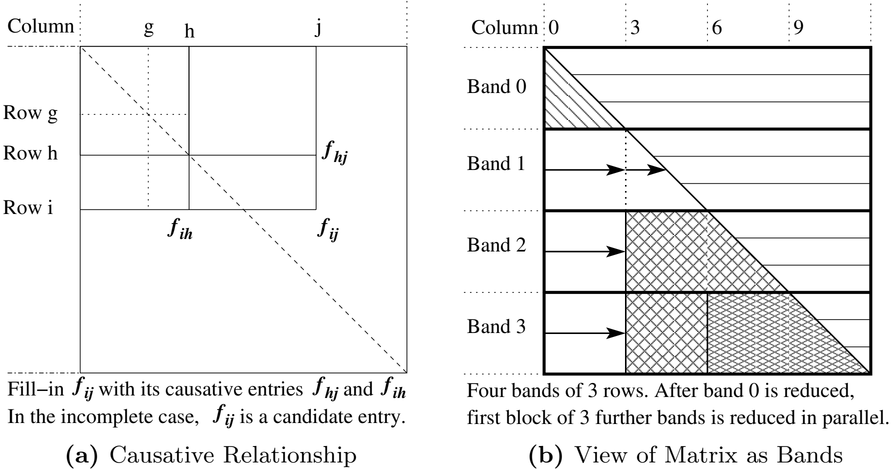
For LU factorization, the defining equation $A = LU$ is expanded into $a_{ij} = \sum_{h=1}^{min(i,j)} l_{ih} u_{hj}$, since $l_{ih} = 0$ for $i > j$ and $u_{hj} = 0$ for $i < j$. When $i > j$, $f_{ij} = l_{ij}$ and we can write $a_{ij} = \sum_{h=1}^{j-1} l_{ih} u_{hj} + f_{ij} u_{jj}$. When $i \leq j$, $f_{ij} = u_{ij}$ and we can write $a_{ij} = \left(\sum_{h=1}^{i-1} l_{ih} u_{hj}\right) + l_{ii} f_{ij} = \left(\sum_{h=1}^{i-1} l_{ih} u_{hj}\right) + f_{ij}$. Rewriting them yields the equations for LU factorization.

$$f_{ij} = \begin{cases} \left(a_{ij} - \sum_{h=1}^{j-1} l_{ih} u_{hj}\right) / u_{jj}, & i > j \\ a_{ij} - \sum_{h=1}^{i-1} l_{ih} u_{hj}, & i \leq j \end{cases} \tag{2}$$

The equations for ILU(k) factorization are similar except that an entry f_{ij} is computed only if $level(i, j) \leq k$. Hence, ILU(k) factorization is separated into two passes: *symbolic factorization* and *numeric factorization.* Symbolic factorization computes the levels of all entries less than or equal to k. Numeric factorization computes the numerical values in the filled matrix of all fill entries with level less than or equal to k. While the remaining description considers numeric factorization, the algorithm applies equally to symbolic factorization.

The ILU(k) algorithm reorganizes the above Equations (2) for efficient use of memory. The filled matrix F is initialized to A. As the algorithm proceeds, additional terms of the form $-l_{ih} u_{hj}$ are added to f_{ij}. Figure 1a illustrates f_{ij} accumulating an incremental value based on the previously computed values of f_{ih} (i.e., l_{ih}) and f_{hj} (i.e., u_{hj}).

The algorithmic flow of control is to factor the rows in order from first to last. In the factorization of row i, h varies from 1 to i in an outer loop, while j varies



(a) Causative Relationship **(b)** View of Matrix as Bands

Fig. 1. Parallel Incomplete LU Factorization

from h to n in an inner loop. In the example of Figure 1a, f_{hj} has clearly already been fully completed. Before the inner loop, f_{ih} is divided by u_{hh} following the case $i > j$ of Equations (2) since $i > h$. This is valid because f_{ih} depends on terms of the form $l_{ig}u_{gh}$ only for the case $g < h$, and those terms have already been accumulated into f_{ih} by previous inner loops. Inside the inner loop, we just subtract $l_{ih}u_{hj}$ from f_{ij} as indicated by Equations (2).

The algorithm has some of the same spirit as Gaussian elimination if one thinks of ILU(k) as using the earlier row h to *reduce* the later row i. This is the crucial insight in the parallel ILU(k) algorithm of this paper. One splits the rows of F into bands, and reduces the rows of a later band by the rows of an earlier band. Distinct threads can reduce distinct bands simultaneously, as illustrated in Figure 1b.

3 TPILU(k): Task-Oriented Parallel ILU(k) Algorithm

3.1 Parallel Tasks and Static Load Balancing

To describe a general parallel model valid for Gaussian elimination as well as ILU(k) and ILUT, we introduce the definition *frontier*: the maximum number of rows that are currently factored completely. The frontier i is the limit up to which the remaining rows can be partially factored except for the $(i+1)^{th}$ row. The $(i+1)^{th}$ row can be factored completely. That changes the frontier to $i+1$.

Threads synchronize on the frontier. To balance and overlap computation and synchronization, the matrix is organized as bands to make the granularity of the computation adjustable, as demonstrated in Figure 1b. A task is associated to a band and is defined as the computation to partially factor the band to the current frontier.

For each band, the program must remember up to what column this band has been partially factored. We call this column the *current position*, which is the start point of factorization for the next task attached to this band. In addition, it is important to use a variable to remember the first band that has not been factored completely. After the first unfinished band is completely factored, the frontier global value is increased by the number of rows in the band.

The smaller the band size, the larger the number of synchronization points. However, TPILU(k) prefers a smaller band size, that leads to more parallel tasks. Moreover, the lower bound of the factorization time is the time to factor the last band, which should not be very large. Luckily, shared memory allows for a smaller band size because the synchronization here is to read/write the frontier, which has a small cost.

While the strategy of bands is well known to be efficient for dense matrices (e.g., see [9]), researchers hesitate to use this strategy for sparse matrices because they may find only a small number of relatively dense bands, while all other bands are close to trivial. The TPILU(k) algorithm works well on sparse matrices because successive factoring of bands produces many somewhat dense bands (with more fill-ins) near the end of the matrix. TPILU(k) uses static load balancing whereby each worker is assigned a fixed group of bands chosen round robin so that each thread will also be responsible for some of the denser bands.

3.2 Optimized Symbolic Factorization

Static Load Balancing and TPMalloc. Simultaneous memory allocation for fill-ins is a performance bottleneck for shared-memory parallel computing. TPILU(k) takes advantage of a thread-private malloc library to solve this issue as discussed in [7]. TPMalloc is a non-standard extension to a standard allocator implementation, which associates a thread-private memory allocation arena to each thread. A thread-local global variable is also provided, so that the modified behavior can be turned on or off on a per-thread basis. By default, threads use thread-private memory allocation arenas. The static load balancing strategy guarantees that if a thread allocates memory, then the same thread will free it, which is consistent with the use of a thread-private allocation arena.

Optimization for the Case $k = 1$. When $k = 1$, it is possible to symbolically factor the bands and the rows within each band in any desired order. This is because if either f_{ih} or f_{hj} is an entry of level 1, the resulting fill-in f_{ij} must be an element of level 2 or level 3. So f_{ij} is not inserted into the filled matrix F. As a first observation, the symbolic factorization now becomes pleasingly parallel since the processing of each band is independent of that of any other.

Second, since the order can be arbitrary, even the purely sequential processing within one band by a single thread can be made more efficient. Processing rows in reverse order from last to first is the most efficient, while the more natural first-to-last order is the least efficient. First-to-last is inefficient, because we add level 1 fill-ins to the sparse representation of earlier rows, and we must then

skip over those earlier level 1 fill-ins in determining level 1 fill-ins of later rows. Processing from last to first avoids this inefficiency.

3.3 Optional Level-Based Incomplete Inverse Method

The goal of this section is to describe the level-based incomplete inverse method for solving $\widetilde{L}x = p$ by matrix-vector multiplication: $x = \widetilde{\widetilde{L}^{-1}}p$. This avoids the sequential bottleneck of using forward substitution on $\widetilde{L}x = p$. We produce incomplete inverses $\widetilde{\widetilde{L}^{-1}}$ and $\widetilde{\widetilde{U}^{-1}}$ so that the triangular solve stage of the linear solver (i.e., solving for y in $\widetilde{L}\widetilde{U}y = p$ as described in Equation (1) of Section 2) can be trivially parallelized ($y = \widetilde{\widetilde{U}^{-1}}\widetilde{\widetilde{L}^{-1}}p$) while also enforcing bit compatibility. Although details are omitted here, the same ideas are then used in a second stage: using the solution x to solve for y in $\widetilde{U}y = x$.

Below, denote the matrix $(-\beta_{it})_{t \le i}$ to be the lower triangular matrix $\widetilde{L}^{-1}$. Recall that $\beta_{ii} = 1$, just as for $\widetilde{L}$. First, we have Equation (3a), i.e., $x = \widetilde{L}^{-1}p$. Second, we have Equation (3b), i.e., the equation for solving $\widetilde{L}x = p$ by forward substitution. Obviously, Equation (3a) and Equation (3b) define the same x.

$$x_i = \sum_{t<i} (-\beta_{it}) p_t + p_i \qquad \text{(3a)} \qquad\qquad x_i = p_i - \sum_{h<i} f_{ih} x_h \qquad \text{(3b)}$$

Substituting Equation (3a) into Equation (3b), one has Equation (4).

$$x_i = p_i - \sum_{h<i} f_{ih} \left(\sum_{t<h} (-\beta_{ht}) p_t + p_h \right) = \sum_{t<i} \left(- \left(f_{it} - \sum_{t<h<i} f_{ih} \beta_{ht} \right) \right) p_t + p_i \tag{4}$$

Combining the right hand sides of equations (3a) and (4) yields Equation (5), the defining equation for β_{it}.

$$\beta_{it} = f_{it} - \sum_{t<h<i} f_{ih} \beta_{ht} \tag{5}$$

Equation (5) is the basis for computing $\widetilde{L}^{-1}$ (a.k.a. $(-\beta_{it})_{t \le i}$). Recall that f_{ij} was initialized to the matrix A. In algorithm steps (6a) and (6b) below, row i is factored using ILU(k) factorization, which computes $\widetilde{L}$ and $\widetilde{U}$ as part of a single matrix. These steps are reminiscent of Gaussian elimination using pivoting element f_{hh}. Steps (6a) and (6b) are used in steps (6c) and (6d) to compute $\widetilde{L}^{-1}$.

$$f_{ih} \leftarrow f_{ih} f_{hh}^{-1} \qquad \text{(6a)} \qquad \forall j > h, f_{ij} \leftarrow f_{ij} - f_{ih} f_{hj} \qquad \text{(6b)}$$

$$\forall t < h, f_{it} \leftarrow f_{it} - f_{ih} f_{ht} \qquad \text{(6c)} \qquad \forall t < i, f_{it} \leftarrow -f_{it} \qquad \text{(6d)}$$

The matrix $\widetilde{L}^{-1}$ is in danger of becoming dense. To maintain the sparsity, we compute the level-based incomplete inverse matrix $\widetilde{\widetilde{L}^{-1}}$ following the same non-zero pattern as $\widetilde{L}^{-1}$. The computation for $\widetilde{\widetilde{L}^{-1}}$ can be combined with the original numeric factorization phase. A further factorization phase is added to

compute $\widetilde{U^{-1}}$ by computing matrix entries in reverse order from last row to first and from right to left within a given row.

Given the above algorithm for $\widetilde{L^{-1}}$ and a similar algorithm for $\widetilde{U^{-1}}$, the triangular solve stage is reduced to matrix-vector multiplication, which can be trivially parallelized. Inner product operations are not parallelized for two reasons: first, even when sequential, they are fast; second, parallelization of inner products would violate bit-compatibility by changing the order of operations.

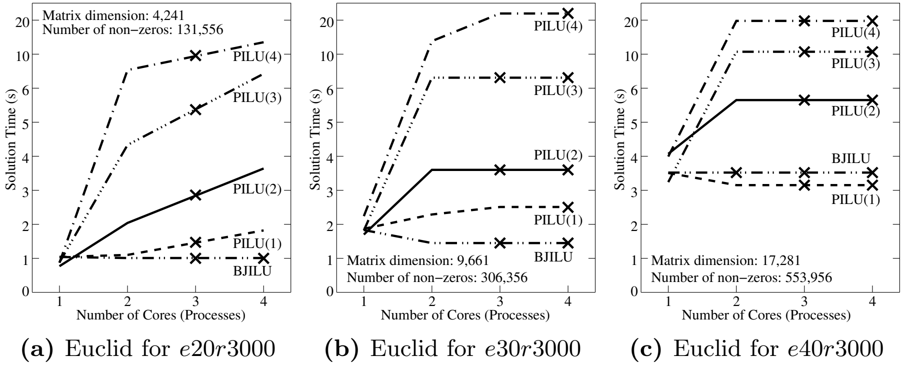
4 Experimental Results

We evaluate the performance of the bit-compatible parallel ILU(k) algorithm, TPILU(k), by comparing with two commonly used parallel preconditioners, PILU [3] and BJILU [5] (Block Jacobi ILU(k)). Both PILU and BJILU are based on *domain decomposition*. Under the framework of Euclid [10, Section 6.12], both preconditioners appear in Hypre [10], a popular linear solver package under development at Lawrence Livermore National Laboratory since 2001.

The primary test platform is a computer with four Intel Xeon E5520 quad-core CPUs (16 cores total). Figure 3 demonstrates the scalability of TPILU(k) both on this primary platform and a cluster including two nodes connected by Infiniband. Each node has a single Quad-Core AMD Opteron 2378 CPU. The operating system is CentOS 5.3 (Linux 2.6.18) and the compiler is gcc-4.1.2 with the "-O2" option. The MPI library is OpenMPI 1.4. Within Hypre, the same choice of iterative solver is used to test both Euclid (PILU and BJILU) and TPILU(k). The chosen iterative solver is preconditioned stabilized bi-conjugate gradients with the default tolerance $rtol = 10^{-8}$. Note that the Euclid framework employs multiple MPI processes communicating via MPI's shared-memory architecture, instead of directly implementing a single multi-threaded process.

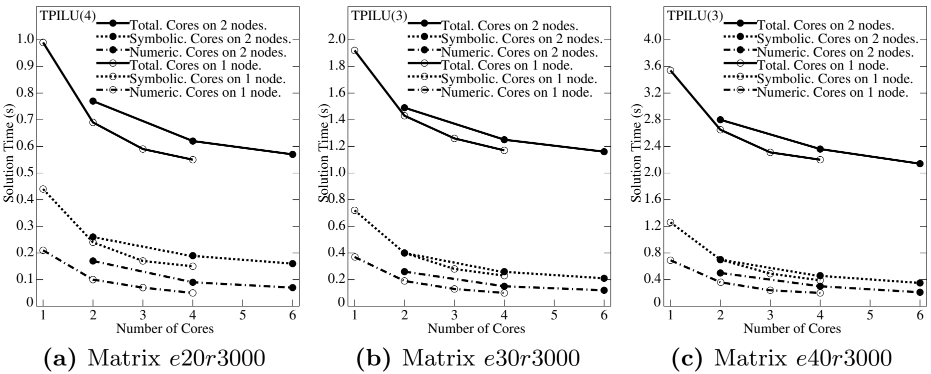
Driven Cavity Problem. This set of test cases [11] consists of some difficult problems from the modeling of the incompressible Navier-Stokes equations. These test cases are considered here for the sake of comparability. They had previously been chosen to demonstrate the features of PILU by [4]. Here, we test on three representatives: $e20r3000$, $e30r3000$ and $e40r3000$. Figure 2 shows that both Euclid PILU and Euclid BJILU are influenced by the number of processes and the level k when solving driven cavity problems. With more processes or larger k, both the PILU and BJILU preconditioners tend to slow down, break down or diverge.

Euclid registers its best solution time for $e20r3000$ by using PILU(2) with 1 process, for $e30r3000$ by using BJILU with 2 processes, and for $e40r3000$ by using PILU(1) with 2 processes. The reason that Euclid PILU obtains only a small speedup for these problems is that PILU requires the matrix to be *well-partitionable*, which is violated when using a larger level k or when employing more processes. Similarly, Euclid BJILU must approximate the original matrix by a number of subdomains equal to the number of processes. Therefore, higher parallelism forces BJILU to ignore even more off-diagonal matrix entries with



(a) Euclid for $e20r3000$ **(b)** Euclid for $e30r3000$ **(c)** Euclid for $e40r3000$

Fig. 2. Euclid PILU and BJILU for Driven Cavity Problem using a Single AMD Opteron (4 Cores). "X" means fail, and the time is arbitrarily shown to be an interpolated value or the same as for the preceding number of threads. Note that in Figure 2(a), PILU(k) actually breaks down for 3 threads, while then succeeding for 4 threads.



(a) Matrix $e20r3000$ **(b)** Matrix $e30r3000$ **(c)** Matrix $e40r3000$

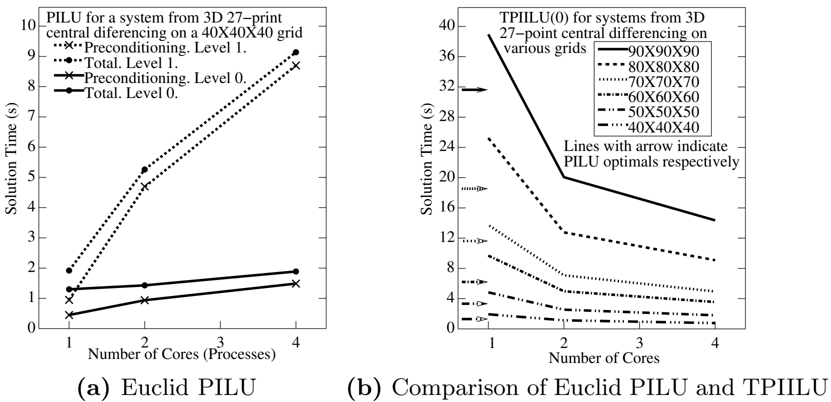
Fig. 3. TPILU(k) for the Driven Cavity Problem Using 2 AMD Opteron (2×4 Cores). The experimental runs for 1,2,3,4 threads are all for a 4-core shared memory CPU. The experimental runs for 2,4,6 threads are all for two nodes with 4-cores per node, while an additional thread per node is reserved for communication between nodes in order to replicate bands.

more blocks of smaller block dimension, and eventually the BJILU computation just breaks down.

In contrast, TPILU(k) is bit-compatible. Greater parallelization only accelerates the computation, while also never introducing instabilities or other negative side effects. Figure 3a illustrates that for the $e20r3000$ case, TPILU with level $k = 4$ and 4 threads leads to a better performance (0.55 s) than Euclid's 0.78 s (Figure 2a). For the $e30r3000$ case, TPILU(k) finishes in 1.16 s (Figure 3b), as compared to 1.47 s for BJILU and 1.64 s for PILU (Figure 2b). For the $e40r3000$ case, TPILU(k) with $k = 3$ finishes in 2.14 s (Figure 3c), as compared to 3.15 s

for PILU and 3.52 s for BJILU (Figure 2c). Figure 3c demonstrates the potential of TPILU(k) for further performance improvements when a hybrid architecture is used to provide additional cores: the hybrid architecture with 6 CPU cores over two nodes connected by Infiniband is even better (2.14 s) than the shared-memory model with a single quad-core CPU (2.20 s).

3D 27-point Central Differencing. As pointed out in [4], ILU(k) preconditioning is amenable to performance analysis since the non-zero patterns of the resulting ILU(k) preconditioned matrices are identical for any partial differential equation (PDE) that has been discretized on a grid with a given stencil. However, a parallelization based on domain decomposition may eradicate this feature since it generally relies on re-ordering to maximize the independence among subdomains. The re-ordering is required for domain decomposition since it would otherwise face a higher cost dominated by the resulting denser matrix. As Figure 4a shows, Euclid PILU degrades with more processes when solving a linear system generated by 3D 27-point central differencing for Poisson's equation. The performance degradation also increases rapidly as the level k grows.



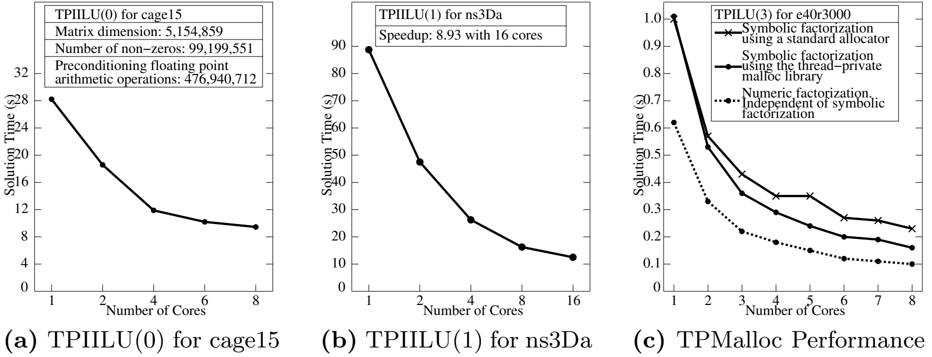
(a) Euclid PILU **(b)** Comparison of Euclid PILU and TPIILU

Fig. 4. Solving Linear System from 3D 27-point Central Differencing on Grid using a Single AMD Quad-Core Opteron. Focusing on the algorithm only, the comparison ignores reusing the domain decomposition over multiple linear system solutions.

This performance degradation is not an accident. The domain-decomposition computation dominates when the number of non-zeros per row is larger (about 27 in this case). Therefore, the sequential algorithm with the level $k = 0$ wins over the parallelized PILU in the contest for the best solution time. This observation holds true for all grid sizes tested: from $50 \times 50 \times 50$ to $90 \times 90 \times 90$. In contrast, for all of these test cases, TPIILU (the level-based incomplete inverse submethod of TPILU(k)) leads to improved performance using 4 cores, as seen in Figure 4b.

Model for DNA Electrophoresis: cage15. The cage model of DNA electrophoresis [12] describes the drift, induced by a constant electric field, of homogeneously charged polymers through a gel. We test on the largest case in this

problem set: *cage*15. For *cage*15, TPIILU(0) obtains a speedup of 2.93 using 8 threads (Figure 5a). The ratio of the number of FLoating point arithmetic OPerations (FLOPs) to the number of non-zero entries is less than 5. This implies that ILU(k) preconditioning just passes through matrices with few FLOPs. In other words, the computation is too "easy" to be further sped up.



(a) TPIILU(0) for cage15 **(b)** TPIILU(1) for ns3Da **(c)** TPMalloc Performance

Fig. 5. TPIILU(k)/TPILU(k) using 4 Intel Xeon E5520 (4×4 Cores)

Computational Fluid Dynamics Problem: ns3Da. The problem *ns*3*Da* [12] is used as a test case in FEMLAB, developed by Comsol, Inc. Because there are zero diagonal elements in the matrix, we use TPIILU with level $k = 1$ as the preconditioner. Figure 5b shows a speedup of 8.93 with 16 threads since the preconditioning is floating-point intensive.

TPMalloc Performance. For a large level k, the symbolic factorization time will dominate. To squeeze greater performance from this first phase, glibc's standard malloc is replaced with a thread-private malloc (TPMalloc). Figure 5c demonstrates that the improvement provided by TPMalloc is significant whenever the number of cores is greater than 2.

4.1 Experimental Analysis

Given a denser matrix, or a higher level k or more CPU cores, the time for domain-decomposition based parallel preconditioning using Euclid's PILU(k) can dominate over the time for the iterative solving phase. This degrades the overall performance, as seen both in Figure 4a and in Figures 2(a,b,c). A second domain-decomposition based parallel preconditioner, Euclid's BJILU, generally produces a preconditioned matrix of lower quality than ILU(k) in Figure 2(a,b,c). This happens because it ignores off-diagonal non-zero elements. Therefore, where Euclid PILU(k) degrades the performance, it is not reasonable to resort to Euclid BJILU. Figures 2a and 2c show that the lower quality of BJILU-based solvers often performed worse than PILU(k). Figure 3 shows TPILU(k) to perform better than either while maintaining the good scalability expected of a bit-compatible algorithm. TPILU(k) is also robust enough to perform reasonably even in a

configuration with two quad-core nodes. Additionally, Figures 4b and 5 demonstrate very good scalability on a variety of applications when using the optional level-based incomplete inverse optimization.

5 Related Work

ILU(k) [1] was formalized to solve the system of linear equations arising from finite difference discretizations in 1978. In 1981, ILU(k) was extended to apply to more general problems [2]. Some previous parallel ILU(k) preconditioners include [3,13,14]. The latter two methods, whose parallelism comes from level/backward scheduling, are stable and were studied in the 1980's and achieved a speedup of about 4 or 5 on an Alliant FX-8 [5, 1st edition, page 351] and a speedup of 2 or 3 on a Cray Y-MP. The more recent work [3] is directly compared with in the current work, and is not stable.

References

1. Gustafsson, I.: A Class of First Order Factorization Methods. BIT Numerical Mathematics, Springer Netherlands 18(2), 142–156 (1978)
2. Watts III, J.W.: A Conjugate Gradient-Truncated Direct Method for the Iterative Solution of the Reservoir Simulation Pressure Equation. SPE Journal 21(3), 345–353 (1981)
3. Hysom, D., Pothen, A.: A Scalable Parallel Algorithm for Incomplete Factor Preconditioning. SIAM J. Sci. Comput 22, 2194–2215 (2000)
4. Hysom, D., Pothen, A.: Efficient Parallel Computation of ILU(k) Preconditioners. In: Supercomputing 1999 (1999)
5. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM, Philadelphia (2003)
6. Bollhöfer, M., Saad, Y.: On the Relations between ILUs and Factored Approximate Inverses. SIAM J. Matrix Anal. Appl. 24(1), 219–237 (2002)
7. Dong, X., Cooperman, G., Apostolakis, J.: Multithreaded Geant4: Semi-Automatic Transformation into Scalable Thread-Parallel Software. In: Euro-Par 2010 (2010)
8. Saad, Y., van der Vorst, H.A.: Iterative Solution of Linear Systems in the 20th Century. J. Comput. Appl. Math. 123(1-2), 1–33 (2000)
9. Cooperman, G.: Practical Task-Oriented Parallelism for Gaussian Elimination in Distributed Memory. Linear Algebra and Its Applications 275-276, 107–120 (1998)
10. hypre: High Performance Preconditioners. User's Manual, version 2.6.0b, `https://computation.llnl.gov/casc/hypre/download/hypre-2.6.0b_usr_manual.pdf`
11. Matrix Market.: Driven Cavity from the SPARSKIT Collection, `http://math.nist.gov/MatrixMarket/data/SPARSKIT/drivcav/drivcav.html`
12. UF Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices/`
13. Anderson, E.: Parallel Implementation of Preconditioned Conjugate Gradient Methods for Solving Sparse Systems of Linear Equations. Master's Thesis, Center for Supercomputing Research and Development, University of Illinois (1988)
14. Heroux, M.A., Vu, P., Yang, C.: A Parallel Preconditioned Conjugate Gradient Package for Solving Sparse Linear Systems on a Cray Y-MP. Appl. Num. Math. 8, 93–115 (1991)