# A Scalable Task Parallelism Approach For LU Decomposition With Multicore CPUs

Verinder S. Rana
Stony Brook University
Brookhaven National Laboratory

Meifeng Lin
Brookhaven National Laboratory

Barbara Chapman
Stony Brook University
Brookhaven National Laboratory

***Abstract*—Many scientific applications have linear systems $A \cdot x = b$ which need to be solved for different vectors $b$. LU decomposition, which is a variant of Gaussian Elimination, is an efficient technique to solve a linear system. The main idea of the LU decomposition is to factorize A into an upper (U) triangular and a lower (L) triangular matrix such that $A = LU$. This paper presents an OpenMP task parallel approach for the LU factorization of dense matrices. The tasking model is based on the individual computational tasks which occur during the block-wise LU factorization. We describe the right-looking variant of the LU decomposition algorithm in the task parallel approach, and provide an efficient implementation of the algorithm for shared memory machines. We demonstrate that with the task scheduling features provided by OpenMP 4.0, the right-looking LU decomposition can scale well. We then conduct an experimental evaluation of the task parallel implementation in comparison with the parallel-for implementation of the Gaussian elimination with pivoting and LU decomposition using the GNU Scientific Library on a multicore platform. From the experiments we conclude that the proposed task-based implementation is a good solution for solving large systems of linear equations using LU decomposition.**

***Index Terms*—High performance computing, multithreading, parallel algorithms.**

## I. Introduction

Finding the solutions of linear systems is an important methodology that is used by many scientific disciplines ranging from density functional theory [11], quantum transport [14], dynamical mean field theory [16], and uncertainty quantification [2], for example. There are many methods for solving a linear system with direct or iterative methods. In this paper, we study the task-based parallelization based on a direct method for solving a linear system. We focus on the block LU decomposition method as a kernel which can be used to solve a plethora of scientific problems.

Task parallelism exploits the uncoupled nature of individual computational tasks and executes these tasks in parallel. As it is shown in Fig. 1, tasks are generated by a single master thread and executed by different threads in parallel. This type of parallelism is referred to as unstructured parallelism, since the tasks are uncorrelated amongst them and the temporal order of execution is not important. It exposes us to multiple program multiple data paradigm with each task containing both work and data. As a result, care needs to be taken with respect to its use and implementation to avoid problems with memory locality. For many scientific computations it is more natural to express parallelism in terms of independent loop iterations. Tasks were introduced to OpenMP in version 3.0, but there are relatively few reports of their use, and few existing benchmarks are from technical codes.

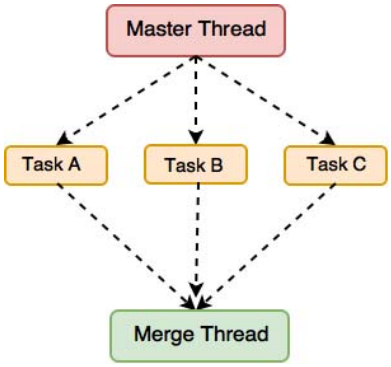


Fig. 1. Master thread creates a team of parallel worker threads; the tasks are executed in parallel by the worker threads; at the end of the parallel region, the tasks are merged.

Efficient usage of task-based parallelism can reduce the idle time and aid in balancing the cores due to the use of dynamic scheduling of the computational work. Dynamic scheduling assigns work to the cores on the basis of the availability of data for computation. This scheduling technique results in a directed acyclic graph (DAG) and allows the cores to explore the graph in many ways without blocking execution or letting it idle.

A literature survey shows that many researchers have studied different parallelization schemes of LU decomposition method on a variety of high performance computer systems and multicore systems (i.e. dual/quad cores) using standard interfaces such as MPI, or parallel extension of language C/C++ such as OpenMP [6], [19]. Tan [17] shows the block LU decomposition adaptive to a domain specific embedded language. Kim et al. [13] presents a block Incomplete Cholesky factorization that uses task-based parallelism. Booth et al. [3] have presented a threaded version of a sparse LU decomposition algorithm that takes advantage of both data and task-based parallelism.

Further, experimental results of the LU decomposition on

978-1-5090-3858-9/16 $31.00 © 2016 IEEE
DOI 10.1109/ESPM2.2016.6

are presented in [1] and [5] respectively. There also exist implementations on multicore system for factorization algorithms for solving systems of equations (LU, QR and Cholesky) [4], [9], [15], respectively.

Some efforts of using OpenMP tasks in the implementation of LU decomposition exist. In [10], the authors showed the performance of a block LU decomposition in the context of extending the OpenUH compiler runtime to support flexible task synchronization with the presence of task dependencies. To test the efficiency of the OpenMP implementation of tasks, a benchmark suite KASTORS is introduced [18]. KASTORS allows the comparison of algorithms that are implemented using task dependence versus `taskwait`.

In this work, we develop a task-based C++ implementation of the right-looking LU algorithm for shared-memory machines. The parallelization makes use of the task scheduling features provided by OpenMP 4.0. We demonstrate the performance of our implementation on a number of test matrices and compare it to two reference implementations: i) LU decomposition in the GNU Scientific Library (GSL) and ii) LU decomposition using the OpenMP `parallel for` clause.

## II. LU Factorization

Although there are many different schemes to factorize matrices, LU decomposition is one of the more commonly-used algorithms. Firstly, we describe the known LU method for solving a system of linear equations. Consider the following real linear algebraic system $A \cdot x = b$, where $A$ is a non-singular matrix, $b$ is the right-hand side and $x$ is the vector of the unknowns. $A$ may be decomposed into a lower triangular part $L$ and an upper triangular part $U$ that will lead us to a direct procedure for the solution of the original system. This decomposition procedure is especially useful when more than one right-hand side (more than one $b$) is to be used. The LU decomposition method uses the same number of multiply and add operations as the Gauss-Jordan method. However the LU decomposition method is much easier to parallelize because it can be implemented using a recursive block method.

The algorithm is relatively straightforward. First, we determine the upper and lower triangular parts:

$$A = L \cdot U. \tag{1}$$

Then

$$A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = L \cdot y \tag{2}$$

where $y = U \cdot x$. Then $L \cdot y = b$ can be solved with forward substitution and $U \cdot x = y$ can be solved using backward substitution. Algorithm 1 is the well known Crout method which makes LU factorization a byproduct of Gaussian elimination. Factoring A into LU requires approximately $\frac{2}{3}n^3$ floating point operations (FLOPS) and doing the forward and backward solves requires $n^2$ FLOPS. Thus the algorithm involves a total of $\frac{2}{3}n^3 + n^2$ FLOPS. This algorithm is the basis for the naive OpenMP `parallel for` implementation given in Algorithm 2.

**Algorithm 1** Serial Crout LU

```
for k < n do
    for i = k + 1; i < n do
        A_{i,k} = A_{i,k} / A_{k,k}
    end for
    for j = k; j < n do
        for i = k + 1; i < n do
            A_{i,j} = A_{i,j} − A_{i,k} · A_{k,j}
        end for
    end for
end for
```

**Algorithm 2** Parallel Crout LU

```
for k < n do
    for i = k + 1; i < n do
        A_{i,k} = A_{i,k} / A_{k,k}
    end for
  #pragma omp parallel for
    for j = k; j < n do
        for i = k + 1; i < n do
            A_{i,j} = A_{i,j} − A_{i,k} * A_{k,j}
        end for
    end for
end for
```

### A. Block LU algorithm

As mentioned before, the LU decomposition equates the product of $L \cdot U$ to $A$. There are three variants of the block LU algorithm: the left-looking, right-looking and the block based Crout algorithm. Two of the most widely used implementations are the left-looking and right-looking factorization algorithms. Consider the following illustration of a $3 \times 3$ block matrix,

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{bmatrix} \tag{3}$$

Using the left-looking algorithm, we already know the values for $L_{11}, L_{21}$ and $L_{31}$ and we would like to solve for the next block column of some width $NB$ in $L$ and $U$. Equating the second column of $L$ and $U$ with that of $A$ we obtain the following system of equations,

$$A_{12} = L_{11} \cdot U_{12}, \tag{4}$$

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \cdot U_{12} + \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} \cdot U_{22}$$

Algorithm 3 computes an LU factorization using a partitioned outer product implementation.

**Algorithm 3** Block LU

Factor $A_{11} = L_{11} \cdot U_{11}$
Solve $L_{11} \cdot U_{12} = A_{12}$ for $U_{12}$
Solve $L_{21} \cdot U_{11} = A_{21}$ for $L_{21}$
Form $S = A_{22} - L_{21} \cdot U_{12}$
Repeat 1-4 on $S$

If we substitute $L_{21}$ and $U_{12}$ from step 3 into step 4, we obtain the Schur complement of $A_{11}$. In the right-looking variant, partial pivoting is equivalent to an update operation. This is a particular feature of the right-looking algorithm due to the fact that the elements of the current column are swapped and then the swaps are also performed on columns to the right of the current column. There are a lot of differences between the left and the right-looking algorithms, each having their own strengths and weaknesses. The key difference is the way the data are accessed from the factorized part of matrix and applied to the part of matrix that is not factorized. Therefore the left-looking and right-looking algorithms exhibit different memory access and data communication patterns. As a result, their performance can be quite different on shared-memory and distributed-memory parallel machines.

The left-looking algorithm could be regarded better in terms of performance for data access than the right-looking variant. Referring to equations 3 and 4, we observe that the data access only occurs to the left of the block column which is being updated. However, the storage layout of the matrix in memory and how the data is distributed may lead to different choices for one or the other. Our block based implementation is that of the right-looking kernel.

The Crout block algorithm is a hybrid approach in which a block row and column is computed at each step using the previously computed rows and columns. When solving for the $U_{12}$ block matrix, we need to use the lower triangular matrix $L_{11}$. As a result, there is a matrix-matrix multiplication that is then used to compute the term involving $U_{12}$ in the second equation, which is then subtracted from the left hand side. An unblocked variant of the LU decomposition is then applied to the rectangular column of width $NB$ to compute the remaining block matrices $L_{22}$, $L_{32}$ and $U_{22}$, along with the pivot entries.

## III. Task-based Implementation

In this section, we present an OpenMP task-based implementation of the block LU method using the tasking clause structure. The right-looking LU decomposition has complex data dependencies. Naive parallelization strategies based on multicore BLAS or `parallel for` loops cannot achieve efficient scalability on multicore shared memory clusters [12]. An efficient scalable implementation of the right-looking LU decomposition requires a meticulous code structure of the computation in terms of individual tasks to carry out the computational work load.

In addition, the independent tasks also need to properly resolve any task dependencies and granularity. We demonstrate how the right-looking block algorithm can be parallelized on shared memory machines using OpenMP. The computational tasks and task dependencies are easily described, thanks to the tasking feature in OpenMP 4.0. We do a two way comparison with respect to Algorithm 4. One of the modes of comparisons is using the LU decomposition from GSL. GSL offers a variety of computing routines. The algorithm for the LU decomposition in GSL is based on Gaussian Elimination with partial pivoting.

### A. Implementation

When using multicore CPUs, the aforementioned LU factorization algorithm imposes a severe bottleneck, due to the fact that all updates need to wait for the current panel to have been computed. In addition, parallelization methods within the panel computation are very limited, which lead to inefficient performance on multicore systems. To resolve this issue, we further split the panel into square sub-matrices or otherwise known as tiles. When the tiled computations in the panel are computed, we can then begin the update operations for blocks in the trailing sub-matrix, which takes away the synchronization problem. For the dense LU factorization, the tile computation or the update of a block via matrix multiplication, creates a single task with data dependencies.

**Algorithm 4** Task Based Block LU Decomposition

```
   #pragma omp parallel
2: #pragma omp master
   for step ∈ numberOfBlocks do
4:     LUPivot()
       #pragma omp taskwait
6:     LUPermutations()
       #pragma omp taskgroup {
8:     for j ∈ steps do
           #pragma omp task depend(out: shadedMatrix)
10:        LUSolve
       end for
12:    for k ∈ steps do
           for l ∈ steps do
14:            #pragma omp task depend(in: shadedMatrix)
               LUmatrixMult
16:        end for
       end for}
18: end for
```

The latest feature of OpenMP 4.0 augments tasks with tasking clauses and task dependencies by using the `depend` clause. Our task-based algorithm for the block LU, as shown in Algorithm **??**, exploits the new features. In the simplest sense, the `depend` keyword employs a list of input and output dependencies for each task, which is essentially a list of variables or memory addresses. This makes way for the given task to read its input data or write its output data. The `depend` keyword basically specifies the access mode for each shared variable of a certain task.

The access modes correspond to `in` (read data), `out` (write data) and `inout`(read and write data). Before beginning the execution of a particular task, it is required by the OpenMP

tasking queue that all previously submitted dependencies be completed. Enforcing this rule allows the application to dynamically set implicit barriers to related tasks without taking a performance hit for the execution of the rest of the tasks.

Task dependencies have several benefits associated with them. The primary advantage is that task dependencies have decentralized synchronization operations that scale better than the `taskwait` approach. Using the `taskwait` approach, it can slow down the performance of the application because the `taskwait` keyword waits for all of the tasks to be completed in the particular section. In essence, the `taskwait` keyword is more conservative syncing than that of just using the task dependency clause. Another benefit of using the task dependencies is that of a potential optimization during runtime, which includes better data storage for NUMA systems.

Algorithm 4 shows the task and task dependencies in steps 9-14. For example, the task that has been augmented with `depend (out: shadedMat)` means that once that task has executed the LU solve block operation, any task that is decorated with the dependency clause `depend (in: shadedMat)` can be executed. In the case when multiple dependency clauses are present, the task can only be executed after all tasks in the dependency list are complete. The `taskwait` acts similar to a barrier but for tasks. The `taskwait` keyword ensures that the current execution will pause until all tasks have been executed. It can be thought of as a scheduling point, where threads process tasks. The `master` construct is needed so that tasks will be created by a single thread only. If we do not employ the `master` construct, each task would get created `omp_num_threads` times, which would be incorrect. If we do not explicitly implement the task scheduling points that are present inside the code region, there is a chance that the OpenMP runtime might start the execution of the tasks at its own discretion.

## IV. Computational Results

### A. System Platform and Computational Process

For our experimental evaluation we used a dual-socket Intel Xeon CPU E5-2680 v2 (20 total cores) with 2.80 GHz clock speed and 64 GB of memory. The system ran GNU/Linux, kernel version 2.6, for the x86 64 ISA. All programs were implemented in C++ using OpenMP and were compiled using GCC, version 6.1.0, with the $-O3$ optimization flag. Several sets of randomly generated matrices in double precision with sizes ranging from 1024 x 1024 to 16384 x 16384 were used to evaluate the performance of the parallel algorithms . To compare the parallel algorithms, the practical execution time (in seconds) was used as a measure. Practical execution time is the total time in seconds an algorithm needs to complete the computation, and it was measured using the function of the C++ Chronos Library.

### B. Analysis Based on Computational Results

In the serial case, we notice in Tables I and II that the GSL library is slower in all aspects than that of the task-based and for-loop based implementations for large matrix systems. However, for small to medium sized systems, Table I shows that the execution time is similar between all three implementations. The performances given in Table II, in terms of giga floating point operations per second (GFLOP/s), between the three implementations vary. That is due to the fact that three implementations are in fact three different algorithms. If the algorithms were identical then the FLOP/s would have been similar. For the largest matrix tested, 16384 x 16384, we notice that with a single thread, the total execution time taken by the task-based approach is 10% less than the OMP-For implementation, and 34% less than the GSL implementation. In terms of FLOP/s, the GSL implementation is approximately 30% less than that of the block LU approach used in the task-based implementation. The conclusion drawn from the experiments on the single thread is that the block based matrix LU decomposition is more effective on larger systems than those that do not take advantage of such properties.

The experiments that are run on multiple threads paint a different picture. What we notice is that for smaller to mid-sized systems, 1024 x 1024 to 1936 x 1936, the OMP-For based implementation tends to get better performance in terms of FLOP/s as depicted in Figures 3 and 5. The better performance is seen as the number of threads is increased compared to the OMP-Task based algorithm. We can also see OMP-For's better performance in terms of the execution time in Figures 2 and 4.

On the contrary, for the medium to large-sized systems, the task-based implementation outperforms the OMP-For implementation. With 16 threads the execution time for the task-based approach with a 4096 x 4096 matrix is approximately half of the OMP-For implementation, as shown in Figures 6, while for the large matrix of size 16384 x 16384, the task-based implementation is four times faster than that of the OMP-For counterpart, shown in Fig. 8. The scalability of the task based algorithm for the medium sized 4096 x 4096 system has more desirable results. Figure 7 shows that with 16 threads, the task-based approach is three times more scalable than the naive OMP-For implementation. Figure 9 shows that we achieve seven times more scalability than the `parallel for` loop implementation with the 16384 x 16384 matrix.

We notice that the `parallel for` loop implementation loses its linearity in scaling as shown in Figure 9. We hypothesize the possibilities as to why this may be happening. The data initialization is being done in such a way that the data is not being distributed evenly among the NUMA domains. Since we are loading the matrix in a serial manner, all of the memory that is being allocated is residing on a single NUMA node. If this is the case, the naive implementation will not benefit from the full memory bandwidth available on our system. If we enable memory interweaving within NUMA policy so that we can have the memory allocation spread across both NUMA nodes, the scaling behavior may improve. The option of enabling the memory interweaving would allow each thread with 50% local memory access and 50% remote memory access instead of having all threads on a single CPU being hit by 100% remote memory access. Our

TABLE I
COMPARISON OF OMP-TASK, OMP-FOR AND GSL LU DECOMPOSITION ON A SINGLE THREAD. VALUES REPRESENT TOTAL TIME OF EXECUTION FOR EACH ALGORITHM IN SECONDS. AS WE CAN SEE, THAT THE BLOCK DECOMPOSITION IS EFFECTIVE FOR LARGER NUMBER OF ELEMENTS.

| | OMP-Task | OMP-For | GSL |
|---|---|---|---|
| 1024 x 1024 | 0.2688 | 0.2635 | 0.3078 |
| 1936 x 1936 | 2.03 | 1.75 | 2.18 |
| 4096 x 4096 | 18.07 | 17.97 | 25.68 |
| 16384 x 16384 | 1056.85 | 1147.34 | 1596.62 |

TABLE II
COMPARISON OF OMP-TASK, OMP-FOR AND GSL LU DECOMPOSITION ON A SINGLE THREAD. VALUES REPRESENT TOTAL NUMBER OF GIGA-FLOATING POINT OPERATIONS PER SECOND FOR EACH ALGORITHM. AS WE CAN SEE THAT THE BLOCK DECOMPOSITION IS EFFECTIVE FOR LARGER NUMBER OF ELEMENTS.

| | OMP-Task | OMP-For | GSL |
|---|---|---|---|
| 1024 x 1024 | 2.643 | 2.68 | 2.327 |
| 1936 x 1936 | 2.394 | 2.74 | 2.178 |
| 4096 x 4096 | 2.541 | 2.445 | 1.693 |
| 16384 x 16384 | 2.788 | 2.568 | 1.845 |

second hypothesis is that the tapering off of the performance of the naive implementation may be due to the use of the static scheduler with a large number of threads. There is an inevitable unbalance between the slowest and fastest threads, which could introduce large variations during the runtime when the iterations are divided in large chunks. However, we have not tested these hypotheses yet, and further investigations are needed to understand the loss of scalability beyond 4 threads in the OMP-For implementation.

For the case of poor performance of the task-based implementation with small matrices, we attribute it to the cost of the task generation. There have been methods proposed to limit the overheads in task generation when some degree of cutoff threshold is signaled [8]. One of the proposed cutoff method, max-level, is based on the depth level of the recursion for divide-and-conquer based programs. Another is based on the number of tasks in the system, specified as some factor $k$ times the number of parallel execution threads. The study in [8] claims that the culprit for poor tasking performance is when no cutoff is specified. There are different cutoff strategies that are best suited for different applications and it is worthwhile to explore this avenue as they may decrease task overhead and increase application performance. For example, Adaptive Task Cutoff (ATC) is a strategy which dynamically selects the cutoff at runtime based on profiling data that is sampled early in the program's execution [7]. These strategies are not investigated in this study. However, when we reach larger sized systems, the task based approach clearly outperforms the other two implementations, diminishing the issue of the task generation overhead for large matrix systems.

## V. CONCLUSIONS

On multicore systems a formulation of the block-wise LU factorization based on individual tasks and the dependencies between them is introduced. The design of such an algorithm

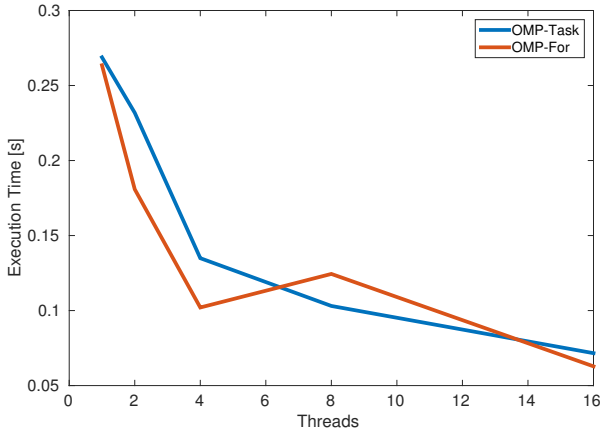


Fig. 2. Comparison of the execution time of the LU decomposition of a 1024 x 1024 matrix with different numbers of threads for the task-based (OMP-Task) and omp `parallel for` (OMP-For) implementations.

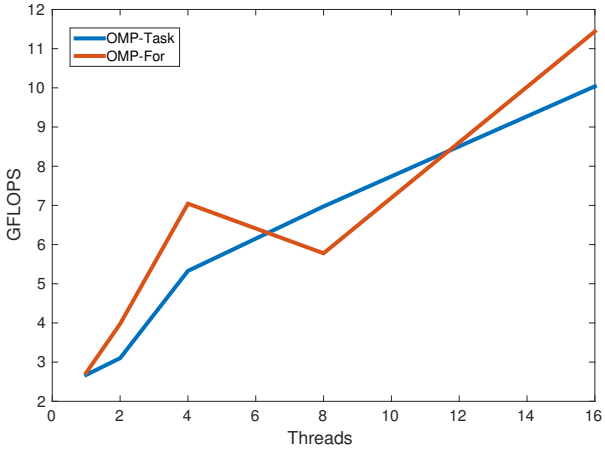


Fig. 3. Scalability of the OMP-Task and OMP-For implementations for a 1024 x 1024 matrix.

leads to an almost optimal scaling behavior for different matrices. We showed via numerical experiments that the modified LU factorization algorithm yields a higher parallel speedup compared to a naive parallel-for implementation and the GSL LU decomposition. Furthermore, the principal technique of task-based parallelism is applicable to various other matrix algorithms. This forms a foundation for matrix implementations on future architectures with multicore or many-core processors. The study needs to be extended to study the modified block task-based algorithm of the LU factorization on distributed memory machines.

There are still many things to improve. One is to improve the experimental study of the task based implementation and the two reference implementations for large numbers of cores, across the nodes. Future work will also study the performances of the task-based implementation with different

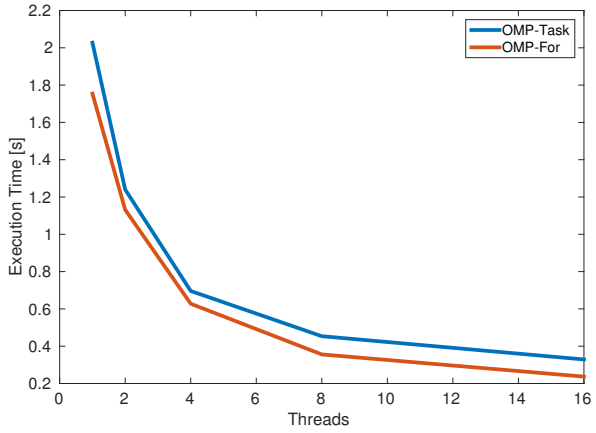


Fig. 4. Comparison of the execution time of the LU decomposition of a 1936 x 1936 matrix with different numbers of threads for the task-based (OMP-Task) and omp `parallel for` (OMP-For) implementations.

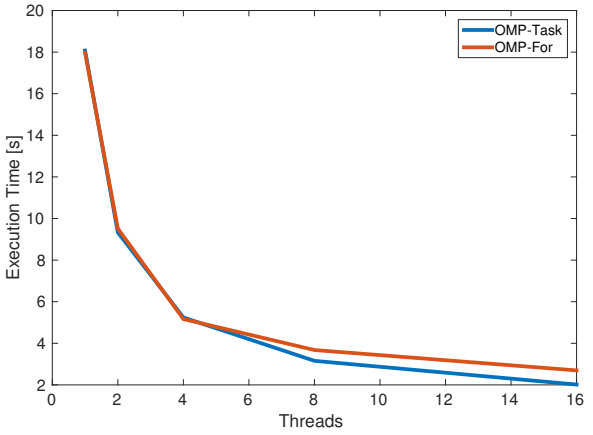


Fig. 6. Comparison of the execution time of the LU decomposition of a 4096 x 4096 matrix with different numbers of threads for the task-based (OMP-Task) and omp `parallel for` (OMP-For) implementations.

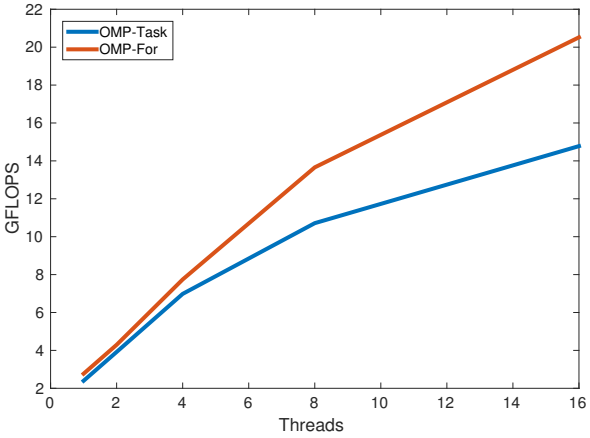


Fig. 5. Scalability of the OMP-Task and OMP-For implementations for a 1936 x 1936 matrix.

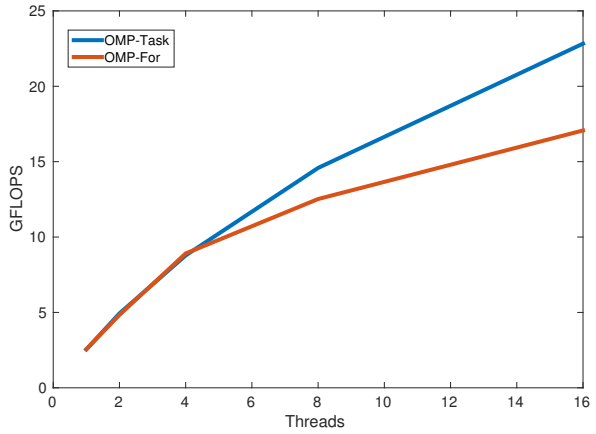


Fig. 7. Scalability of the OMP-Task and OMP-For implementations for a 4096 x 4096 matrix.

compilers. Ultimately, programming scalable algorithms for future architectures requires important design decisions for the multi-core systems. Such design decisions have to take into account task scheduling, load balancing and minimizing task overhead costs.

## Acknowledgments

This work is supported in part by the U.S. Department of Energy, Office of Science under Contract Number DE-SC0012704 through which Brookhaven National Laboratory is operated.

## References

[1] J. I. Aliaga, M. Barreda, M. Bollhöfer, and E. S. Quintana-Ortí, *Exploiting Task-Parallelism in Message-Passing Sparse Linear System Solvers Using OmpSs*, 2016.

[2] C. Bekas, A. Curioni, and I. Fedulova, "Low cost high performance uncertainty quantification," in *Proceedings of the 2Nd Workshop on High Performance Computational Finance*, ser. WHPCF '09, 2009.

[3] J. D. Booth, S. Rajamanickam, and H. K. Thornquist, "Basker: A threaded sparse LU factorization utilizing hierarchical parallelism and data layouts," *CoRR*, vol. abs/1601.05725, 2016.

[4] A. Buttari, J. Dongarra, P. Husbands, J. Kurzak, and K. Yelick, "Multithreading for synchronization tolerance in matrix factorization," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012028, 2007.

[5] B. Carpentieri, J. Liao, M. Sosonkina, A. Bonfiglioli, and S. Baars, "Using the {VBARMS} method in parallel computing," *Parallel Computing*, 2016.

[6] J. Dongarra, M. Abalenkovs, A. Abdelfattah, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. YarKhan, "Parallel programming models for dense linear algebra on heterogeneous systems," *Supercomputing frontiers and innovations*, vol. 2, no. 4, 2016.

[7] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08, 2008.

[8] ——, "Evaluation of openmp task scheduling strategies," in *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, ser. IWOMP'08, 2008.

[9] E. Elmroth and F. Gustavson, *High-Performance Library Software for QR Factorization*, 2001, pp. 53–63.

[10] P. Ghosh, Y. Yan, and B. Chapman, "Support for dependency driven

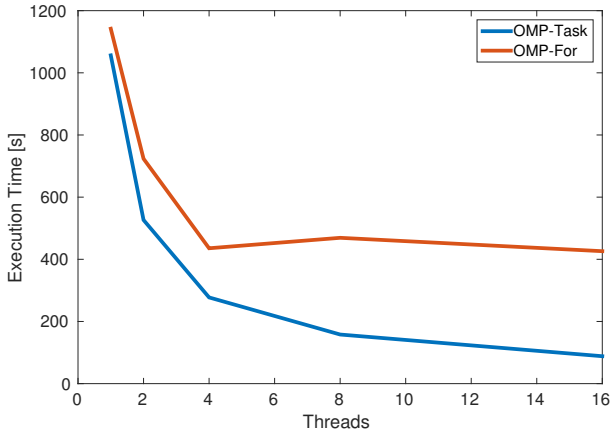


Fig. 8. Comparison of the execution time of the LU decomposition of a 16384 x 16384 matrix with different numbers of threads for the task-based (OMP-Task) and omp `parallel for` (OMP-For) implementations.

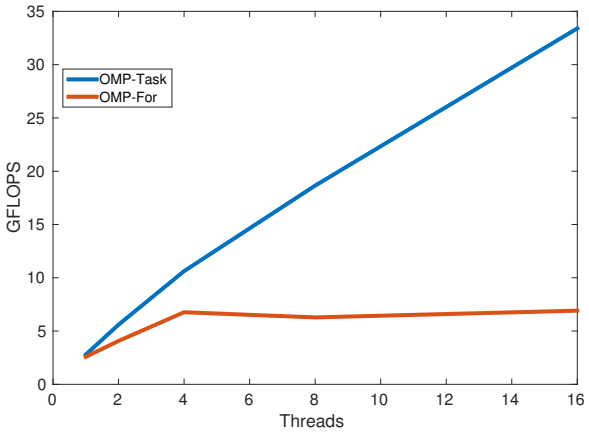


Fig. 9. Scalability of the OMP-Task and OMP-For implementations for a 16384 x 16384 matrix.

executions among openmp tasks," in *Proceedings of the 2012 Data-Flow Execution Models for Extreme Scale Computing*, ser. DFM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 48–54.

[11] P. Hohenberg and W. Kohn, "Inhomogeneous electron gas," *Phys. Rev.*, vol. 136, pp. B864–B871, Nov 1964.

[12] M. Jacquelin, L. Lin, W. Jia, Y. Zhao, and C. Yang, "A Left-Looking Selected Inversion Algorithm and Task Parallelism on Shared Memory Systems," *ArXiv e-prints*, Apr. 2016.

[13] K. Kim, S. Rajamanickam, G. Stelle, H. C. Edwards, and S. L. Olivier, "Task parallel incomplete cholesky factorization using 2d partitioned-block layout," *CoRR*, vol. abs/1601.05871, 2016.

[14] S. Li, W. Wu, and E. Darve, "A fast algorithm for sparse matrix computations related to inversion," *Journal of Computational Physics*, vol. 242, pp. 915–945, 2013.

[15] S. McGinn and R. E. Shaw, "Parallel gaussian elimination using openmp and mpi." in *HPCS*, 2002, pp. 169–176.

[16] J. M. Tang and Y. Saad, "A probing method for computing the diagonal of a matrix inverse," *Numerical Linear Algebra with Applications*, vol. 19, no. 3, 2012.

[17] A. Tran Tan, J. Falcou, D. Etiemble, and H. Kaiser, "Automatic task-based code generation for high performance domain specific embedded language," *International Journal of Parallel Programming*, vol. 44, no. 3, pp. 449–465, 2016.

[18] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier, *Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite*. Cham: Springer International Publishing, 2014, pp. 16–29.

[19] A. YarKhan, J. Kurzak, P. Luszczek, and J. Dongarra, "Porting the plasma numerical library to the openmp standard," *International Journal of Parallel Programming*, pp. 1–22, 2016.