High Performance Matrix Inversion Based on LU Factorization for Multicore Architectures

Jack Dongarra[*] [†] [‡]
University of Tennessee
1122 Volunteer Blvd
Knoxville, TN
dongarra@eecs.utk.edu

Mathieu Faverge
University of Tennessee
1122 Volunteer Blvd
Knoxville, TN
mfaverge@eecs.utk.edu

Hatem Ltaief
KAUST Supercomputing Laboratory
Thuwal, Saudi Arabia
Hatem.Ltaief@kaust.edu.sa

Piotr Luszczek
University of Tennessee
1122 Volunteer Blvd
Knoxville, TN
luszczek@eecs.utk.edu

ABSTRACT

The goal of this paper is to present an efficient implementation of an explicit matrix inversion of general square matrices on multicore computer architecture. The inversion procedure is split into four steps: 1) computing the LU factorization, 2) inverting the upper triangular U factor, 3) solving a linear system, whose solution yields inverse of the original matrix and 4) applying backward column pivoting on the inverted matrix. Using a tile data layout, which represents the matrix in the system memory with an optimized cache-aware format, the computation of the four steps is decomposed into computational tasks. A directed acyclic graph is generated on the fly which represents the program data flow. Its nodes represent tasks and edges the data dependencies between them. Previous implementations of matrix inversions, available in the state-of-the-art numerical libraries, are suffer from unnecessary synchronization points, which are non-existent in our implementation in order to fully exploit the parallelism of the underlying hardware. Our algorithmic approach allows to remove these bottlenecks and to execute the tasks with loose synchronization. A runtime environment system called QUARK is necessary to dynamically schedule our numerical kernels on the available processing units. The reported results from our LU-based matrix inversion implementation significantly outperform the state-of-the-art numerical libraries such as LAPACK (5X), MKL (5X) and ScaLAPACK (2.5X) on a contemporary AMD platform with four sockets and the total of 48 cores for a matrix of size 24000. A power consumption analysis shows that our high performance implementation is also energy efficient and substantially consumes less power than its competitors.

[*]Research reported here was partially supported by the National Science Foundation, Department of Energy and Microsoft Research.

[†]Computer Science and Mathematics Division, Oak Ridge National Laboratory, USA

[‡]School of Mathematics & School of Computer Science, University of Manchester, United Kingdom

MTAGS '11 Seattle, WA USA

1. INTRODUCTION

Forsythe, Malcolm, and Moler [?, p. 31] famously pointed out that "In the vast majority of practical computational problems, it is unnecessary and inadvisable to actually compute A^{-1}". The minority of the cases where the explicit inverse is needed include parameter estimation [?, sec. 7.5], mathematical modeling [?, p. 342ff], computing matrix sign function [?], and polar matrix decomposition [?]. Additional applications of explicit inverse come from wireless networks design [?, ?, ?, ?], optimal control theory [?, ?], and signal analysis [?]. Even though the explicit matrix inversion is not as numerically stable as the application of the L and U factors [?, ?], the loss of a few digits of accuracy is often justified in the above applications. Pair-wise pivoting, on the other hand, introduces a prohibitive loss of accuracy [?] and hence is a poor choice for matrix inversion. We therefore make use of partial pivoting version of tile LU factorization [?] that gives satisfactory accuracy for the L and U factors. From the purely performance-oriented perspective, explicit inverse has a clear advantage. In order to apply the computed inverse on a multi-column matrix of unknowns, one should use a BLAS routine called GEMM [?, ?, ?] that efficiently implements a matrix-matrix multiplication operation. On the other hand, an implicit application of inverse by applying L and U factors from the LU decomposition calls for the TRSM routine from Level 3 BLAS. While both operations may be implemented with affine and properly nested loops, the latter is always at the disadvantage as it has inherent loop-carried data dependencies and achieves a substantially smaller percentage of the peak performance of a given system.

This paper introduces a new implementation of the LU-based matrix inversion. The standard numerical algorithm, as it is implemented in LAPACK [?], is composed of four stages: 1) calculating the LU factorization, 2) inverting the upper triangular U factor, 3) solving a linear system, whose solution yields inverse of the original matrix and 4) applying backward column pivoting on the inverted matrix. Based on block formulation, this standard algorithm is characterized by artifactual synchronization points imposed not only between the different stages but also within each stage, due to the expensive fork-join paradigm. By the same token, the parallelism clearly becomes limited and the algorithm can simply not exploit and fully benefit from the fine-grain parallelism offered by the underlying multicore hardware. To overcome those bottlenecks, tile algorithms have shown promising results by drastically weakening the synchronization points as well as by exposing more parallelism to the user. Applied to the LU-based matrix inversion algorithm, the whole computation is split into fine-grain loosely-coupled computational tasks. The program data flow can then be represented as a directed acyclic graph, where nodes represent tasks and edges the data dependencies between them. A dynamic runtime system QUARK [?] is used to efficiently schedule the various tasks across the available processing units. This may actually result to an out-of-order scheduling, where tasks from multiple stages can concurrently run.

The results reported in this paper are unprecedented. Our high performance implementation achieves a 5-fold improvement against LAPACK with multithreaded MKL BLAS as well as Intel MKL and a 2.5-fold improvement against ScaLAPACK on a quad-socket AMD Opteron Magny-Cours Processor with a total of 48 cores for a matrix of size 24000. A study on power consumption is also provided showing that our high performance algorithm is also energy efficient and substantially consumes less power than the numerical libraries aforementioned.

The reminder of the paper is as follows. Section **??** gives a detailed overview of previous projects in this area. Section **??** recalls the block LU-based matrix inversion algorithm, as implemented in LAPACK [?] and explains its main deficiencies. Section **??** describes our new implementations of the matrix inversion using tile algorithms. Section **??** shows some parallel implementations details using the runtime QUARK. Section **??** presents the performance results. Comparison tests are run on shared-memory architectures against the state-of-the-art, high performance dense linear algebra software libraries, LAPACK [?] (open-source package), Intel MKL 10.2 [?] (commercial package) and ScaLAPACK [?]. Section **??** highlights the power efficiency of our high performance implementation. Finally, Section **??** summarizes the results of this paper and presents the ongoing work.

2. RELATED WORK

Matrix inversion has been an established procedure in statistical community [?, ?] but has mostly concerned symmetric positive definite matrices. As mentioned in Section **??**, there are plenty of applications of the explicit inverse for general square matrices and most of the time the numerical accuracy of obtaining the inverse is is satisfactory for the application at hand regardless of the method chosen to compute [?] – provided the original matrix is not singular nor nearly so. The symmetric positive definite matrices that originate in statistics may use Cholesky factorization as the first step of inversion. The performance of such methods have been studied on multicore computers proving tile algorithms to be beneficial for extracting parallelism [?]. The analysis of available concurrency for explicit inversion was performed by using critical paths of multiple variants of the algorithm [?].

Our work complements these efforts by studying performance and power constraints and opportunities when applied to inversion of general matrices.

3. BLOCK LU-BASED MATRIX INVERSION ALGORITHM

Block algorithms in LAPACK [?] were a software solution to the emergence of cache-based architectures. Such algorithms are characterized by a sequence of two computational phases: panel factorization and an update to the trailing submatrix. The former phase uses transformations that memory-bound while the latter applies the accumulated transformations in a block fashion (hence the name) to the trailing submatrix which, by design, is a much more cache friendly operation and, consequently, is compute-bound in practice. This two-phase sequence has an unwelcome feature of requiring unnecessary synchronization points between the steps. This in turn creates a load imbalance which might be alleviated with the look-ahead technique if the existing block-oriented code is rewritten.

The common practice in LAPACK-derived libraries is for the parallelism to be relegated to the BLAS library. Such implementations are customarily categorized as fork-join or bulk synchronous parallelism (BSP). In the end, the block implementation of LU factorization suffers from the atomicity of the pivot selection has further exacerbated the problem of the lack of parallelism and the synchronization overhead. Last but not least, the LAPACK-based implementations also uses the standard column-major layout as is practiced in Fortran. This becomes less appropriate in the current and next generation of multicore architectures due the resulting false sharing of cache lines and increased overload of the Translation Look-aside Buffer (TLB).

A valid parallelization strategy that turned out quite successful in practice [?] involves making all components of the factorization to run in parallel especially the panel factorization which would severely limit the performance for large matrices if execute serially. The success here is more remarkable as the data partitioning for such parallelization is mostly limited to one dimension: across rows in the panel and across columns for the triangular solve. Only the Schur's complement may use good scalability properties of two-dimensional data and work partitioning because it is based on matrix-matrix multiplication that is internal free from data dependencies.

Algorithm **??** shows in detail the operations and BLAS calls involved the block implementation. The four stages of the algorithm are clearly visible and, in LAPACK, they correspond to multiple functions calls. This produces the second set of synchronization points in addition to the synchronization occurring inside each routine.

We show how these synchronization overheads may be rendered unnecessary in the tile-based implementation. While at the same time, we take advantage of the fact the LU factorization is numerically stable, and, in practice, produces a reasonable growth factor.

4. TILE LU-BASED MATRIX INVERSION ALGORITHM

Synchronization reduction as well as fine-grain computations are not an option in a multicore environment anymore and one has to employ those key concepts to fully take advantage of the hardware specifications. This is exactly the aim of tile algorithms in the context of dense linear algebra. The main idea is to split the original dense matrix into tiles as shown in Figure **??**, in which elements are contiguous in memory, in order to drastically remove the overhead of TLB misses, as seen in block algorithms (see Section **??**). The common coarse grain parallelism is then replaced by a fine-grain computation, alleviating all together the artifactual synchronization

Algorithm 1 Block LU-based matrix inversion. A is an $N \times N$ matrix with a panel size of NB.

```
1:  {Stage 1: Compute $A = L \times U$ (DGETRF)}
2:  for $j = 1$ to N step NB do
3:      DGETF2($A_{j:N,j:j+NB-1}$, $IPIV_j$) {Panel factorization}
4:      DLASWP($A_{:,1:j-1}$, $IPIV_j$) {Swap behind the panel}
5:      DLASWP($A_{:,j+1:N}$, $IPIV_j$) {Swap in front of the panel}
6:      DTRSM($A_{j:j+NB-1,j:j+NB-1}$, $A_{j:j+NB-1,j+NB:N}$) {Compute block row of U}
7:      if $j + NB \leq N$ then
8:          DGEMM($A_{j+NB-1:N,j:j+NB-1}$,
9:                $A_{j:j+NB-1,j+NB-1:N}$, $A_{j+NB-1:N,j+NB-1:N}$)
10:     end if
11: end for
12: {Stage 2: Calculate $U^{-1}$ (DTRTRI)}
13: for $j = 1$ to N step NB do
14:     DTRMM($A_{1:j-1,j:j-1}$, $A_{1:j-1,j:j+NB-1}$)
15:     DTRSM($A_{j:j+NB-1,j:j+NB-1}$, $A_{j:j+NB-1,j+NB:N}$)
16:     DTRTI2($A_{j:j+NB-1,j:j+NB-1}$)
17: end for
18: {Stage 3: Solve the equation $A^{-1} \times L = U^{-1}$ for $A^{-1}$}
19: for $j = N$ to 1 step -NB do
20:     {Copy current block column of L to WORK and replace with zeros}
21:     DLACPY($WORK_{j:N,1:NB}$, $A_{j:N,j:j+NB-1}$
22:     DLASET($A_{j:N,j:j+NB-1}$, 0
23:     {Compute current block column of $A^{-1}$}
24:     if $j + NB \leq N$ then
25:         DGEMM($A_{1:N,j+NB:N}$,
26:               $WORK_{j+NB:N,1:NB}$, $A_{1:N,j:j+NB-1}$)
27:     end if
28:     DTRSM($WORK_{j:j+NB-1,1:NB}$, $A_{1:N,j:j+NB-1}$)
29: end for{Stage 4: Apply column interchanges}
30: DLASWP($A$, $IPIV$)
```

points. The parallelism is rather brought to the fore and does not reside in the BLAS calls.

Algorithm **??** describes the tile version of the four stages LU-based matrix inversion. The first stage i.e., LU factorization, has been replaced by a tile recursive parallel panel LU factorization, introduced by the authors in [?]. The major differences with the block LU factorization are twofold. The panel computation has been enhanced in terms of level 3 BLAS operations using a parallel recursive scheme. The update of the trailing submatrix now operates on tiles, which increases the degree of parallelism and allows look-ahead techniques. The second stage computes the inverse of the U factor. The LAPACK function call DTRTRI has been *unrolled* to fit the new tile algorithmic design, which generates a multitude of independent tasks and removes, within the stage, the synchronization points aforementioned. And these significant algorithmic changes apply to the third and fourth stage following the same principle.

This new tile implementation of the LU-based matrix inversion considerably weakens the synchronization points *within* each of the four stages. But it can clearly be seen also that the in-between synchronization points may be removed, whenever data dependencies permit. Indeed, the second and third stages can start executing while the first stage has not finished yet. However, the fourth stage i.e., column interchange application, requires that previous stages are totally completed before proceeding, since no practical assump-

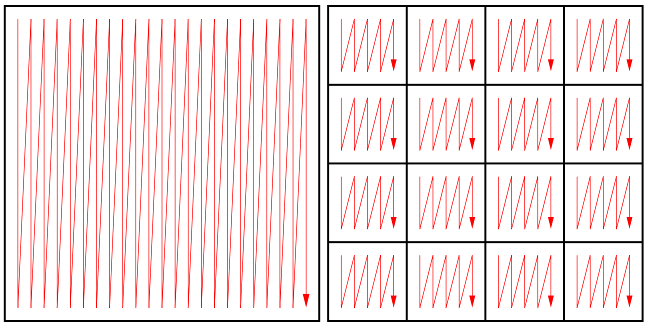


Figure 1: Column-major (left) and tile data layout (right) for a matrix.

tion is possible due to the natural atomicity of the column pivoting operation.

Once the various numerical kernels defined, a runtime environment system becomes crucial to schedule the operations on the available processing units, which is the topic of the next Section.

5. PARALLEL IMPLEMENTATION DETAILS USING THE QUARK FRAMEWORK

Our approach to extracting parallelism is based on the principle of separation of concerns. We define high performance computational kernels and submit them to the QUARK [?] scheduler for parallel execution as dependent tasks.

In fact, the execution flow in our implementation is not driven by a set of loops or recursive calls, as seen in Algorithm **??**, but rather by the data dependencies that are communicated to the QUARK [?] runtime scheduling system in a form of tasks. In practice, this results in an asynchronous out-of-order scheduling. The dynamic runtime environment ensures that enough parallelism is available throughout the entire execution as is assured by the right looking formulation of the algorithm. The advancement of the critical path for the look-ahead purposes, prominently featured in the left looking formulation, is achieved with locality-based task selection and may be enhanced with scheduler hints.

Indeed, computational tasks residing in the critical path (e.g., panel factorization) have a higher priority. On the opposite, the row interchange operations behind the panel during the first stage are not critical to pursue the computation forward and can be delayed as much as possible in favor of critical tasks scheduling. The strict right-looking variant available in LAPACK [?] and ScaLAPACK [?] cannot then be guaranteed anymore. The asynchronous nature of the DAG execution provides sufficient look-ahead opportunities for many algorithmic variants to coexist with each other regardless of the visitation order of the DAG [?].

6. PERFORMANCE RESULTS

This Section reports the performance results on a cutting-edge shared-memory multicore system based on Non Uniform Memory Access (NUMA). Directed acyclic graphs have been also generated along with execution traces to highlight the strength of our high performance implementation. A power consumption study is also presented to show its power efficiency.

6.1 Experimental Setup

Algorithm 2 Tile sequential in-place LU-based matrix inversion. A is an $NT \times NT$ tile matrix.

```
 1: {Stage 1: Compute $A = L \times U$ using parallel recursive panel}
 2: for $k = 0$ to NT−1 do
 3:    CORE_DGETRFR($A_{k,k}$, $IPIV_k$)
 4:    for $n$ = k+1 to NT−1 do
 5:       CORE_DLASWP($A_{k,n}$, $IPIV_k$) {Apply row interchange after the panel}
 6:       for $m$ = k+1 to NT−1 do
 7:          CORE_DGEMM($A_{m,k}$, $A_{k,n}$, $A_{m,n}$)
 8:       end for
 9:    end for
10:    for $n = 0$ to $k-1$ do
11:       CORE_DLASWP($A_{k,n}$, $IPIV_k$) {Apply row interchange behind the panel}
12:    end for
13: end for
14: {Stage 2: Calculate $U^{-1}$ (DTRTRI)}
15: for $m = 0$ to NT−1 do
16:    for $n$ = m+1 to NT−1 do
17:       CORE_DTRSM($A_{m,m}$, $A_{m,n}$)
18:    end for
19:    for $n = 0$ to $m-1$ do
20:       for $k = m+1$ to NT−1 do
21:          CORE_DGEMM($A_{n,m}$, $A_{m,k}$, $A_{n,k}$)
22:       end for
23:       CORE_DTRSM($A_{m,m}$, $A_{n,m}$)
24:    end for
25:    CORE_DTRTRI($A_{m,m}$)
26: end for
27: {Stage 3: Solve the equation $A^{-1} \times L = U^{-1}$ for $A^{-1}$}
28: for $k$ = NT−1 to 0 step −1 do
29:    for $m$ = k to NT−1 do
30:       CORE_DLACPY($WORK_m$, $A_{m,k}$)
31:       CORE_DLASET($A_{m,k}$, 0)
32:    end for
33:    for $m = 0$ to NT−1 do
34:       for $p$ = k+1 to NT−1 do
35:          CORE_DGEMM($A_{m,p}$, $W_p$, $A_{m,k}$)
36:       end for
37:       CORE_DTRSM($WORK_k$, $A_{m,k}$)
38:    end for
39: end for
40: {Stage 4: Apply column interchanges}
41: for $k$ = NT−1 to 0 step −1 do
42:    for $m = 0$ to NT−1 do
43:       CORE_DLASWP($A_{m,k}$, $IPIV_k$)
44:    end for
45: end for
```

All of our performance experiments were done on a single hardware system that we will call *MagnyCour-48*. *MagnyCour-48* is composed of a mother board with four sockets each featuring an AMD Opteron 6172 processor, code named Magny-Cours. Each processor consisted of twelve cores, which made it 48 cores in total. The operating frequency was 2.1 GHz and the main memory size was 128 GB. The theoretical peak for this machine in double precision arithmetic is 403.2 Gflop/s (8.4 Gflop/s per core). All the results were obtained with the Intel MKL 10.3.2 library which was selected as the best BLAS implementation for this system in terms of sequential and parallel performance.

6.2 Performance Comparisons

We compare five different versions of the *LU* inversion algorithm. First, we present the performances of the two versions that can be used in a shared memory system: the Netlib LAPACK library linked with the multi-threaded BLAS from MKL, as well as the multi-threaded MKL LAPACK version of the algorithm. The results from Figure **??** show that even if the MKL library has a better implementation of the LU factorization, no efforts have been made for the triangular inversion to outperform the Netlib version of LAPACK, resulting in similar performances for both libraries.

Second, we present results obtained with the MKL implementation of SCALAPACK using one process per core and the sequential MKL BLAS internally. This version shows how the distributed memory implementation, with good memory locality by default, performs on a NUMA architecture opposed to a shared-memory implementation. We observe that the performances are far better than the MKL multi-threaded version, but require the user to move to a distributed memory implementation of the algorithm. Finally, we present the results of our algorithms implemented in PLASMA library. For both algorithms, threads are bind linearly to each core thanks to the Hardware Locality library [**?**], HWLOC. This, with the QUARK scheduler, ensures a good data locality and reuse which are necessary to achieve performances on NUMA architectures. It is noteworthy to mention that we could not reproduce the same setup for MKL-based implementations since we do not have access to the software package internals. Furthermore, the *ColMajor* version takes as input the matrix in column major layout as the standard of LAPACK. The LU factorization is then performed on the matrix and a layout conversion to tiles is required to perform the three last stages as well as to return the result to the user in column major layout. The *Tile* version takes directly as input the matrix stored in the block layout format and perform all four stages without changes in the data storage. We observe on the Figure **??** than this layout provides 50Gflop/s more than the *ColMajor* version, and both outperform SCALAPACK for matrix sizes over 5000 to reach more than 60% of efficiency. The block size chosen for these experiments is 280, and 320 for the *Tile* version when the size is over 20000.

6.3 Data Flow Representation using Directed Acyclic Graphs

This section presents the DAGs of the four stages of the `GETRFR + GETRI` operations obtained with the scheduler QUARK. Figure **??** shows the DAGs of the four different stages in the routine: the LU Factorization (Figure **??**), the computation of the inverse of L (Figure **??**), the triangular solve for U^{-1} (Figure **??**) and the column swapping (Figure **??**). Figure **??** shows the interleaving of these four DAGs, with the same color code than the previous ones, when you performed them in an asynchronous way while the scheduler

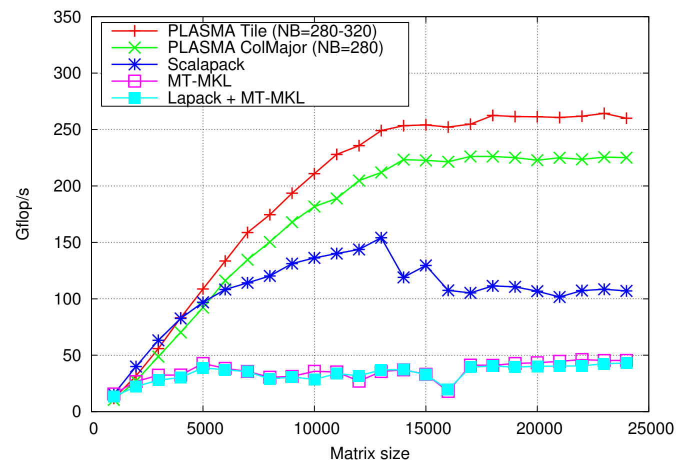


Figure 2: LU-based Matrix Inversion (DGETRF+DGETRI) on *MagnyCour-48*

preserves the dependencies. The nature of the third stage with its backward panel by panel computations naturally delays the beginning of the last stage. Indeed, the first set of swapping are applied, which are to the last column of A, is blocked by the input dependencies of this same block column within the update of the first columns of the stage three (solve for U^{-1}). Then, by limitation of the algorithm to one block column of workspace, the column swapping stage can not proceed before the last step of stage three has started and tiles $A_{m,NT-1}$ with $m \in [0, NT-1]$ have been used for the update.

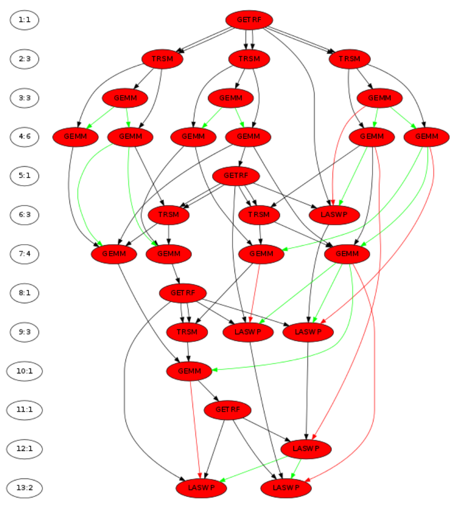
6.4 Execution Traces

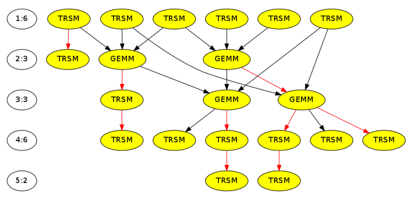
Figure **??** shows the execution traces of the PLASMA LU inversion algorithm with or without synchronization between each stage. For both figures, the LU factorization is in red, the computation of L^{-1} in yellow, the triangular solve for U^{-1} in blue and the column swapping in green. By comparison to the synchronous case shown in Figure **??**, the second trace shown in Figure **??** indicates that the interleaving steps in mainly between the first and the second stages which are respectively the LU factorization and the computation of the inverse of L. This result is expected as shown in the complete DAG of the LU inversion in Figure **??**.

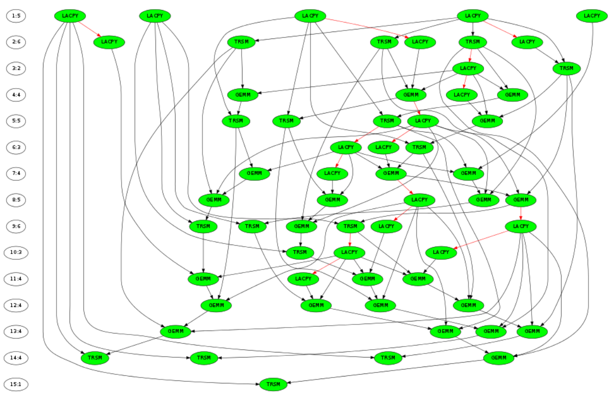
7. POWER PROFILING

The goal of this Section is to show that our high performance implementation is also energy efficient. The experiments have been conducted on a single node of a distributed system named *systemg.cs.vt.edu* from Virginia Tech composed of 324 nodes with Infiniband interconnect. Each node is a dual-socket quad-core Intel Xeon 2.8GHz (2592 cores total) with 8GB of memory. It is actually the largest power-aware compute system in the world. It has over 30 power and thermal sensors per node and relies on PowerPack [**?**] to obtain measurements of the major system components' power consumption (e.g., the CPU, memory, hard disk, and motherboard) using power meters attached to the hardware of the system.

Figures **??**, **??** and **??** show the power consumptions of the LU-based matrix inversion using LAPACK (with multithreaded MKL BLAS), MKL, and PLASMA, respectively. The findings presented below coincide with the analysis of power consumptions in dense linear algorithms studied before [**?**]. In particular, we are able to ob-


(a) DAG for the first stage: LU factorization.


(b) DAG for the second stage: inverse computation of L.


(c) DAG for the third stage: triangular solve for U^{-1}.

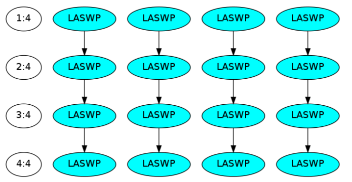

(d) DAG for the fourth stage: column swapping.

Figure 3: DAGs of the four stages composing the `GETRI` operation on a 4-by-4 tiles matrix.

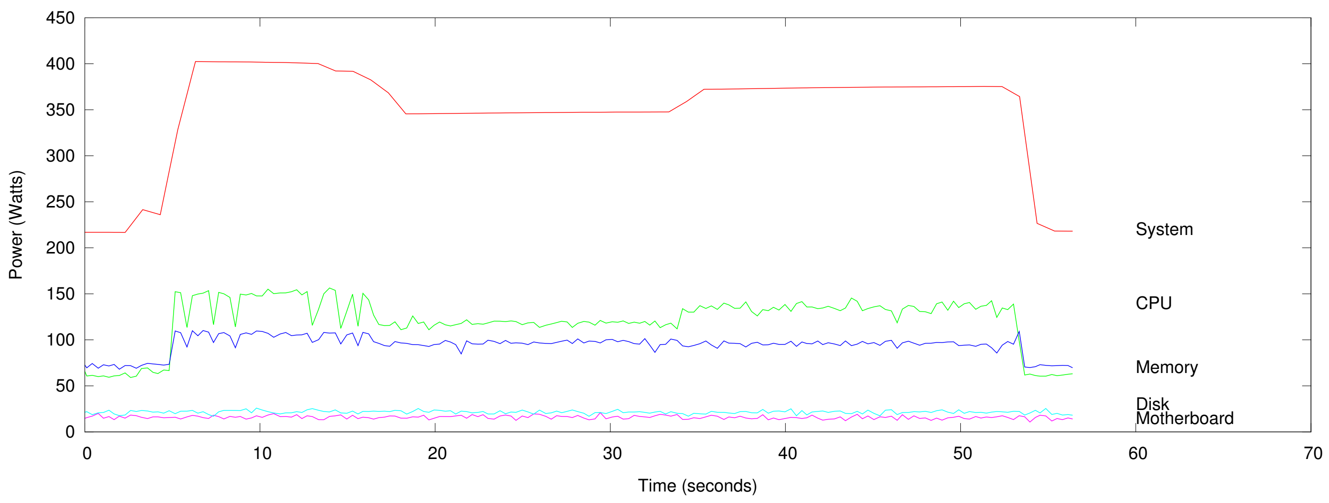


Figure 6: LAPACK LU-based matrix inversion (N=10000) on a dual-socket quad-core Intel Xeon at 2.80GHz (8 cores total).

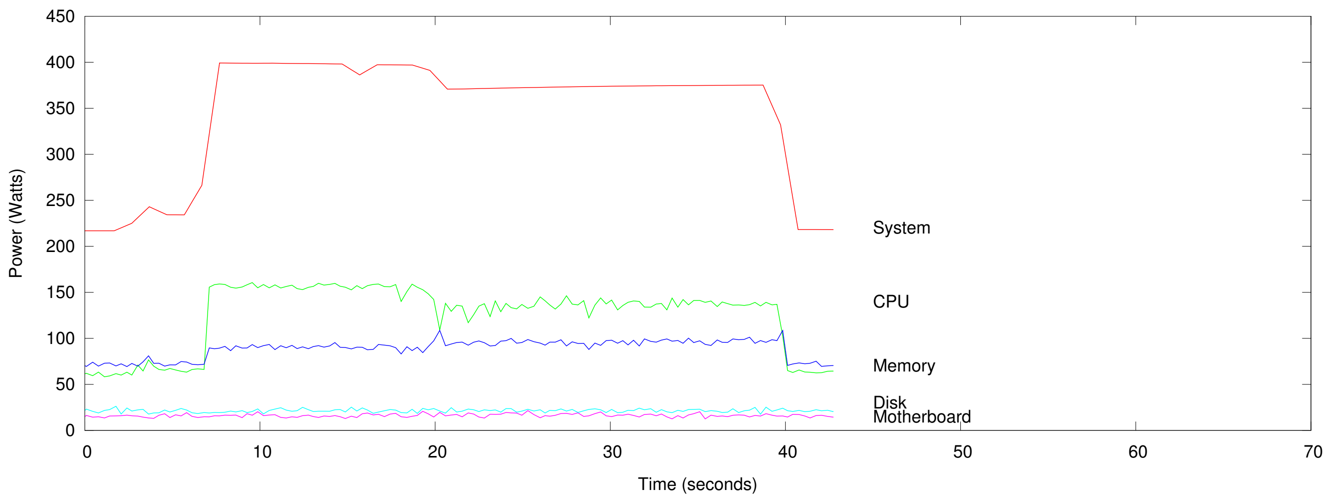


Figure 7: MKL LU-based matrix inversion (N=10000) on a dual-socket quad-core Intel Xeon at 2.80GHz (8 cores total).

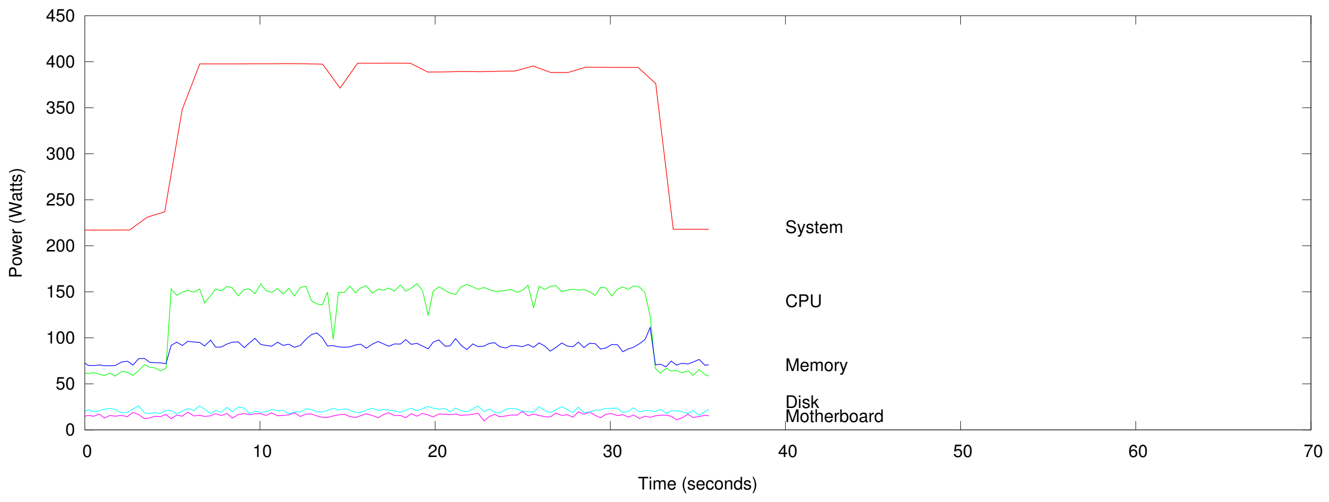


Figure 8: PLASMA LU-based matrix inversion (N=10000) on a dual-socket quad-core Intel Xeon at 2.80GHz (8 cores total).

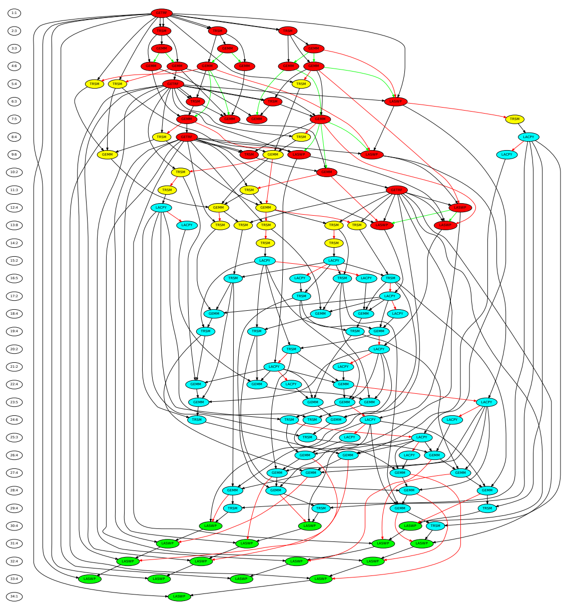
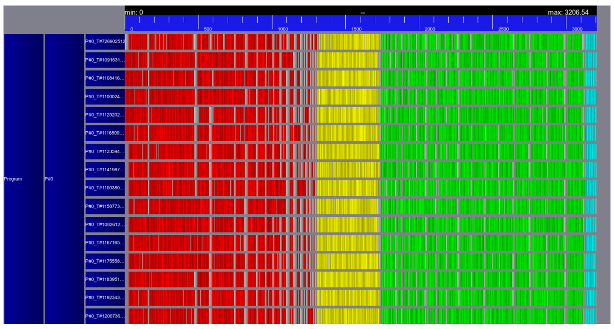
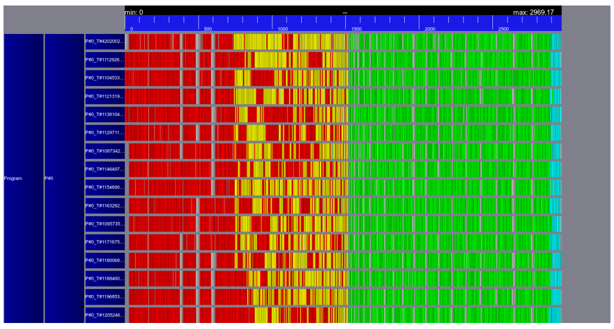


Figure 4: DAG containing the four stages which are interleaved with the dependencies preserved on a 4-by-4 tiles matrix. The same color code as in Figure ?? is used.



(a) With synchronization between each stage.



(b) With interleaved stages.

Figure 5: Execution traces of the `DGETRI` **routine with a 5000-by-5000 matrix and** $NB = 250$ **on a 16-cores architecture.**

serve phases of the computation from the power charts. Although the number of cores is small (i.e., 8 cores total), PLASMA is substantially more power efficient than LAPACK and to a lesser extent than MKL. PLASMA would even consume less power in the context of many cores against it competitors because this is where PLASMA excel the most thanks to the tremendous amount of independent computational tasks generated through tile algorithms.

8. SUMMARY AND FUTURE WORK

We have presented a tile implementation of the matrix inversion algorithm based on LU factorization. Our core shares the desirable numerical properties with the formulations that use partial pivoting. At the same time, however, we introduce plentiful opportunities for parallel execution and data partitioning that is cache-friendly and works well across complex memory hierarchies of multicore architectures. In the end, our approach yields vast improvements from the performance perspective. In fact, we observed many-fold speedup against the best implementations of numerically comparable codes.

As a future direction, we consider extending our methodology to distributed memory machines using a different version of DAG scheduler called DAGuE [**?**, **?**, **?**, **?**, **?**].

9. ACKNOWLEDGMENT

The authors would like to thank Kirk Cameron and Hung-Ching Chang from the Department of Computer Science at Virginia Tech, for granting us access to their experimental platforms.

10. REFERENCES

[1] E. Agullo, H. Bouwmeester, J. Dongarra, J. Kurzak, J. Langou, and L. Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *VECPAR'10. 9th International Meeting High Performance Computing for Computational Science*, Berkeley, CA (USA), June 22-25 2010.

[2] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[3] Y. Bard. *Nonlinear Parameter Estimation*. Academic Press, 1974.

[4] L. S. Blackford, J. Choi, A. Cleary, E. F. D'Azevedo, J. W. Demmel, I. S. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[5] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. b Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-11)*, page to appear, Anchorage, AK, May 2011.

[6] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Le marinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Distributed-memory task execution and dependence tracking within DAGuE and the DPLASMA project. Technical Report 232, LAPACK Working Note, Sept. 2010.

[7] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. Technical Report 231, LAPACK Working Note, Sept. 2010.

[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. In *Proceedings of the 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*, Anchorage, AL, USA, May, 20 2011. to appear.

[9] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed dag engine for high performance computing. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-11)*, page to appear, Anchorage, AK, May 2011.

[10] C. Boulanger and L. Ouvry. Multistage linear DS-CDMA receivers. In *Proc. IEEE ISSSTA 98*, volume 2, pages 663–667, Sept. 1998.

[11] H. Bouwmeester and J. Langou. A critical path approach to analyzing parallelism of algorithmic variants. Application to Cholesky inversion. Technical Report arXiv:1010.2000v1 [cs.DC], arXiv online archive, Oct 11 2010. Submitted to Parallel Computing. Available at arxiv.org/pdf/1010.2000.

[12] R. Byers. Solving the algebraic Riccati equation with the matrix sign function. *Linear Algebra and Appl.*, 85:267–279, 1987.

[13] A. M. Castaldo and R. C. Whaley. Scaling LAPACK panel operations using Parallel Cache Assignment. *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 223–232, 2010.

[14] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, Ostrouchov, S., A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK, a portable linear algebra library for distributed memory computers-design issues and performance. *Computer Physics Communications*, 97(1-2):1–15, 1996.

[15] J. J. D. Croz and N. J. Higham. Stability methods for matrix inversion. *IMA J. Numer. Anal.*, 12, January 1992.

[16] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):18–28, March 1990.

[17] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting Fine-Grain Parallelism in Recursive LU Factorization. *Accepted at the International Conference on Parallel Computing*, 2011.

[18] J. J. Dongarra, I. S. Duff, J. D. Croz, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM TOMS*, 16(1):1–17, March 1990.

[19] S. L. Fagin. Measurement matrix partitioning theorem. *IEEE Trans. Autom. Control*, AC-14(6):773–774, Dec. 1969.

[20] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[21] D. A. S. Fraser. *Statistics, An Introduction*. Wiley, NewYork, 1958.

[22] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, PDS-21(5):658–671, May 2010.

[23] B. Goglin, J. Squyres, and S. Thibault. Hardware Locality: Peering under the hood of your server. *Linux Pro Magazine*, 128:28–33, July 2011.

[24] G. H. Golub and C. F. Van Loan. *Matrix Computation*. John Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.

[25] A. Haidar, H. Ltaief, A. YarKhan, and J. J. Dongarra. Analysis of Dynamically Scheduled Tile Algorithms for Dense Linear Algebra on Multicore Architectures. *ICL Technical Report UT-CS-11-666, LAPACK working note #243, Submitted to Concurrency and Computations*, 2010.

[26] N. J. Higham. Computing the polar decomposition – with applications. *SIAM J. Sci. Stat. Comput.*, 7:1160–1174, 1986.

[27] G. H. Jowett. Applications of Jordan's procedure for matrix inversion in multiple regression and multivariate distance analysis. *Journal of the Royal Statistical Society. Series B (Methodological)*, 25(2):352–357, 1963.

[28] Z. Lei and T. Lim. Simplified polynomial-expansion linear detectors for DS-CDMA receivers. *IEEE Elec. Lett.*, 34(16):1561–1563, 1998.

[29] H. Ltaief, P. Luszczek, and J. Dongarra. Profiling high performance dense linear algebra algorithms on multicore architectures for power and energy efficiency. In *EnA-HPC 2011: International Conference on Energy-Aware High Performance Computing*, Hamburg, Germany, September 07-09 2011.

[30] P. McCullagh and J. A. Nelder. *Generalized Linear Models*. Chapman and Hall, second edition edition, 1989.

[31] Intel, Math Kernel Library (MKL). http://www.intel.com/software/products/mkl/.

[32] S. Moshavi, E. Kanterakis, and D. Schilling. Multistage linear receivers for DS-CDMA systems. *Int. J. Wireless Inf. Networks*, 3(1):1–17, Jan. 1996.

[33] R. T. M. Mozaffaripour. Suboptimum search algorithm in conjunction with polynomial expanded multiuser detection for uplink. *Wireless Personnal Comm.*, 24(1):1–9, 2003.

[34] D. Simon. *Optimal State Estimation*. Wiley, New York, 2006.

[35] D. C. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Transactions on Computers*, C-34(3):274–278, 1985. http://dx.doi.org/10.1109/TC.1985.1676570DOI: 10.1109/TC.1985.1676570.

[36] Q. Xie and N. xian Chen. Matrix-inversion method: Applications to Möbius inversion deconvolution. *Physical Review E*, 52(6):6055–6065, December 1995.

[37] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02*, 2011.