Master's Thesis

An in depth look at LU-decomposition on modern multi-socket hardware.

Tobias Maier

date of submission: 20. March 2015

Supervisor: Prof. Dr. Peter Sanders
Dipl.-Inform., Dipl.-Math. Jochen Speck

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 20.03.2015

Zusammenfassung

Moderne Hardware stellt neue Herausforderungen an Algorithmen. Um Rechnerarchitekturen mit mehreren Sockeln und komplexen Cachehierarchien gut auszulasten, müssen Algorithmen mit den Stärken und Schwächen der Hardware im Blick entworfen werden. Richtig eingesetzt kann moderne Hardware auch neue und interessante Optimierungsansätze bieten. In dieser Arbeit präsentieren wir einen neuen Ansatz zur Arbeitsverteilung, anhand einer LU-Zerlegung. Bei der Arbeitsverteilung betrachten wir Möglichkeiten gemeinsame Daten zwischen verschiedenen Teilaufgaben wiederzuverwenden. Sofern dies möglich ist werden die entsprechenden Teilaufgaben so auf Rechenkerne zugewiesen, dass sie gemeinsame Eingaben über den L3 Cache teilen können.

Das bestimmen der LU-Zerlegung einer Matrix ist eine der wichtigsten numerischen Aufgaben. Die LU-Zerlegung wird verwendet, um Matrizen zu invertieren, ihre Determinante zu bestimmen und um lineare Gleichungssysteme zu lösen. Der Algorithmus zur LU-Zerlegung ist außerdem repräsentativ für viele weitere numerische Berechnungen. Dies ist auch der Grund, weshalb er der Hauptbestandteil des berühmten LINPACK Benchmarks ist, welcher verwendet wird, um die stärksten Supercomputer für die TOP500 Liste zu bestimmen.

Es ist wichtig, die Laufzeit der LU-Zerlegung zu verbessern ohne die numerische Genauigkeit zu verschlechtern. Deshalb verwenden wir den selben numerischen Algorithmus wie die anerkannte PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures) Bibliothek. Durch unsere Anpassungen an die Arbeitsverteilung, die es uns ermöglichen Teilaufgaben mit gemeinsam genutzten Daten besser zu verteilen, erreichen wir eine Performanzsteigerung, zwischen 15% und 29%.

Abstract

Modern hardware makes new demands on algorithms. To properly utilize multi-socket architectures with non-trivial cache hierarchies, algorithms have to be designed with the hardware's strengths, and weaknesses in mind. If properly addressed, modern hardware offers new, and interesting optimization possibilities. In this thesis we present a new scheduling approach, using the example of an LU-decomposition. Our scheduler considers data reuse opportunities between different subtasks, and schedules these subtasks in a way that lets them share common inputs through the L3 cache.

The LU-decomposition of matrices is one of the most important numerical algorithms. It is used for matrix inversion, to compute the determinant of a matrix, and to solve systems of linear equations. The algorithm for LU-decomposition is also representative for many other numerical computations. This is the reason, why it is part of the famous LINPACK benchmark, that is used to rank the most powerful supercomputers in the TOP500 list.

It is important, to improve the performance of the LU-decomposition, without decreasing its numerical accuracy. Therefore, we use the same numerical algorithm for the LU-decomposition as PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures), which is a current state of the art implementation. By adapting the scheduling scheme, to take better advantage of data sharing, we achieve performance increases between 15%, and 29%.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

Modern computers become faster, and faster. Today, there are two main factors that cause the increase in computing power, the increase in the number of cores per processor, and the fact that modern cores can perform more and more computations per cycle. Through Advanced Vector Extensions (AVX and AVX2), cores can compute up to eight floating point operations per cycle (16 with AVX2). At this rate of computation, memory access becomes an important bottleneck. Especially the latency of uncached memory accesses can be detrimental for high performance computations. This trend is further intensified on multi socket-hardware by non-uniform memory access (NUMA). A single memory access can slow down computations significantly. To counteract this problem, manufacturers use complex cache hierarchies. Modern processors have multiple levels of caches, some of which are shared between cores. It has become an important, and difficult task, for programmers to optimally use these complex cache architectures.

We chose the LU-decomposition problem to show that even problems which were long believed to be compute bound can profit from dedicated cache optimizations. LU-decomposition is interesting in this context for multiple reasons. (1) It is one of the most important numerical algorithms. LU-decompositions are the main tool to solve systems of linear equations, which is an important subtask for many applications. Systems of linear equations are used in many different areas, for example in engineering, physics, chemistry, computer science, and economics. The LU-decomposition can also be used to invert matrices, and to compute determinants. (2) The algorithm for LU-decomposition is characteristic for many dense linear algebra computations. If our methods succeed at accelerating the LU-decomposition, it is very likely that similar ideas can be used to accelerate other numerical linear algebra computations. (3) Because of its importance, there are already optimized implementations of LU-decomposition algorithms, like the one used in the PLASMA [11, 1] library (Parallel Linear Algebra for Scalable Multi-core Architectures). Therefore, even small improvements can be worthwhile, and show the novelty of our approach. Well optimized solutions can not only be used as competitors in our experiments, we can also use them as a starting point for our own solution. (4) LU-decomposition is the core of the famous LINPACK [12] benchmark. This benchmark is used to rank the worlds most powerful supercomputers in the TOP500 list [9].

For large numerical computations like the LU-decomposition it is common to generate parallelism by dividing the computation into small sequential subtasks. Oftentimes, many subtasks can be executed independently creating parallelism. This execution method is often called DAG-scheduling (Directed Acyclic Graph), because one can construct a graph with the subtasks as vertices, and the dependencies between subtasks as edges (see Section 2.3). In the context of numerical linear algebra computations, subtasks are often routines of highly optimized BLAS libraries (Basic Linear Algebra Subprograms). Therefore, it is more interesting to optimize the scheduling of subtasks than to optimize the subtasks themselves.

Even in modern, state of the art libraries like PLASMA very little care is taken to optimize data sharing between independent subtasks. Our approach is to take into account, when multiple subtasks access the same input data. We show, that scheduling subtasks with data sharing opportunities in mind, can greatly reduce the number of cache misses, and in turn improve the running time of the algorithm. To reduce the number of cache misses, we exploit the fact that modern processors use shared caches, which connect multiple cores of one processor. Scheduling tasks, that read the same memory regions, onto cores that share a common cache, can lead to significant speedups, because the common memory region will only be cached once.

In addition to our cache optimizations, we explicitly address, and optimize the NUMA-behavior of our implementation. By controlling the NUMA-distribution of the matrix, we are able to schedule tasks in a way, that minimizes the access to non-local memory. Tasks are scheduled on the node, that holds the majority of their input tiles. This accelerates data accesses even in case of cache faults.

What we propose, can be described as a data-computation-co-scheduling. Algorithms can only reach their optimal performance, if data flow, and work flow are considered together. To analyze the data flow and the usage of shared caches during the computation we develop a theoretical cache model. This cache model predicts the contents of shared caches during the computation depending on the order, and placement of executed subtasks. Using this theoretical model, we optimize the scheduling process that is used for our algorithm's subtasks.

An important aspect for all numerical linear algebra computations is accuracy. There are papers, that propose a trade-off between the speed, and the accuracy of a computation. We aim to achieve the best possible accuracy. Therefore, we use the same numerical algorithm that is used within PLASMA to compute the LU-decomposition. This algorithm was described by Dongarra et al. [10], and we use it as a starting point for our implementation. Most of our code is taken directly from PLASMA, and nearly all of our changes are part of the scheduling process. Notably, all of our numerical computations are exactly the same operations as in PLASMA, and therefore produce the same result with the same accuracy. An analysis of the algorithm's accuracy can be found in the original description [10].

Through our changes in the scheduling strategy, we achieve a performance improvement of 15% on big matrices (32768 rows/columns), and up to 29% on smaller matrices (8192 rows/columns).

1.2 Related Work

In principle, there are two fields of work, that we consider to be related to our findings. The first field is related through the common topic of LU-decomposition, while the second field is related through similar optimization methods.

As described in Section 1.1, LU-decompositions are a very important for common problems like solving systems of linear equations. For that reason, there are already many publications on the efficient computation of LU-decompositions. Over the last decades a lot of work has gone into the efficient parallelization of the LU-decomposition.

It has become clear, that LAPACK [2], which has been the standard library for linear algebra computations for a long time, has performance issues on modern

multi-core machines. The reason for this is, that LAPACK creates parallelism with a fork-join based approach. Instead of splitting the computation into small subtasks, that can be computed independently. LAPACK uses bigger subtasks, that are executed in parallel. This results in a significant synchronization overhead. Modern contributions to the area of dense matrix computations rely on the dynamic scheduling of fine grained, (mostly) sequential subtasks. Through this approach, the synchronization between different threads is reduced significantly.

As we already mentioned, the algorithm for LU-decomposition can be split into subtasks, that can be executed somewhat independently. There are different kinds of subtasks, within the LU-decomposition. The most complex type of subtasks is called a panel task (see Section 2.2.3). These panel tasks, are considered too big, to be executed sequentially. Therefore, it has been a focus of many papers to improve their execution. Buttari et al. [7], and Quintana-Ortí et al. [17] seek to improve the panel parallelization, by changing the pivoting scheme (see Section 2.2) such that the panel factorization can be split into smaller subtasks. They use what they call *incremental pivoting* to parallelize panel tasks more efficiently. This method is also employed by Kurzak et al. [14] in their work on dynamically scheduling dense linear algebra workloads.

But, changing the pivoting scheme comes at a cost in numerical stability. Dongarra et al. [10] show that the loss of precision over the more commonly used partial pivoting can be significant. Dongarra et al. also show a recursive variant of the panel algorithm – using *partial pivoting* – that can be parallelized. This algorithm combines high accuracy with high performance. It is the algorithm that is used within the PLASMA [11] library, and because we used PLASMA as a starting point, for our implementation, this is also the algorithm that we use for this thesis.

Related works, that use similar optimization, and scheduling techniques can be found in the field of DAG-scheduling. A lot of work has been done in the context of DAG-scheduling, as it is applicable to many problems. Kwok, and Ahmad [15] give a good overview of many aspects of DAG-scheduling.

Bosilca et al. [5] present a framework for high performance computations on distributed hardware, using DAG-scheduling. Their goal is to simplify development by offering a run-time system, which schedules tasks, and data-flows. This framework has already been used by Bosilica et al. [6] to implement some distributed numerical linear algebra functions, including a LU-decomposition.

One technique, that we are using for our optimizations is to manipulate the execution order of subtasks in a way that allows subtasks to reuse cached memory slots. Cheng et al. [8] show that parallel depth first traversal of the task-DAG can reduce the number of cache misses compared to more traditional work stealing methods. While this method shows large improvements for many problems, they did not reach significant improvement for the LU-decomposition.

Apart from application level scheduling, there are also more indirect methods, to improve the cache locality of parallel work loads. Tam et al. [18] use an approach, where they change the operating systems scheduling mechanisms. With this method, they try to optimize the allocation of threads to processors. This is done, by monitoring the memory accesses of applications with performance counters, and rescheduling threads appropriately.

1.3 Overview

We begin this thesis with the Preliminaries in Section 2. In this section, we describe the background of our work. It contains theoretical backgrounds, like the definition of an LU-decomposition, and a short introduction to DAG-scheduling as well as the high level description of the numerical algorithm that we use for our factorization.

In Section 3 we explain our optimization techniques from a theoretical point of view. There, we develop the theoretical cache model, that we base our cache optimizations on, and we describe the the cache, and NUMA-optimizations that we implemented.

After this theoretical view on our optimizations, we use Section 4 to detail the architecture of our scheduler, and the scheduling process. Here we also describe how we incorporate the ideas – developed in Section 3 – into the scheduling approach. In Section 5 we describe the detailed implementation, of our scheduling process. There, we describe details, that are important for the scheduling of tasks within our implementation. This includes the data structures of our scheduler, the sequence of a threads work cycle, and the process of readying, and scheduling subtasks.

To validate our findings, we conducted the performance experiments shown in Section 6. There we compare the performance of our implementation to that from the PLASMA library. We also conduct a detailed analysis, of our optimization techniques, and their influence on the running time of different subtasks of the algorithm. In the end, we summarize our findings in Section 7.

2 Preliminaries

This section contains some theoretical background for this thesis. We begin with Section 2.1 where we explain different matrix layouts. Then we use Section 2.2 to define the problem of LU-decomposition. Within this section, we also introduce the numerical algorithm that we use to compute the decomposition. It works by splitting the work of the computation into small subtasks. These subtasks can either be executed sequentially – one after the other – or they can be reordered. Many subtasks are independent from one another. They can be executed in parallel.

In Section 2.3 we describe DAG-scheduling which is a method to schedule independent subtasks. Subsequently, we use Section 2.4 to show the task structure of the algorithm for LU-decomposition when it is scheduled with a DAG-scheduler.

Then, we describe the architecture of modern computers in Section 2.5. There, we describe the hardware properties that we exploit to achieve better running times.

2.1 Matrix Layout

There are multiple ways to efficiently store matrices. Classically matrices are stored in the column-major format. This means that all elements of the matrix are stored in one consecutive piece of memory, in a column by column ordering (see Figure 1a). This data layout is very popular, because it is the native format

used in Fortran, and therefore LAPACK [2], which is a widely-used linear algebra interface written in Fortran.

There are other data layouts which are designed for better data locality. Such layouts have proven effective especially for matrix multiplication. Examples for this are hierarchical storage options and layouts that store matrices along space-filling curves [3, 13] (see Figure 1b). Layouts of this kind can often lead to *cache oblivious* algorithms – algorithms that effectively use caches without being tuned to specific cache sizes.

We are using a *blocked data layout* [7]. This means, that the matrix is stored in square blocks, which we call tiles. Each tile is stored in a column major format (see Figure 1c). This layout is used in modern linear algebra software like PLASMA [11]. It improves data locality, but submatrices are still stored in the column major format. This enables us to use common BLAS libraries like Intel's *MKL* (Math Kernel Library) for subtasks of our computation.

In this thesis we are handling a lot of tiled matrices, therefore we want to specify some naming conventions.

Definition 2.1 (Matrix Layout)**.** *All matrices used in this paper are square matrices of size $n_{glob} \times n_{glob}$ (unless specified otherwise). A tiled matrix is a matrix that is subdivided into $n \times n$ tiles of size $b \times b$.*

$$n = \left\lceil \frac{n_{glob}}{b} \right\rceil$$

If $n_{glob} \neq n \cdot b$ then there are offsets in the lowest tile-row (rightmost tile-column).

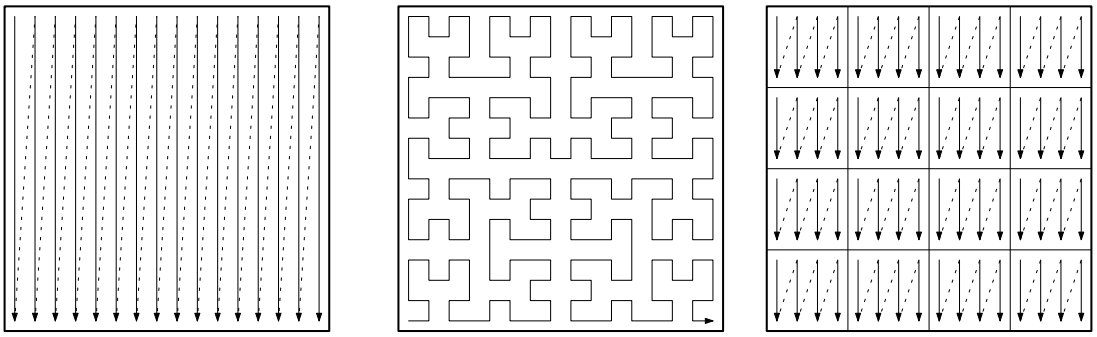
We call the tile in matrix A at position (i, j) A_{ij}. The top left tile is $A_{1\,1}$, and the bottom right tile is $A_{n\,n}$. We also use $A_{i\ldots k\,j}$ as a shorthand, for the tiles A_{ij}, $A_{i+1\,j}$, …, A_{kj} (part of a column), and $A_{i\,j\ldots k}$ for A_{ij}, $A_{i\,j+1}$, …, A_{ik} (part of a row).

As previously mentioned, this data layout combines good data locality with the ability to use existing libraries. It can also facilitate the parallelization of matrix algorithms. An algorithm that operates on a matrix can often be split into different subtasks that only work on certain tiles of the matrix. Two subtasks that work on different parts of the same matrix can usually be executed in parallel. This facilitates parallel execution without unnecessary synchronizations (or data locking).

2.2 LU-Decomposition

The LU-decomposition is a factorization of a matrix into a *lower triangular matrix* (zeros above the diagonal) and an *upper triangular matrix* (zeros below the diagonal). These two matrices can in turn be used to solve linear systems of equations or to invert the original matrix.

There are different varieties of LU-decompositions. Before we describe the version that is commonly used (also used for this thesis), we begin by defining the most basic variation of LU-decomposition.



(a) *Column Major* (b) *Space Filling Curve* (c) *Blocked Data Layout*

Figure 1: *Different data layouts for storing matrices.*

Strict LU-Decomposition

Definition 2.2 (Strict LU-Decomposition)**.** *The matrix A is said to have an LU-decomposition, if there exists a factorization $A = L \cdot U$ where:*

- *L is a lower triangular matrix (only zeros above the diagonal) with ones on the diagonal.*
- *U is an upper triangular matrix (only zeros below the diagonal).*

If there exists such a decomposition for a matrix A then it is unique. This is forced through the constraint that L must have ones on its diagonal.

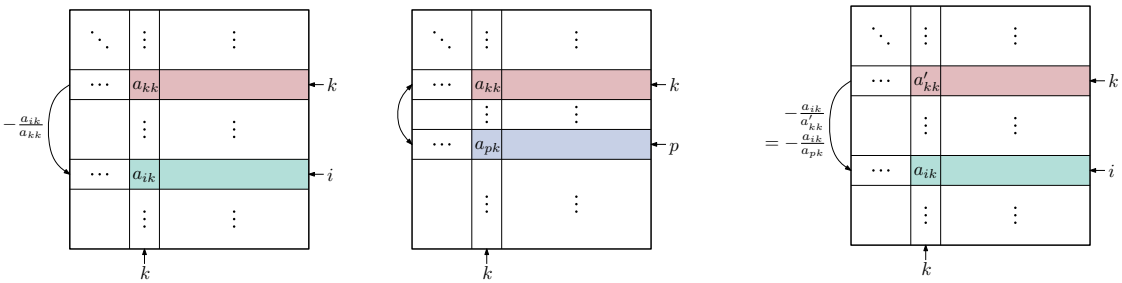
There are two problems with this general definition of LU-decompositions. The first problem is that an LU-decomposition of this form is not possible for some matrices. Especially *singular matrices* (not invertible) and *permutation matrices* (exactly one "1" per row, and column) often do not have an LU-decomposition. The second problem is the numerical stability of such a decomposition. This kind of decomposition is only guaranteed to be numerically stable in certain special cases, for example for *diagonally dominant matrices* (diagonal elements are greater than the sum of other elements in the same row).

LU-Decomposition with Row Pivoting (partial pivoting) To reduce the problems of strict LU-decompositions, it is very common to allow *row pivoting* for the decomposition. Row pivoting means that rows of the original matrix A are permuted during the decomposition. This can improve the numerical stability because it can prevent that large numbers are divided by very small numbers. Rows are usually exchanged according to a pivoting scheme. We use the *partial pivoting* scheme, which is the most common pivoting scheme, that in practice leads to good numerical stability.

Definition 2.3 (LU-Decomposition with Row Pivoting)**.** *An LU-decomposition of A with row pivoting is a decomposition where $P \cdot A = L \cdot U$ and:*

- *$L \cdot U$ is a strict LU-decomposition of $P \cdot A$ (as described in Definition 2.2).*
- *P is a permutation matrix ($P \cdot A = A$ with permuted rows).*

If we allow row pivoting, then every squared matrix has an LU-decomposition. Throughout this thesis LU-decomposition will always mean an LU-decomposition with partial pivoting (unless specifically mentioned otherwise).



(a) *Without partial pivoting – $\frac{a_{ik}}{a_{kk}}$ can lead to numerical problems.*

(b) *The pivoting step – the pivot row is swapped into row k.*

(c) *Row elimination after row exchange – $|\frac{a_{ik}}{a'_{kk}}| < 1$.*

Figure 2: *Description of partial pivoting. The elimination of values below the diagonal with, and without partial pivoting.*

Partial Pivoting During the algorithm used for the decomposition, we eliminate values in column k (below the diagonal) by subtracting a multiple of row k (see Figure 2a). If the entry a_{kk} is zero, this leads to a division by zero. This is the reason why not every matrix has a strict LU-decomposition. Even if $|a_{kk}| > 0$, there can be numerical problems if a_{kk} is small compared to other entries in column k. These numerical problems can reduce the overall accuracy of the computation significantly.

Partial pivoting reduces these inaccuracies by choosing a pivot row and swapping that row into row k (see Figure 2b). The pivot row is chosen as the row that has the highest absolute value in column k. This way $|\frac{a_{ik}}{a_{kk}}| \leq 1$ can be guaranteed for every "elimination factor" $\frac{a_{ik}}{a_{kk}}$ (see Figure 2c).

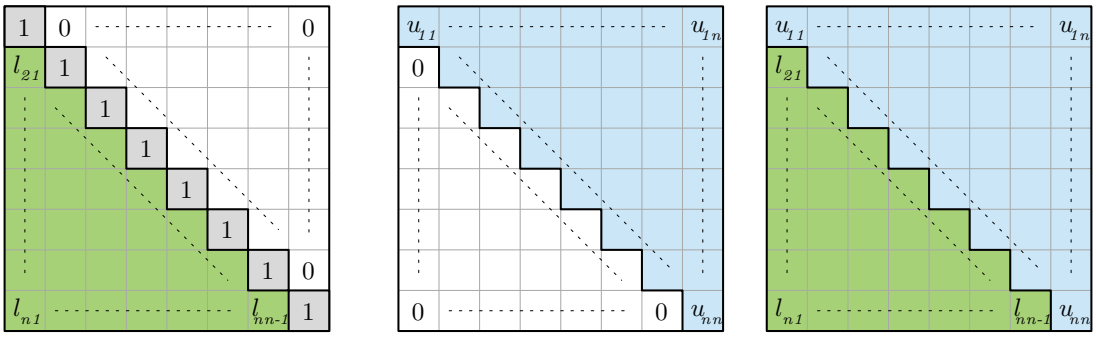
In this thesis we do not go into further detail on the numerical accuracy of our LU-decomposition. The numerical accuracy of our LU-decomposition should be equal to the accuracy of any similar implementation using partial pivoting. Since our implementation is closely related to the implementation used in PLASMA, we expect our accuracy to be the same. Results on the accuracy of PLASMA's LU-decomposition can be found in [10].

2.2.1 Representation of the LU-Decomposition in one Matrix

Memory is a concern when factoring big matrices. Therefore, it is preferable not to allocate multiple matrices at once. Especially, if two of these matrices will be triangular (one half of all elements are zero). Luckily, L and U can be stored together in one matrix. This is possible, because L is a lower triangular matrix and U is an upper triangular matrix. The only positions where non-zero elements of L and U collide are on the diagonal, but all diagonal entries of L are one. Thus, both triangular matrices can be stored in one matrix, in which the zero entries and L's diagonal are stored implicitly (see Figure 3).

The LU-decomposition can even be computed in place without allocating any new matrices. Both algorithms that we present throughout this section are examples for in-place LU-decompositions (see Section 2.2.2, and Section 2.2.3).

Only the permutations have to be stored externally. However, storing a permutation matrix can be done a lot less memory consuming than a regular matrix. The permutation matrix is used to create a row permutation of the original ma-



(a) *lower triangular (with ones on diagonal)*

(b) *upper triangular*

(c) *combined (lower diagonal stored implicitly)*

Figure 3: *Representation of L, and U in one matrix.*

Algorithm 1: Gaussian Elimination with Pivoting

`Input`: matrix $A \in \mathbb{R}^{n_{glob} \times n_{glob}}$

1 `for` $1 \leq k \leq n_{glob}$ `do`
2 $\quad q_k \leftarrow \arg\max_{k \leq i \leq n} |a_{ik}|$ *// find pivot row with partial pivoting*
3 $\quad$ `exchange_rows`(k, q_k) *// Figure 4a*
4 $\quad$ `for` $k+1 \leq i \leq n_{glob}$ `do` *// Figure 4b*
5 $\quad\quad a_{ik} \leftarrow \frac{a_{ik}}{a_{kk}}$ *// generates entry of L*
6 $\quad$ `for` $k+1 \leq j \leq n_{glob}$ `do` *// Figure 4c*
7 $\quad\quad$ `for` $k+1 \leq i \leq n_{glob}$ `do`
8 $\quad\quad\quad a_{ij} \leftarrow a_{ij} - a_{ik} \cdot a_{kj}$ *// update matrix with row operations*

`Output`: matrix A containing L and U, and the pivoting vector $Q = (q_1, \ldots, q_n)$

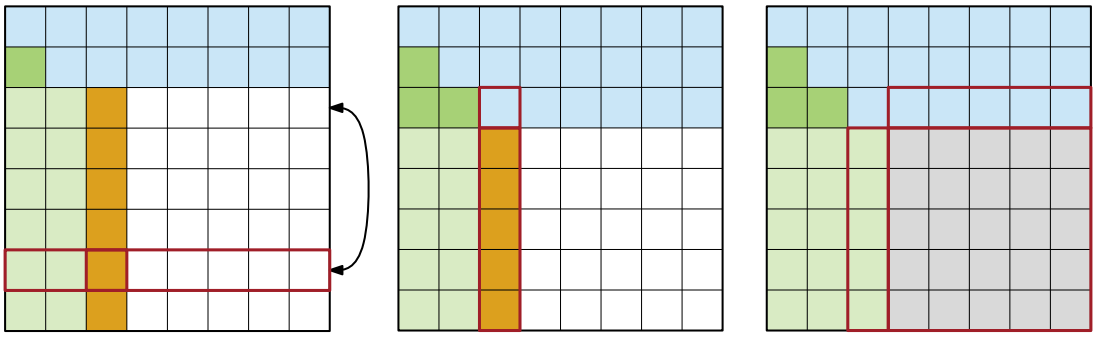
trix A. So instead of saving all permutations in the form of a matrix, we are storing permutations in the form of an integer vector. Every value q_i of this vector denotes that row i was exchanged with row q_i (compare Algorithm 1 line 2, and 3). Saved like this, the permutation matrix only uses memory space linear in the matrix size n_{glob}.

2.2.2 Gaussian Elimination

Originally the LU-decomposition was computed via the Gaussian Elimination algorithm that can be seen in Algorithm 1 and Figure 4. This is the same algorithm that is commonly used when manually solving linear systems of equations. The matrix is triangulated – from top left to bottom right – by eliminating entries under the diagonal.

For this algorithm, we assume that we work on a matrix with tile size $b = 1$. So $n_{glob} = n$ and A_{ij} is the element in position (i, j) of A. As it is customary to use lower case letters for individual matrix elements, we use lower case letters while $b = 1$.

There is one global loop (Algorithm 1 line 1 to 8) over k. Each *global iteration* k works in three steps:



(a) *Find the optimal pivot row and swap it into the row k.*

(b) *Compute factors for row operations – write them into column k.*

(c) *Execute row operations on the rest of the matrix.*

Figure 4: *One iteration of the LU-decomposition via Gaussian Elimination*

1. (lines 2-3) We find a pivot row. This is the row with which values under the diagonal are eliminated. Then, the pivot row is exchanged with row k. We call this a *row exchange* (see Figure 4a).

2. (lines 4-5) Afterwards, we compute all factors which are later used to eliminate values under the diagonal. These factors make up the k-th column of L (see Figure 4b). Hence, L can be seen as a history of all used elimination factors.

3. (lines 6-8) At last, we subtract multiples of the right side of the pivot row (row k right of column k) from each lower row. The corresponding factors for each row were computed in step two and are stored in the k-th column of L. We call these subtractions *row operations* (see Figure 4c).

Each iteration of this loop finalizes row k of matrix U. It also computes all values that are in column k of matrix L. The values of column k might still be reordered through row exchanges of future iterations. All values above row k are final.

In step one the pivot row is chosen according to the pivoting scheme. Partial pivoting is the most commonly used pivoting scheme. It is a method that in practice achieves good numerical stability while still being fast, and easy to implement. Partial pivoting means that we always use the row with the biggest absolute value in column k as the pivot row. Therefore, a_{kk} is be the biggest value in column k. This makes the computation of a_{ik}/a_{kk} more numerically stable.

The problem with this algorithm is that it heavily uses matrix-vector products (level 2 BLAS operations), these cannot fully take advantage of modern hardware. Especially the matrix update (see Figure 4c), which is by far the biggest portion of the algorithm's work, could be executed more efficiently if it consisted of matrix-matrix products (level 3 BLAS operations). Therefore, modern libraries (for example PLASMA) use a slightly different version of this algorithm.

Algorithm 2: Tiled LU-Decomposition with Pivoting

```
Input: a tiled matrix A ∈ (ℝ^{b×b})^{n×n}
1 for 1 ≤ k ≤ n do
2     A_{k..n k}, Q_k ← PANEL(A_{k..n k})          // finds pivots + column updates
3     for k+1 ≤ j ≤ n do
4         A_{k(..n) j} ← TOP(A_{k(..n) j}, A_kk, Q_k)   // exchange rows + computation
5         for k+1 ≤ i ≤ n do                        // update the rest of the matrix
6             A_ij ← SUBMATRIX_UPDATE(A_ij, A_ik, A_kj)
7     for 1 ≤ i < k do                              // exchange rows left of the panel
8         A_{k..n i} ← BEHIND_PANEL_PERMUTATION(A_{k..n i}, Q_k)
Output: matrix A that now contains L and U, and the pivoting vector
        Q = (Q_1, ..., Q_n)
```

2.2.3 Algorithm for Tiled Matrix Layouts

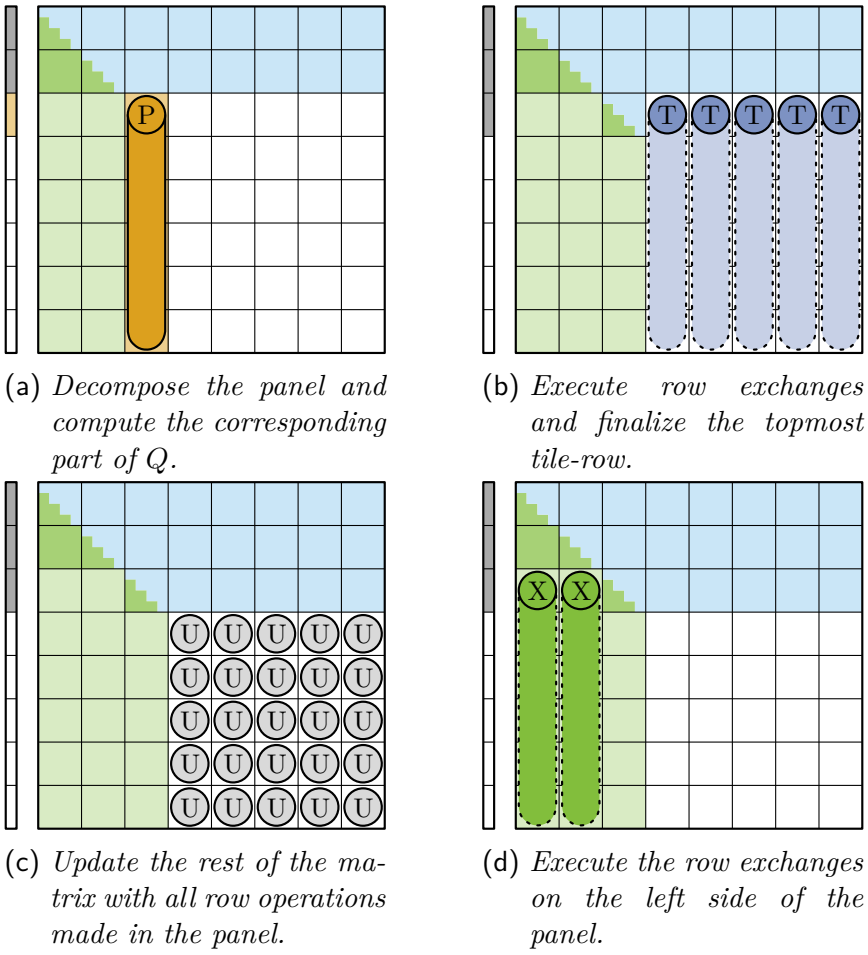
The LU-decomposition algorithm for tiled matrices, that can be seen in Algorithm 2 and Figure 5, effectively recreates Gaussian Elimination on a matrix of tiles. The main difference is that it combines row exchanges and row operations from multiple columns (one tile-column) before applying them to the rest of the matrix. For each *global iteration* over k (lines 1-8 in Algorithm 2), we call tile-column k the *panel* of iteration k ($A_{k..n\,k}$; orange in Figure 5a). The panel is comparable to column k of iteration k during the Gaussian algorithm. At the beginning of each iteration, all permutations and elimination factors within the panel are computed. Afterwards, the rest of the matrix is updated with all row exchanges and row operations made within the panel. The algorithm splits its work load into four major classes of subtasks:

PANEL-tasks (or *P-tasks*): In each global iteration, there is exactly one P-task. It computes the LU-decomposition of the panel and the corresponding subsection of the permutation vector Q (see Figure 5a). In Section 5.4 we go into further detail about the implementation of the panel algorithm that we use. At this point it is only important to know that the panel algorithm is parallelized which means that there are multiple cores that work on each P-task cooperatively.

The P-task can be implemented very similar to the Gaussian Elimination algorithm from Section 2.2.2. The P-task finds the pivot row for each column within its tile-column and computes the corresponding tile-column of matrix L. A single P-task has an asymptotic complexity of $\mathcal{O}((n-k+1)\cdot b^3)$ flops. All n P-tasks combined have an asymptotic complexity of

$$\textit{complexity(P-tasks)} = \mathcal{O}\left(n^2 \cdot b^3\right).$$

TOP-tasks (or *T-tasks*): They are the tasks that execute row exchanges on the right side of the panel. In each global iteration, there is one T-task per tile-column on the right side of the panel. They also compute the effects of row operations within the topmost tile-row (see Figure 5b). The row exchanges execute $\mathcal{O}(b^2)$ reads and writes (no flops), while the row operations have an



(a) *Decompose the panel and compute the corresponding part of Q.*

(b) *Execute row exchanges and finalize the topmost tile-row.*

(c) *Update the rest of the matrix with all row operations made in the panel.*

(d) *Execute the row exchanges on the left side of the panel.*

Figure 5: *One iteration of the LU-decomposition on a tiled matrix ($k = 3$ on a matrix with 8×8 tiles).*

asymptotic complexity of $\mathcal{O}(b^3)$ flops. All T-tasks combined have a complexity of

$$\textit{complexity(T-tasks)} = \frac{n \cdot (n-1)}{2} \cdot \mathcal{O}\left(b^3\right) = \mathcal{O}\left(n^2 \cdot b^3\right).$$

SUBMATRIX_UPDATE-tasks (or *U-tasks*): These are the tasks that update most of the matrix. In every iteration, there is one U-task for each tile that has to be updated (see Figure 5c). Every U-task operates on one tile. It computes the effects of row operations – made within the panel – on its tile. This is done with a simple matrix multiplication. The U-task operating on tile A_{ij} during iteration k computes $A_{ij} = A_{ij} - A_{ik} \cdot A_{kj}$. This is very similar to the matrix update of the Gaussian Elimination seen in Algorithm 1 (line 6-8) and Figure 4c. The difference is that matrix-matrix products are significantly better to use a processor at full capacity than vector-matrix products. One U-task consists of $2b^3$ flops. All U-tasks together have a complexity of

$$\textit{complexity(U-tasks)} = \frac{n \cdot (n-1) \cdot (2n-1)}{6} \cdot 2b^3 \ .$$

BEHIND_PANEL_PERMUTATION-tasks (or *X-tasks*): An X-task executes row exchanges made within the panel on a tile-column left of the panel (see Figure 5d). In each iteration, there are as many X-tasks as there are tile-columns

left of the panel. X-tasks fulfill a special role in the algorithm. They do not compute any flops. Because of this, it is beneficial to omit X-tasks from certain considerations or images. Instead of computing any values X-tasks exchange values from the topmost matrix tile with values from the corresponding pivot rows. Therefore, they cause $\mathcal{O}(b^2)$ reads/writes without having any computational complexity.

In Conclusion U-tasks are the only class of tasks which has a cubic complexity (cubic in the matrix size n_{glob}). They dominate the running time of the algorithm. Therefore, we will be paying a lot of attention to optimizing the execution of U-tasks. The LU-decomposition with this algorithm takes

$$\begin{aligned} \textit{complexity(tiled algorithm)} &= \frac{n \cdot (n-1) \cdot (2n-1)}{6} \cdot 2b^3 + \mathcal{O}\Big(n^2 b^3\Big) \\ &= \frac{2}{3} n^3 b^3 - \frac{1}{2} n^2 b^3 + \frac{1}{6} n b^3 + \mathcal{O}\Big(n^2 b^3\Big) \quad . \end{aligned}$$

The theoretical flop count taken from LAPACK working note 41 [4] (with $m = n$ for square matrices) is

$$\textit{complexity(LU-decomposition)} = \frac{2}{3} n_{glob}^3 - \frac{1}{2} n_{glob}^2 + \frac{5}{6} n_{glob} \quad .$$

This is also the flop count that we use to compute the performance (Gflop/s) of our algorithm in Section 6.

2.2.4 Spatial View on the Tiled LU-decomposition

In this section, we want to introduce a geometrical view on all subtasks of the numerical algorithm for tiled matrices. We identify each subtask with one tile that it operates on (even if it changes multiple tiles). This allows us to spatially arrange subtasks in a grid (see Figure 6). This grid structure simplifies the formulation of dependencies between subtasks.

At first we introduce new version numbers for all tiles of the matrix. We call the tile at position (i, j) after iteration k of the algorithm $A_{ij}^{(k)}$. With each iteration we finish the k-th tile row. This means that $A_{ij}^{(i)}$ is the final version of tile A_{ij} ($A_{ij}^{(i)} = A_{ij}^{(n)}$) because in future iterations there are no tasks that change A_{ij}.

With the definition of version numbers, we can introduce a new naming system to easily differentiate all subtasks of the algorithm. Each task is indexed with the global iteration that it belongs to and with the position of one matrix tile that it works on (See Figure 6):

- $P_{kk}^{(k)}$ is the P-task of iteration k.
- $T_{kj}^{(k)}$ is the T-task that operates on column j during iteration k.
- $U_{ij}^{(k)}$ is the U-task that operates on the tile A_{ij} during iteration k.
- $X_{kj}^{(k)}$ is the X-task that operates on column j during iteration k.

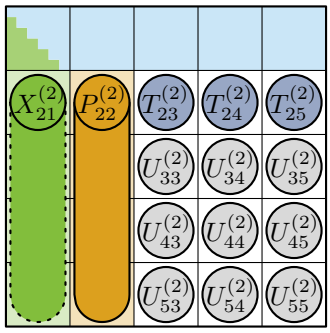


Figure 6: *Task naming example ($k = 2$ on 5×5 matrix)*

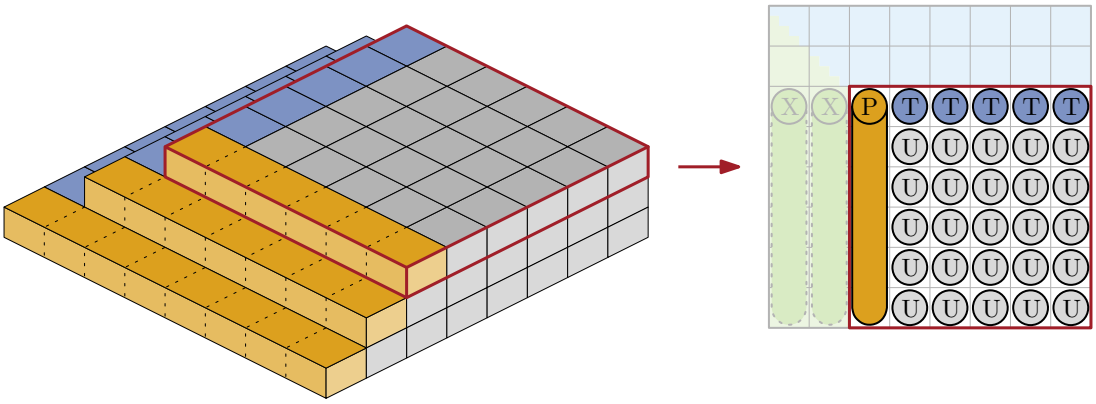


Figure 7: *Spatial representation of subtasks from three iterations of the tiled LU-algorithm (without X-tasks).*

If we start working on a square matrix then the part of the matrix that is actively worked on (through P-, T-, and U-tasks) remains a square at the bottom right corner of the matrix (each iteration finishes one tile-row and one tile-column). We can arrange all tasks of one iteration according to the matrix tiles that the task operates on (see task indices), an example of this can be seen in Figure 6. Additionally, we can stack the tasks of consecutive iterations on top of each other – creating a time axis. This generates a pyramid structure of tasks, part of which can be seen in Figure 7.

In this structure, the task occupying position (i, j, k) is responsible for the computation of tile $A_{ij}^{(k)}$. This view on the algorithm's subtasks is beneficial because we can use it to describe relations between subtasks more easily.

2.3 DAG-Scheduling

DAG-Scheduling is a very important computation model to expose opportunities for parallelization within an algorithm. An algorithm is divided into a set V of subtasks $(t_1 ... t_{|V|})$. Where each subtask has to be executed sequentially.

Some of these subtasks might be independent from one another. Hence, they may be scheduled simultaneously. Other tasks might not be independent, for example if task t_b uses task t_a's result then t_b cannot be scheduled until t_a is finished. We call this a *precedence constraint* or we say t_b depends on t_a. Such precedence constraints can be denoted by tuples (t_a, t_b). We say E is the set of all precedence constraints which are implicitly given by the algorithm ($E \subset V \times V$).

We can visualize all tasks and dependencies as a directed graph $G = (V, E)$. The vertices of the graph are the subtasks of the algorithm, and the edges of the graph are the dependencies between subtasks. In the example above t_b depends on t_a. Therefore, the graph contains an edge from vertex t_a to vertex t_b. Every valid algorithm that is parallelized through subtasks implies such a graph. There cannot be any directed cycles in this graph because a cycle would imply a task that transitively depends on itself (This task could never be ready for execution). Thus, we call this graph the *task-DAG* (Directed Acyclic Graph) of an algorithm.

There are four basic states a task can have during the computation:

- (not ready) A task is not ready as long as it depends on at least one unfinished task.
- (ready) Ready tasks are ready to be scheduled and executed. A task is ready if it has no dependency to an unfinished task and it is not already working or finished.
- (working) This means that the task is currently being executed. Each task t has a running time $w(t)$.
- (finished) Finished tasks are tasks that have already been executed.

In the beginning of an algorithm's execution, there are usually only a few ready tasks and many more tasks that are not ready. Once a ready task is finished other dependent tasks may become ready. These ready tasks are also scheduled until every task is finished. An execution of the algorithm can be compared to a traversal of the task-DAG. The target of DAG-scheduling is usually to find a schedule that minimizes the time needed for the computation.

2.4 Task-DAG of the LU-decomposition

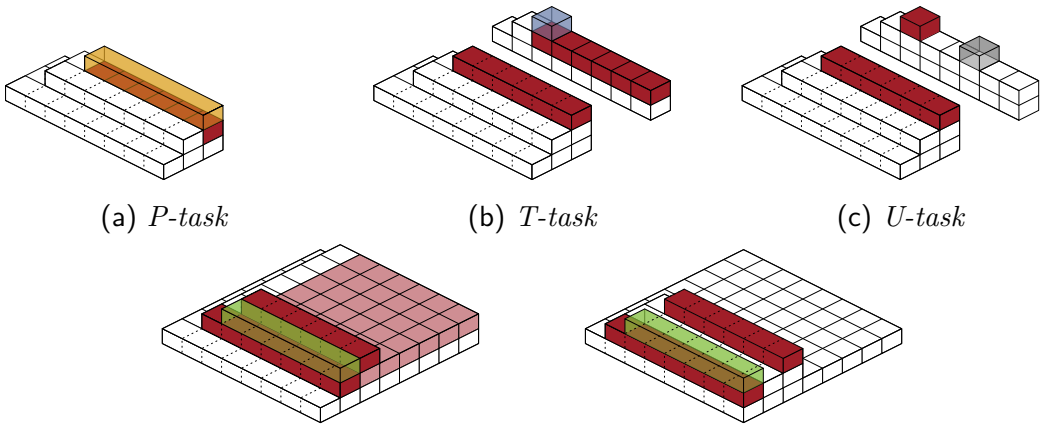
The algorithm from Section 2.2.3 is already separated in small subtasks which we indexed in Section 2.2.4. Now, we analyze the precedence constraints between these subtasks to see which of them can be executed in parallel. A summary of all precedence constraints can be seen in Table 1 and a visualization can be seen in Figure 8:

P-task: (see Figure 8a) During each iteration $P_{kk}^{(k)}$ works on the panel$(A_{k\dots n\,k})$. It reads tiles $A_{k\dots n\,k}^{(k-1)}$ and transforms them to version $A_{k\dots n\,k}^{(k)}$. The tiles $A_{k\dots n\,k}^{(k-1)}$ are usually written by the tasks $U_{k\dots n\,k}^{(k-1)}$ (except for $k = 1$). Therefore, $P_{kk}^{(k)}$ cannot be started until $U_{k\dots n\,k}^{(k-1)}$ are finished. $P_{kk}^{(k)}$ also generates the k-th part of the pivoting vector (Q_k).

T-task: (see Figure 8b) The T-task $T_{kj}^{(k)}$ performs row exchanges on column j. Therefore, it reads tiles $A_{k\dots n\,j}^{(k-1)}$ and the k-th part of the pivot vector Q_k. It changes the tiles $A_{k\dots n\,j}^{(k-1)}$ into intermediate tile versions $\widetilde{A}_{k\dots n\,j}^{(k-1)}$ where permutations are already performed but row operations are not. Row operations of all previous iterations have to be completed to do this, therefore, $T_{kj}^{(k)}$ depends on the tasks $U_{k\dots n\,j}^{(k-1)}$. After all row permutations, $T_{kj}^{(k)}$ performs the row operations on all rows of the uppermost tile $\widetilde{A}_{kj}^{(k-1)}$ to create tile $A_{kj}^{(k)}$. Hence, $T_{kj}^{(k)}$ needs tile $A_{kk}^{(k)}$ which

Task t	Inputs IN(t) Dependencies In	Outputs OUT(t) Dependencies Out
$P_{kk}^{(k)}$	$A_{k\ldots n\,k}^{(k-1)}$ $U_{k\ldots n\,k}^{(k-1)}$	$A_{k\ldots n\,k}^{(k)}, Q_k$ $T_{k\,k+1\ldots n}^{(k)}, X_{k\,0\ldots k-1}^{(k)}, U_{**}^{(k)}$
$T_{kj}^{(k)}$	$A_{k\ldots n\,j}^{(k-1)}, A_{kk}^{(k)}, Q_k$ $U_{k\ldots n\,j}^{(k-1)}, P_{kk}^{(k)}$	$\widetilde{A}_{kj}^{(k)}, \widetilde{A}_{k+1\ldots n\,j}^{(k-1)}$ $U_{k+1\ldots n\,j}^{(k)}$
$U_{ij}^{(k)}$	$\widetilde{A}_{ij}^{(k-1)}, A_{ik}^{(k)}, A_{kj}^{(k)}$ $P_{kk}^{(k)}, T_{kj}^{(k)}$	$A_{ij}^{(k)}$ $[X_{k+1\,k}^{(k+1)}]$, $P_{k+1\,k+1}^{(k+1)}$ or $T_{k+1\,j}^{(k+1)}$
$X_{kj}^{(k)}$	$A_{k\ldots n\,j}^{(k-1)}, Q_k$ $P_{kk}^{(k)}$, $X_{k-1\,j}^{(k-1)}$ or $[U_{**}^{(k-1)}]$	$A_{k\ldots n\,j}^{(k)}$ $X_{k+1\,j}^{(k+1)}$

Table 1: *Tasks with corresponding input and output tiles, and the resulting dependencies. Square brackets* $[*]$ *denote read before write dependencies.*



(a) *P-task* (b) *T-task* (c) *U-task*

(d) *X-task with* $j = k - 1$ (e) *X-task with* $j < k - 1$

Figure 8: *Incoming dependencies of each class of tasks. The examined task is drawn transparent and highlighted in its color (see Figure 5). Direct dependencies are highlighted in red (read before write in brighter red). All white tasks are indirect dependencies.*

at this point was last written by $P_{kk}^{(k)}$ (as was Q_k which was needed for the row permutations). Therefore, $T_{kj}^{(k)}$ is also dependent on $P_{kk}^{(k)}$.

U-task: (see Figure 8c) The U-task $U_{ij}^{(k)}$ performs all row operations of the k-th iteration on tile $\widetilde{A}_{ij}^{(k-1)}$. Therefore, it reads the tiles $\widetilde{A}_{ij}^{(k-1)}$, $A_{ik}^{(k)}$ (for the elimination factors) and $A_{kj}^{(k)}$ (for the pivot row elements). This makes $U_{ij}^{(k)}$ dependent on $T_{kj}^{(k)}$ (who writes $\widetilde{A}_{ij}^{(k-1)}$ and $A_{kj}^{(k)}$), and on $P_{kk}^{(k)}$ (who writes $A_{ik}^{(k)}$). The dependence on $P_{kk}^{(k)}$ can be ignored, since $U_{ij}^{(k)}$ is already dependent on $T_{kj}^{(k)}$ which in turn is also dependent on $P_{kk}^{(k)}$.

X-task: (see Figure 8d and 8e) The X-task $X_{kj}^{(k)}$ performs row interchanges on column j (left of the panel). Thus, it reads the permutation vector Q_k which makes it dependent on $P_{kk}^{(k)}$. It also reads the tiles $A_{k\ldots n\,j}^{(k-1)}$ and changes them to

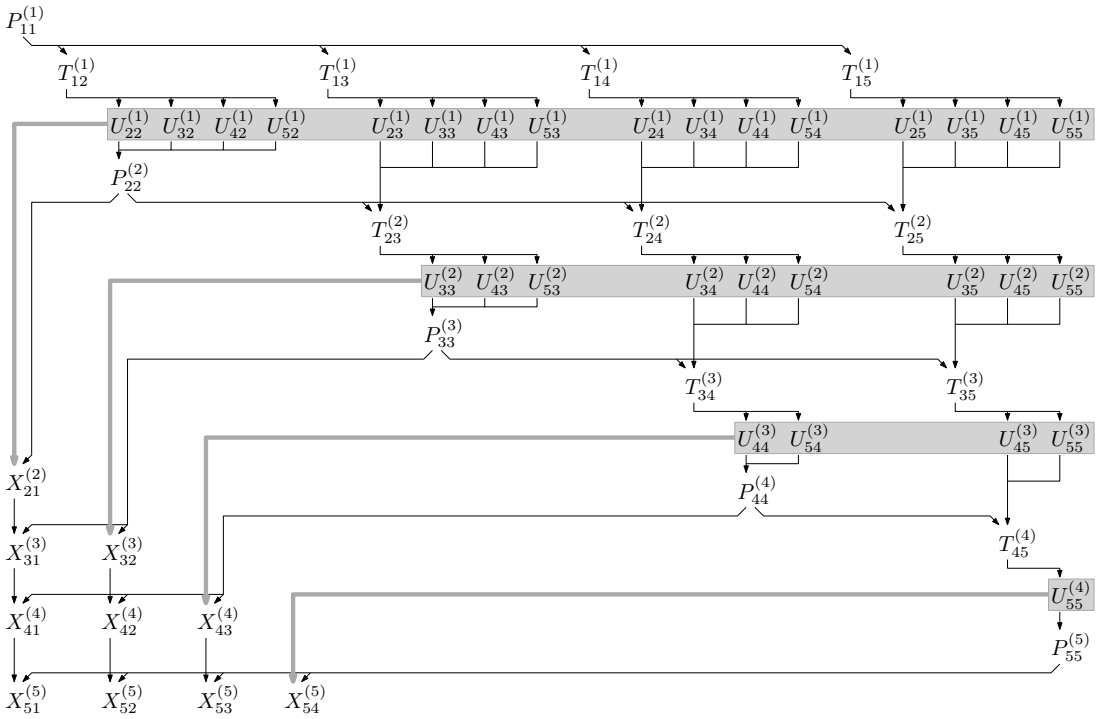


Figure 9: *task-DAG of the tile based algorithm on a* 5×5 *tile matrix.*

version k. These tiles were previously written by either task $P_{k-1\,k-1}^{(k-1)}$ (if $j = k-1$ this means $A_{k\ldots n\,j}^{(k-1)}$ is the previous panel) or task $X_{k-1\,j}^{(k-1)}$ (if $j < k-1$).

If $A_{k\ldots n\,j}^{(k-1)}$ was written by $P_{k-1\,j}^{(k-1)}$, then it was part of the panel from the last iteration $(k-1)$ which means many tasks of that iteration read its contents. Therefore, we have to make sure that each of those tasks is finished before $X_{kj}^{(k)}$ can change the panel. Thus, $X_{kj}^{(k)}$ is dependent on each U-task of the previous iteration $(k-1)$. We call this kind of precedence constraint a *write after read constraint*.

Write after read constraints can generally be circumvented by duplicating the affected memory. This is not important here since X-tasks only make up an insignificant amount of the algorithm's running time. Hence, it is no problem to wait for all dependencies and postpone X-tasks until later in the computation. This is the only instance of a write after read precedence constraint within the algorithm.

With the help of these precedence constraints we can build the task-DAG (see Figure 9). One can see that there is a lot of structure to this graph. This makes it easy to pre-compute the incoming and outgoing dependencies when the subtasks are created.

2.5 Modern Multi-Socket Machines

Modern processors become increasingly complicated. They get more cores, deeper cache hierarchies, and they have more advanced arithmetic units. Most of our experiments were done on a Sandy Bridge system, where each core can compute up to four double precision multiplications and four double precision additions per cycle (AVX). Newer machines, that support AVX2 instructions, can even com-

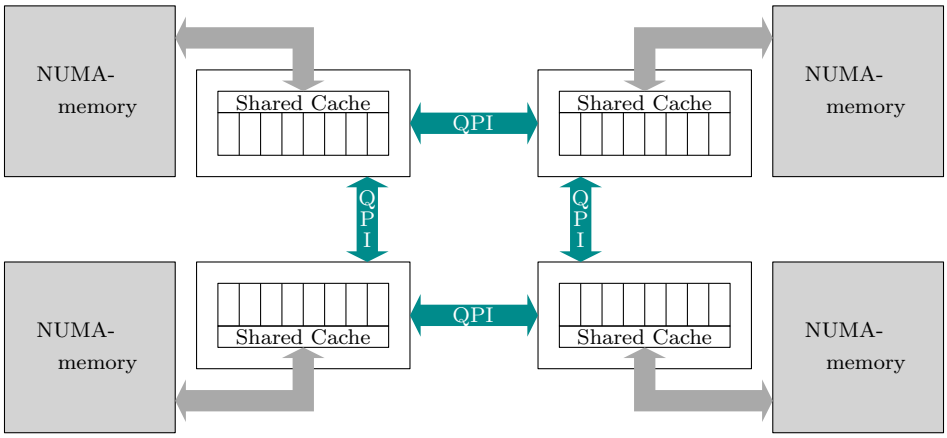


Figure 10: *Schematic sketch of a four socket architecture. It shows the NUMA-memory, shared caches, as well as QPI-links connecting the sockets.*

pute eight combined multiply and add functions per cycle, resulting in a peak performance of 16 flop per cycle. Luckily modern processors are also equipped with bigger and bigger caches to hide long latencies when accessing slow off-core memory.

Properly using these big caches is becoming an important issue for programmers. Especially the last level of on-chip cache (L3) is crucial for the performance of many algorithms because it is the largest cache that can hide long load times from off-chip memory. Since the L3-cache is commonly shared between all cores of one processor, we can decrease the number of L3-cache-misses by reusing cache-lines that were originally loaded by another core of the same processor. This makes the L3-cache even more interesting from an algorithmic point of view. We call all cores sharing one L3-cache a cache-node.

Another aspect of modern multi-socket machines is that they are usually *NUMA*-machines (Non Uniform Memory Access). This means that there are different memory sections and the access rate of each memory section depends on the processor that it is accessed from. In practice, every socket has its own "local" memory. Local memories are shared between sockets, but accesses are significantly faster when accessing local memory. Everything that is accessed remotely has to be shared through *QPI* links (QuickPath Interconnect) connecting all sockets (see Figure 10).

NUMA-machines accentuate the importance of caches even further because there can be longer latencies when accessing memory remotely. Memory controllers are usually shared between all cores of one processor. Thus, cache-nodes and NUMA-nodes usually correspond with each other. Therefore, we use the terms node, cache-node, and NUMA-node somewhat interchangeably.

In the following, we will also use the terms thread and core interchangeably. At the beginning of the execution, we start one thread per core. Every thread is pinned to one corresponding core. Even though every machine that we used supports hyper-threading, we create only one thread per physical core. Hyper-threading does not actually improve the performance of the LU-decomposition since the two hardware-threads that are executed on the same core share a common floating-point unit.

3 Our Concepts for Performance Improvements

In this section, we present the ideas and concepts that we used to optimize the scheduling of subtasks. As described in Section 2.2.3, U-tasks represent the most work among all task classes. They are the only task class that performs $\Theta\left(n_{glob}^3\right)$ floating point operations. In our experiments, U-tasks take up to 95% of the running time especially on large matrices (see Section 6.4.1). Therefore, we concentrate our efforts on improving the performance of U-tasks.

At first we will describe how the scheduling of subtasks can influence the cache status, and thereby, the performance of a subtask. To do this we first define a cache model that will simplify the cache behavior during the computation (Section 3.1). Then we point out why classical DAG-scheduling (Section 3.2) might not be the best option to schedule the LU-decomposition. After this introduction, we go into further detail on our approach to improve the cache performance (Section 3.3) and NUMA-behavior (Section 3.4) of our implementation.

3.1 Tile-Cache Model

To be able to predict the cache performance of a potential schedule, we developed the following theoretical machine model. It is supposed to replicate the conditions described in Section 2.5 while still being easy to work with.

For our purposes a modern multi socket machine consists of s nodes ($node_0, \ldots, node_{s-1}$). Each node consists of p cores, a shared tile-cache, and local memory. The cores are labeled globally, $core_0, \ldots, core_{p-1}$ are the cores of $node_0$ (subsequently $node_i$ consists of $core_{i \cdot p}, \ldots core_{(i+1) \cdot p-1}$). Each node's shared cache can contain a constant number c of matrix tiles (see Section 2.1 Blocked Data layout) depending on the tile size ($b \times b$) and the physical cache size.

When a task t is executed on $core_i$ all input tiles $A_{in} \in \mathrm{IN}(t)$ have to be read. There are three different memory locations where A_{in} could be read from:

- (Cache Hit) If tile A_{in} is already loaded into the shared cache of the corresponding node, then it can be read from this cache reducing the loading time significantly.
- (Locally Solvable Cache Miss) The tile is not cached and has to be read from the local memory.
- (NUMA-Miss) A_{in} is not cached and is not stored locally thus it has to be loaded from another node's local memory. We call this a NUMA-miss.

Every tile, that is read this way, is stored in $core_i$'s shared cache. After the execution of t, each output tile $A_{out} \in \mathrm{OUT}(t)$ replaces the corresponding input tile in the local cache. Eventual copies of the old version in other caches are invalidated (removed).

T- and X-tasks are the only tasks that read partial tiles. Therefore their input tiles have to be treated differently. T- and X-tasks perform row exchanges on non-panel tiles. A T-task $T_{kj}^{(k)}$ swaps rows of the tile-column $A_{k \ldots n\, j}$ into its uppermost tile A_{kj}. Therefore, it reads/changes A_{kj} completely and only reads/changes single rows of all other tiles within the column. Hence, we approximate that for cache modeling purposes a T-task (or X-task) only reads and writes the topmost

tile of its column. To represent the additional data that is read from the rest of the column, we load another tile into the local tile cache. This tile is not part of the actual matrix. It is loaded into the cache to "waste" space, therefore, it cannot be used for future cache hits. Since all other tiles of the column are not changed significantly, they are not invalidated or cached.

Each local cache can store up to c different tiles. When the tile A_{in} is loaded and the local cache is already at its capacity, an old tile will be replaced. In our model we replace the tile that has not been used for the longest time.

This cache model is meant to simulate the behavior of modern L3-caches during the computation of the LU-decomposition. There are some key differences between our model and actual hardware caches. The biggest difference is that the model only represents complete matrix tiles. In reality a tile consists of thousands of different cache lines. We think that this simplification is justifiable because most subtasks accesses their input tiles as a whole. This means that their cache lines enter the cache succinctly and are probably also replaced somewhat succinctly.

The simulation assumes a fully associative "least recently used" cache (the oldest tile of the cache is replaced) when in reality caches are not fully associative. This means, that a new cache line can only replace a cache line from a subset of all cache lines. Hence, it might not always be the oldest cache line which is replaced. Cache lines might even be evicted although the cache is not full yet. This happens if the corresponding subset of the cache is full.

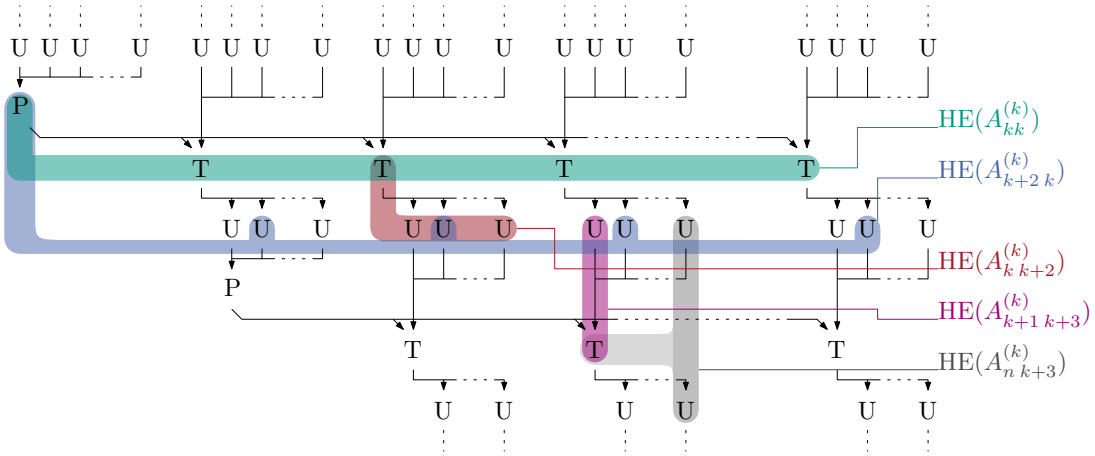
We think that in our case the assumption of a fully associative cache is valid. The machine, that we used for testing (see Section 6.1 SandyBridge32), has a 20-way set associative cache. This means, the cache can simultaneously store up to 20 cache lines associated with one *cache group*. If we assume that no two cache lines of one matrix tile share the same cache group (This assumption is guaranteed if a tile is stored consecutively within the physical memory) then there can be at least 20 different tiles in the shared cache of one node and replacement should work as expected.

Our model only counts the number of "tile-cache hits" (and misses). We do this because reading one tile produces significantly less cache misses than the number of cache lines that the tile consists of. Therefore, we believe that in practice a lot of work is done by the prefetcher. We also assume that a matrix tile is big in comparison to all other data which is loaded during the execution of a task. Therefore, we ignore all data that is not part of the matrix in our cache simulation. Reloading parts of other data structures should not evacuate significant parts of a tile from the cache.

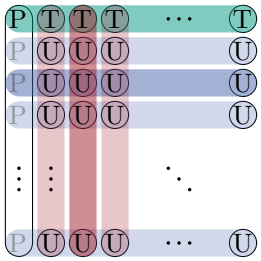
This cache model simplifies the complex behaviors of modern hardware caches good enough to predict the effectiveness of our following optimization techniques. In Section 6.5, we show that it also holds up in real experiments and measurements.

3.2 Data co-use Hypergraph

One major problem with using classical DAG-scheduling for the LU-decomposition is that all relations between subtasks are expressed through precedence constraints. Two tasks that are not directly or indirectly dependent have no con-



(a) *Some examples of hyperedges drawn into a section of the task-DAG (consisting of two iterations and no X-tasks).*



(b) *Hyperedges connecting tasks from one iteration. The different types of hyperedges (panel tile, tile from topmost tile row) are color coded as in Figure 11a. Bright edges do not reappear in Figure 11a.*

Figure 11: *Data co-use hyperedges.*

nection. Therefore, they are scheduled completely independent from one another even if they share common input tiles. This makes expressing data reuse between different tasks difficult. But since our subtask implementation basically uses the same code as our competitors (mostly BLAS library calls), cleverly scheduling different subtasks becomes the only way to improve running times.

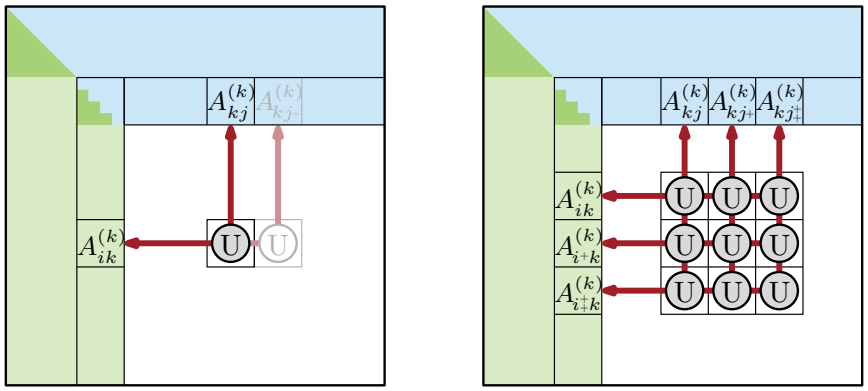
To improve the global schedule, we analyze not only the obvious precedence constraints of the task-DAG. We also review data reuse possibilities. Therefore, we introduce new *hyperedges* (edges connecting more than two tasks) between tasks that read (write) the same version of a memory section.

$$\mathrm{HE}(mem^{(v)}) = \{t_i \in V \mid \textit{task } t_i \textit{ reads or writes into } mem^{(v)}\}$$

In our case the memory sections correspond to matrix tiles. Each version of a matrix tile $A^{(k)}_{ij}$ implicitly defines a hyperedge $\mathrm{HE}(A^{(k)}_{ij})$ which contains all subtasks that read (or write) tile $A^{(k)}_{ij}$.

$$\mathrm{HE}(A^{(k)}_{ij}) = \{t_i \in V \mid \textit{task } t_i \textit{ reads or writes into tile } A^{(k)}_{ij}\}$$

In Figure 11 we show examples of hyperedges within the task-DAG (Figure 11a) and within the spatial representation of one iteration (Figure 11b). One can easily see that each hyperedge $\mathrm{HE}(A^{(k)}_{ij})$ contains exactly one task that writes $A^{(k)}_{ij}$. All other tasks read $A^{(k)}_{ij}$. Each precedence constraint edge (t_a, t_b)



(a) *One U-task and its input tiles (the neighboring U-task shares an input tile).*

(b) *Square group of nine U-tasks sharing multiple input tiles.*

Figure 12: *U-tasks sharing input tiles.*

can be found within such a hyperedge. The constraint is caused by t_b using the results of t_a. In the context of our computation results are matrix tiles. Therefore, the constraint is caused by a tile which t_a writes to that t_b reads. But not all connections of such a hyperedge follow precedence constraints.

If we look at hyperedge HE($A_{k+1\,k}^{(k)}$) (the second panel tile of iteration k), the hyperedge contains all tasks that handle $A_{k+1\,k}^{(k)}$ namely $P_{kk}^{(k)}$ (the panel task), $U_{k+1\,(k+1)\ldots n}^{(k)}$ (the topmost row of U-tasks), and $X_{k+1\,k}^{(k+1)}$. The tasks $U_{k+1\,(k+1)\ldots n}^{(k)}$ have no connection in the original task-DAG. Hence, a classical DAG-scheduler would not know that all these tasks share an input tile, let alone schedule them appropriately.

Our target is to schedule tasks in such a way that these data reuse possibilities are exploited. Scheduling multiple tasks of one data reuse hyperedge appropriately can often lead to an improved cache performance and even reduce the amount of node to node communication.

3.3 Cache Improvements

When a core works on a U-task $U_{ij}^{(k)}$, there are three input tiles that have to be loaded into the local cache: the topmost tile of the tile column $A_{kj}^{(k)}$, the panel tile $A_{ik}^{(k)}$, and the original tile $\tilde{A}_{ij}^{(k-1)}$ (see Figure 12a). If the same core would also compute the neighboring tile update $U_{i\,j+1}^{(k)}$, $A_{ik}^{(k)}$ might still be in the cache. This produces a cache hit in our machine model (see Section 3.1), thus, there would only be the need to load two new matrix tiles.

As described in Section 3.1, caches are shared between all cores of one node. Therefore, we can also reduce the amount of tile loads by scheduling neighboring tasks simultaneously on two cores of the same node. This can be even more beneficial because it reduces the amount of tiles that have to be cached simultaneously. If we schedule all U-tasks that operate on one tile-row i to a node, then we only need to store two tiles per core plus the shared tile $A_{ik}^{(k)}$. Thus, tiles can remain inside the cache longer because less tiles are actively needed for the

computation.

With that said, we need to figure out how to group and schedule U-tasks such that we reuse as many cached tiles as possible. To do this we analyze the data reuse hyperedges described in Section 3.2. The hyperedges between U-tasks of one iteration k have a grid structure (see Figure 11b). All U-tasks working on one column j are connected through the common use of the topmost tile $A_{kj}^{(k)}$ ($U_{(k+1)\ldots n\,j}^{(k)} \in \mathrm{HE}(A_{kj}^{(k)})$), and all U-tasks operating on the same row i are connected through the common use of the same panel tile $A_{ik}^{(k)}$ ($U_{i\,(k+1)\ldots n}^{(k)} \in \mathrm{HE}(A_{ik}^{(k)})$).

Therefore, we propose to group tasks that operate on a square of tiles. This exploits horizontal and vertical data reuse opportunities equally (see Figure 12b). If we group all U-tasks working on a 3×3 square of tiles, then computing all nine U-tasks in parallel loads a total of 15 distinct tiles (three panel tiles, three tiles of the first row, and the nine tiles that are operated on). This way, we can save twelve loaded tiles compared to unoptimized execution (three tiles per U-task = 27 loaded tiles) and four loaded tiles compared to row/column-wise groups (two tile loads per task plus one commonly used tile = 19 loaded tiles). The savings increase even more when we increase the number of tile updates that are grouped together ($4 \times 4 \Rightarrow 24$ tile loads $\Rightarrow 24$ tiles saved).

There are upper limits to the size that groups of U-tasks should have. If a group is too big, cached tiles might be replaced while the node still works on the same group. In this case cache benefits might be lost because tiles are replaced before they can be reused.

3.4 Targeted NUMA-Scheduling

In Section 3.3, we introduce a possibility to reduce the number of tile loads throughout our matrix update tasks. Our next goal is to accelerate the tile loads that are still needed. As described in Section 3.1, there are three different memory locations a read tile can come from, the cache, the local memory node, and a foreign node. We want to make sure that our implementation of the LU-decomposition loads as few tiles as possible from foreign nodes. At first we want to describe what parameters we can influence to achieve this.

The first influence that we have on the NUMA-performance is that we can influence the way in which the matrix is distributed on different nodes of the machine. We store the matrix separated in tiles with a blocked data layout (see Section 2.1). Each tile is stored on one node in a consecutive area of memory. To be able to improve the locality of data reads, we can control the distribution of the matrix tiles to nodes. This distribution can be described as a mapping.

$$t2n : \mathit{Tiles} \rightarrow |\mathit{Nodes}|$$

We chose to keep this mapping constant within one computation. This means that we will not move tiles from one node to another. Moving a tile during the computation would require copying an existing tile. We believe that the time needed for this copy is greater than the time that can be saved through NUMA-hits on this tile. Therefore, all different versions of one tile will always remain on the same node (at the same global position).

The second influence that we have on the NUMA-behavior of our implementation is the placement of subtasks. Our implementation will work on each core of the machine in parallel. Therefore, we can control which core of the machine executes a certain subtask. This can be described as a mapping comparable to t2n. Note, that we do only map to nodes (not to specific cores). The specific core does not matter from a NUMA standpoint and this mapping leaves us more freedom for the specific scheduling.

$$s2n : Subtasks \rightarrow |Nodes|$$

If we schedule a subtask on the node that contains all its input tiles, then that subtask will not lead to any NUMA-misses. It is not possible to place all matrix tiles such that all input tiles of each U-task are located on the same node.

Instead, we have to optimize t2n and s2n together to achieve the best possible NUMA-behavior. To do this, we look at the different tiles that are accessed by each task. What sticks out to us is that all tasks access multiple tiles from one tile column. P-tasks access the whole panel, T- and X-tasks exchange lines within one tile column, and even U-tasks access two tiles from one column ($U_{ij}^{(k)}$ accesses A_{ij} as well as A_{kj}).

Therefore, we propose to distribute the matrix tile-column-wise. This means that all tiles of one tile-column are stored in the local memory of the same node. Subsequently, subtasks are scheduled to be executed on a core that belongs to the node where most of their input tiles are stored.

3.5 Merging Cache and NUMA-Strategies

On first glance our cache and NUMA-strategies work against each other. In Section 3.4 we proposed to store the matrix distributed tile-column-wise and to schedule subtasks on the node that holds their input tiles. For our proposed cache optimizations, introduced in Section 3.3, we suggest to execute square groups of tasks on one node to exploit opportunities for cache co-use.

If we distribute the matrix in a round robin tile-column-wise pattern, the two tasks $U_{ij}^{(k)}$ and $U_{i\,j+1}^{(k)}$ are supposed to be scheduled on two different nodes (according to the NUMA-optimization). For cache purposes they are two neighboring tasks which could benefit from being scheduled on one node (simultaneously).

Another mistake would be to partition the matrix into $|Nodes|$ vertical sections. This would cause an imbalanced work placement. With each iteration the area of the matrix that is worked on by P-, T-, and U-tasks shrinks by one tile-column. Hence, the node that is responsible for the leftmost section will have less tasks to execute (apart from X-tasks, which do not perform any floating point operations).

Our solution to this problem is to distribute m neighboring tile-columns to one node. This allows us to group tasks that work on $m \times m$ squares of tiles for "simultaneous" execution on one node. It also roughly balances the work load of different nodes. To further simplify the grouping of tasks for cache purposes, we introduce *meta-tiles*. Meta-tiles are $m \times m$ squares of tiles. In our implementation we group tasks depending on the meta-tile that they operate on (see Section 4.1.2).

The most obvious way to combine tiles to meta-tiles is to combine the $m \times m$ tiles on the upper left side of the matrix and to continue from there. This is really easy if n is a multiple of m, because then every meta-tile has the same size. But if this is not the case, then tiling the matrix from the upper left side will lead to cutoffs in the last row/column of meta-tasks (cutoffs = smaller meta-tiles; see Figure 13a).

As described in Section 2.2.4, the area of the matrix that is actively being worked on is a shrinking square on the bottom right side of the matrix. Each iteration shrinks this square by one tile row/column. Therefore, the top left border of this square can cut across meta-tiles (see Figure 13a). Hence, there can also be cutoffs on the top left side of the square. So instead of creating meta-tiles from the top left side of the matrix, we start tiling the matrix from the the bottom right corner of the matrix (see Figure 13b). This reduces the absolute number of possible borders with cutoffs (there can never be cutoffs on the bottom/right side of the matrix).

With these considerations we can define t2n and s2n as:

$$\begin{aligned} & \mathit{off} && = m - (n \bmod m) \\ t2n: \quad & A_{ij} && \mapsto ((j + \mathit{off}) \operatorname{div} m) \bmod |\mathit{Nodes}| \\ s2n: \quad & Y_{ij}^{(k)} && \mapsto ((j + \mathit{off}) \operatorname{div} m) \bmod |\mathit{Nodes}| = t2n(A_{ij}) \end{aligned}$$

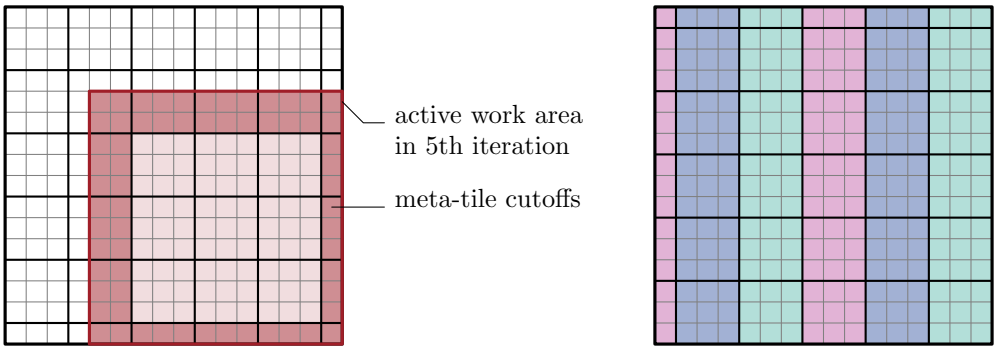
In Section 2.2.3 we describe that P-tasks use a parallelized algorithm. Thus, there can be multiple cores cooperating on one P-task. In this case, we enforce that all cores working on one P-task belong to one node. This creates an upper bound to the number of cores working on one P-task. In Section 5.4 we describe why this restriction is not problematic.

Even with this distribution the workload is not fully balanced between nodes. Therefore, $s2n(t)$ should only be used as the preferred node for task t (no strict allocation). There has to be a system which prioritizes the execution of t on the preferred node but also ensures that no core is idling while there are ready tasks that can be executed. This form of work-stealing or work-rebalancing is important because the time that can be saved through proper NUMA-placement of one task is too small to justify waiting for a more appropriate task while other tasks are ready.

4 Overview of our Solution

Our Scheduler works on a two level hierarchical approach. On the *global level* groups of tasks which we call *meta-tasks* are scheduled to nodes. On the *local level* meta-tasks are unpacked into individual subtasks and then scheduled to cores. This two level approach – with meta-tasks – has a lot of benefits. It reduces creation and scheduling overheads of tasks because we will never handle individual tasks on the global scheduling level. It also simplifies the treatment of parallelized subtasks (P-tasks) and the implementation of our cache improvements described in Section 3.3.

We begin this section by describing the concept and the creation of meta-tasks (in Section 4.1). Then, we describe the architecture of our scheduler (in



(a) *When we start tiling from the top left corner, there are iterations with meta-tile-cutoffs on each side of the matrix.*

(b) *A matrix with 3×3 meta-tiles and NUMA-distribution on three nodes. Tiling from the lower right corner reduces cutoffs.*

Figure 13: *Meta-tiles and NUMA-distribution of the matrix.*

Section 4.2). This scheduler uses methods of DAG-scheduling (see Section 2.3), which we refined with our cache and NUMA-concepts (see Section 3), to schedule tasks.

4.1 Meta-Tasks

The task-generation is done sequentially in the beginning of the execution. One core creates all meta-tasks and inserts them into the data structures of the scheduler. This is done in a way that allows other cores to start working as soon as the first task is fully created (see Section 5.2.1).

The DAG that we schedule during the execution is not exactly the task-DAG that we presented in Section 2.4. On the global scheduling level we work only with meta-tasks. Therefore, we will be scheduling a DAG that has meta-tasks as its vertices.

4.1.1 Meta-Task Definition

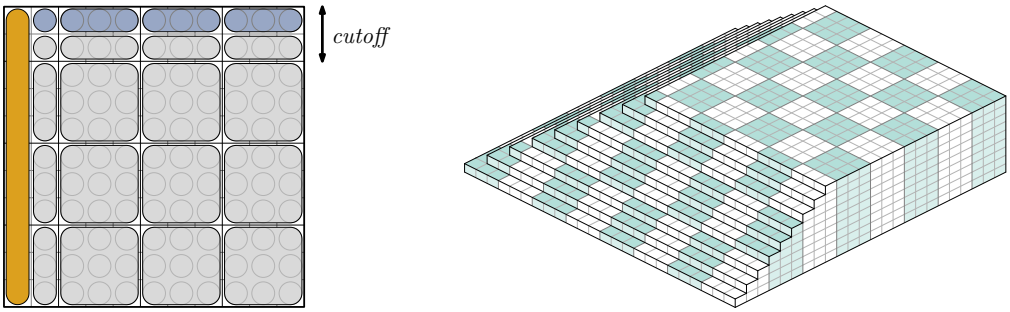
Formally, A meta-task is a set of subtasks. For scheduling purposes, we can extend the definitions of dependency and readiness (see Section 2.3) on meta-tasks. A meta-task mt_2 depends on another meta-task mt_1 if there is a task $t_2 \in mt_2$ that depends on a task $t_1 \in mt_1$. A meta-task is ready to be executed if there are no unfinished meta-tasks that it depends on.

With these preliminary definitions we can define the meta-task-graph. The meta-task-graph is the graph that we get by contracting all tasks belonging to one meta-task within the task-DAG. More formally, it is the graph $\overline{G} = (\overline{V}, \overline{E})$ where $\overline{V}$ is the set of all meta-tasks and $\overline{E}$ is the set of all meta-task dependencies

$$\overline{V} = \{mt_1, \dots mt_\ell\} \qquad mt_i \subset V \qquad V = \dot{\bigcup_{mt \in \overline{V}}} mt$$

$$\overline{E} = \{(mt_1, mt_2) \in \overline{V} \times \overline{V} \mid \exists t_1 \in mt_1,\ t_2 \in mt_2 \quad (t_1, t_2) \in E\}$$

We say a meta-task-graph is schedulable if it has no cycles or (self-)loops. If there is a cycle in the meta-task-graph, then the tasks of this cycle could never



(a) *Combining tasks to meta-tasks with* 3×3 *meta-tiles.*

(b) *Meta-tiles over multiple iterations (colored in a checkerboard pattern)*

Figure 14: *Using meta-tiles to create meta-tasks.*

be ready because they would be (indirectly) dependent on themselves. A bad choice of meta-tasks can lead to an unschedulable meta-task-graph even if the original task-DAG was free of cycles. Cycles can be generated by grouping two tasks when one of them is (indirectly) dependent on the other.

4.1.2 Meta-Task Generation

We construct meta-tasks with a set of rules that are meant to simplify the creation of meta-tasks and keeping track of their dependencies. The most important rules are: we only group tasks of the same task class, and we only group tasks of the same global iteration. These two rules ensure that the resulting meta-task-graph remains a DAG and therefore remains schedulable. In Figure 9 (on page 24) one can easily see why this is the case. All meta-tasks that follow these rules are groups of tasks which lie on one level of the task-DAG. If we contract vertices within one level of the DAG, the resulting graph remains cycle-free.

As motivated in Section 3.3, it is beneficial to group U-tasks that work on a square of tiles. To simplify the grouping of tasks that work on squares of tiles, we introduced *meta-tiles* in Section 3.5. A meta-tile is a square of tiles with dimensions $m \times m$. In each iteration, we group all U-tasks that operate on one meta-tile see Figure 14a (we group T-tasks similarly). The advantage of this approach is that task groups can easily be generated and groups of different iterations work on the same meta-tiles this simplifies dependencies between iterations/task-DAG-levels (see Figure 14b). The resulting meta-task-graph can be seen in Figure 15.

P-tasks There is only one P-task per iteration of the matrix. Therefore, multiple P-tasks will not be grouped together (as it would lead to circular dependencies). But as we mention in Section 2.2.3, our P-task implementation is parallelized (for details see Section 5.4). If a P-task is supposed to be executed with x cores, we will unpack x different subtasks from it. Hence, the P-meta-task can also be viewed as a grouping of x P-task pieces.

U-tasks As described above, we will group U-tasks with the help of meta-tiles. This means that there will usually be groups of m^2 U-tasks.

T-tasks Comparable to U-tasks, T-tasks will also be grouped depending on the meta-tiles that they work on. This leads to groups of m T-tasks building

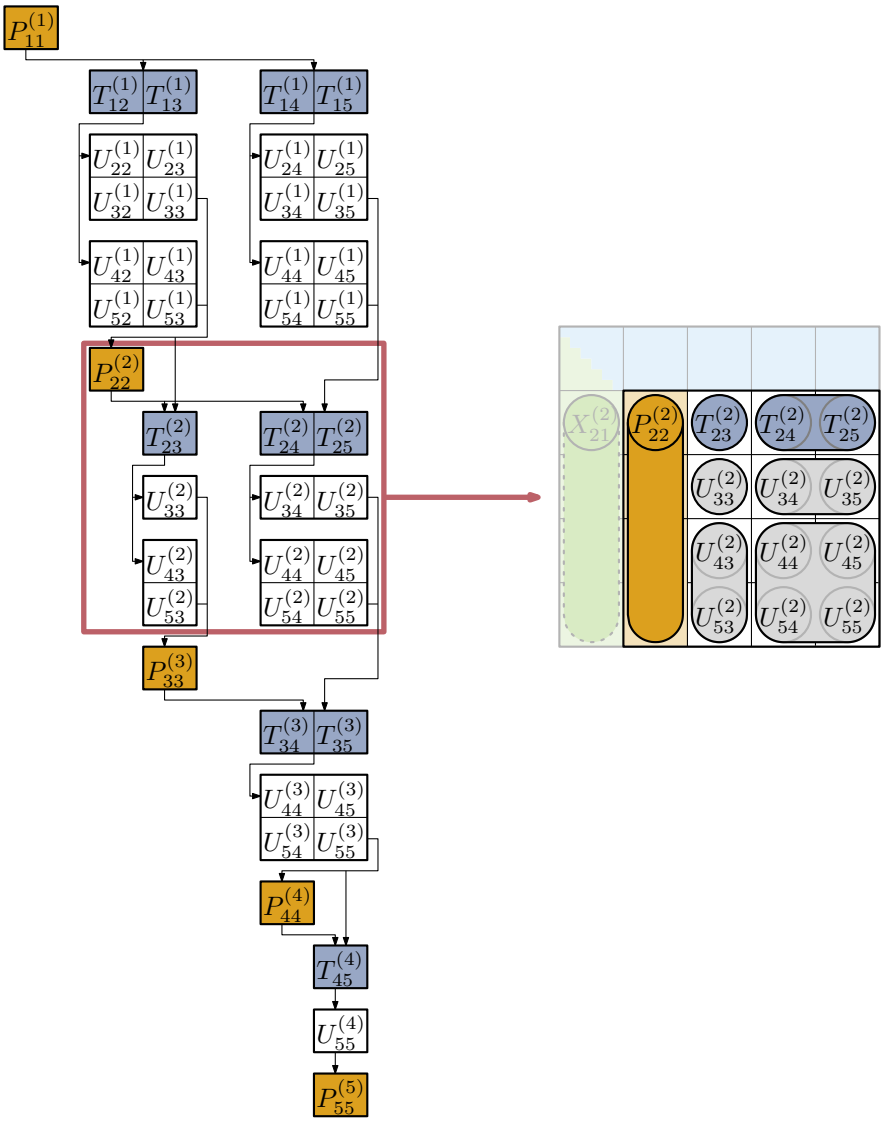


Figure 15: *Meta-Task-Graph. We left out X-tasks because they are not grouped into meta-tasks.*

one T-meta-task. There are two reasons for grouping T-tasks this way. The first reason is that they share a common data dependency towards the panel-tile $A_{kk}^{(k)}$. The second reason is that it simplifies dependencies towards U-meta-tasks (per U-meta-task there is at most one T-task that depends on it, and each U-task depends on exactly one T-task).

X-tasks We do not group X-tasks together into meta-tasks since they only have minor possibilities for data co-use (permutation vector). Additionally they can be used to balance the data between different nodes.

4.1.3 Priorities

Task priorities are very important for parallel computations. We need them to control the order in which tasks are computed. At any given time, there might be considerably more ready tasks than there are cores on the machine. Therefore, we need priorities to specify which tasks should be executed first.

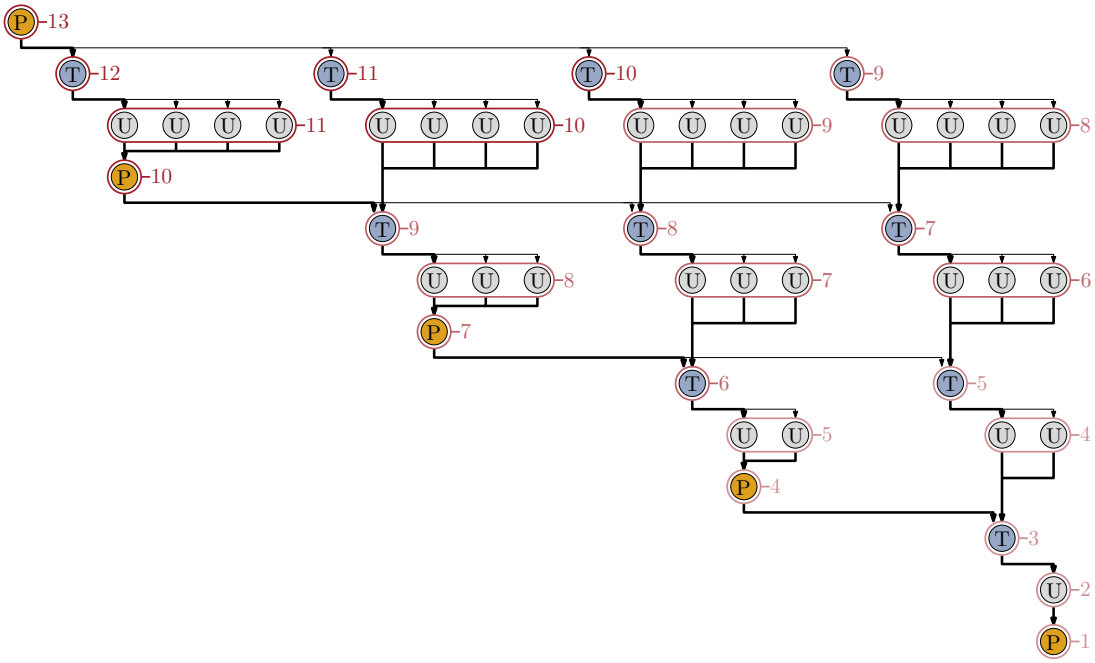


Figure 16: *Task-DAG with priorities. The priorities are marked in shades of red.*

This is important because some tasks are more crucial for the progress of the computation than others. For example, there are tasks that a lot of other tasks depend on. Delaying one of these tasks might lead to a situation were there are less tasks than processors. As one can see in the illustrations of the task-DAG (Figure 9) and the meta-task-graph (Figure 15), every task is at least indirectly dependent on every P-task from earlier iterations. Hence, P-tasks should be executed with a high priority. But static priorities like: P-tasks, are more important than T-tasks, are more important than U-tasks ... are no satisfactory solution. A U-task can be more important than a T-task if there is a P-task that depends on that U-task, or if that U-task belongs to an earlier iteration.

For our priorities we use a metric that is derived from the idea of critical paths. The critical path is the longest path within the task-DAG from one of the *starting tasks* (no incoming dependencies) to one of the *finishing tasks* (no outgoing dependencies). The reason why we study critical paths is that for each directed path $(t_1, \ldots, t_\ell)$ of the task-DAG, $\sum_i w(t_i)$ is a lower bound for the minimal time needed for the computation. This is the case because the task t_i has to be finished before t_{i+1} can be started. Thus, the computation along a path is always sequential. Therefore, we estimate the priority of each vertex t through the number of tasks along the longest path starting from t. These priorities can easily be computed because of the graphs periodic structure.

The overall longest paths within the task-DAG are the paths that go through all P-tasks. They incorporate $3n - 1$ different vertices and have the form $(P^{(1)}_{1\,1}, T^{(1)}_{1\,2}, U^{(1)}_{*\,2}, P^{(2)}_{2\,2}, \ldots, U^{(n-1)}_{n\,n}, P^{(n)}_{n\,n}, X^{(n)}_{n\,*})$ (see Figure 16). Hence, for each iteration the P-task is the task with the highest priority.

Figure 16 shows the priorities of each task as well as the longest way from that task to task $P^{(n)}_{n\,n}$. The figure does not show X-tasks because we defined the priority of each X-task to be the minimal possible priority. This way X-tasks are

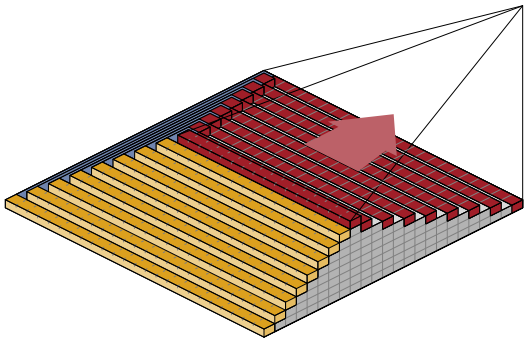


Figure 17: *Priorities within the Spatial Representation. All tasks with priority 46 and greater. All tasks that are marked in red have priority 46. They are on a plane through the spatial representation.*

executed towards the end of the execution time when otherwise there might only be few other tasks that can be scheduled in parallel.

If we look at the tasks of one global iteration, we see that the priorities decrease from left to right. We also see that all priorities of U-tasks within one column are the same. If we look at tasks from multiple iterations in the spatial perspective, described in Section 2.2.4, then we can see that all tasks that share a common priority are positioned on an angled plane within the pyramid (see Figure 17).

During the execution, tasks are computed ordered according to their priority (more or less). Tasks of higher priority are scheduled earlier than tasks of lower priority. If we assume a strict order (no task of a lower priority is executed until all higher priority tasks are finished), then work would begin on the leftmost column of the pyramid. And it would progress in angled planes (as seen in Figure 17) up to the top of the pyramid.

Priorities will mostly be used on the global scheduling level to choose which meta-tasks are scheduled to nodes. Therefore, we have to define the priority of a meta-task. We define the priority of a meta-task tm to be the highest priority among its tasks ($priority(tm) = \max_{t \in tm}\{priority(t)\}$). This is reasonable because dependencies will only be updated once the whole meta-task is finished. A single important task that depends on a meta-task can make the whole meta-task important. Luckily, priorities within a meta-task do not vary a lot (a meta-task contains at most m tile-columns).

4.2 Architecture of our Scheduler

Our scheduling approach is designed with the following goals in mind: (1) All individual tasks of one meta-task should be executed on the same node at approximately the same time. This is important for our cache improvements described in Section 3.3. (2) To enable the NUMA-optimizations described in Section 3.4, the scheduler should be able to schedule meta-tasks to specific nodes. (3) priorities should be considered during the scheduling process.

None of these goals require the scheduling to specific cores. Therefore, we use a two level hierarchical approach for our scheduling. On the global level, meta-tasks are stored until they become ready, then they are scheduled to nodes (see Section 4.2.1). Here, we have to consider NUMA-placement and priorities.

The local level is node specific. On the local level meta-tasks are unpacked into individual tasks and the individual tasks are scheduled to the cores of the local node (see Section 4.2.2).

4.2.1 Global Level Scheduling

As previously mentioned, on the global level there are no individual tasks. All individual tasks are only implicitly defined by their meta-task.

The global scheduling level consists of two main data structures: The meta-task-hash $\mathcal{H}$, that contains a reference to all meta-tasks that have not been finished, and a priority data structure $\mathcal{Q}$, that contains all ready meta-tasks that have not been scheduled to a local node.

The meta-task-hash $\mathcal{H}$ is used to keep track of all dependencies from tasks that have not been declared ready yet. Whenever a meta-task is finished, we check all its outgoing dependencies. When one of the dependent meta-tasks becomes ready, we insert that meta-task into $\mathcal{Q}$.

The priority data structure $\mathcal{Q}$ consists of $|Nodes| + 1$ priority queues that we will call $\mathcal{Q}_0, \ldots, \mathcal{Q}_{|Nodes|-1}$, and $\mathcal{Q}_{global}$. Each queue $\mathcal{Q}_i$ contains all ready tasks that should be scheduled to $node_i$. $\mathcal{Q}_{global}$ contains all ready tasks that are allowed to be scheduled to any node (without NUMA-preferences).

The scheduling process works with a pull approach shown in Figure 18. If the local scheduler from $node_i$ needs a new meta-task, it will first look at the queues $\mathcal{Q}_i$ and $\mathcal{Q}_{global}$. From these queues it will use the task with the highest priority. If both queues are empty, it will take a meta-task that was originally meant for another node. To do this it will look at all other queues $\mathcal{Q}_{j|j\neq i}$ and it will take the task with the highest priority. We call this work-stealing (a node works on a meta-task that was originally meant for another node).

4.2.2 Local Level Scheduling

The local scheduling level is node specific, therefore, each node i has its own local scheduler. On the local level, meta-tasks are unpacked into individual tasks. The individual tasks are put into a FIFO-queue $\mathcal{LQ}_i$ (First In First Out), see Figure 18. This queue acts as a task buffer. Whenever a core needs a new task, it will take the first task from $\mathcal{LQ}_i$. If there are no tasks, it will get a new meta-task from the global scheduler.

This approach guarantees that all tasks that belong to a meta-task will be executed subsequently. This increases the chances that common input tiles are co-used through the shared cache.

The core that finishes the last task of a meta-task will update the dependency information of dependent meta-tasks on the global level. Any core can be responsible for this update since any core can be the last core finishing a task. This balances the scheduling work between all cores.

4.3 Execution Order of Subtasks

Throughout this thesis, our goal is to improve the performance of subtasks by scheduling them with their data-dependencies in mind. In Section 3, we describe our approach how data-dependencies can be exploited. Now we want to analyze

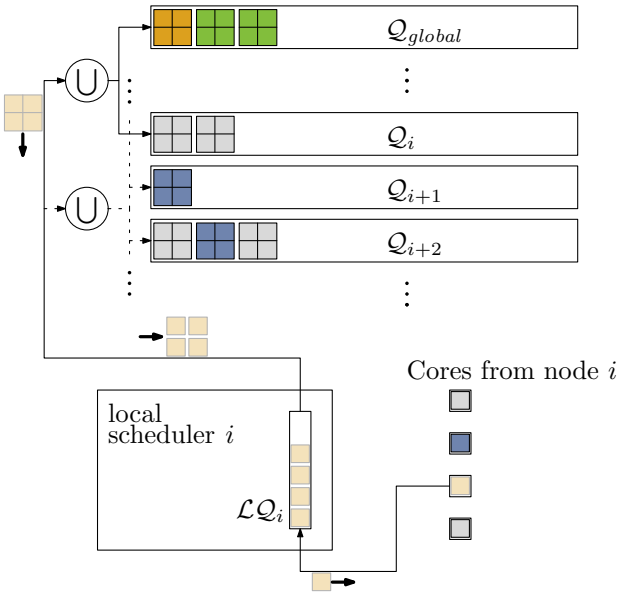


Figure 18: *Pulling a new Task. When a core needs a new task, it will first look into the local FIFO-queue. If that is empty, it will pull a task from $\mathcal{Q}_i$ or $\mathcal{Q}_{global}$. If $\mathcal{Q}_i \cup \mathcal{Q}_{global}$ is also empty, it will look into all other priority queues (dotted lines).*

how different parts of our design come together to create the execution order and how this execution order prevents expensive memory accesses.

This is especially important for U-tasks as most of our optimizations are meant to improve the execution of U-tasks. On the local level, subtasks are unpacked from a meta-task. All subtasks that belonged to one meta-task are executed consecutively on one node. Each U-task reads three tiles, the panel tile L, the tile H at the top of the matrix, and the tile A that it operates on. In Figure 19, we can see how different subtasks share the same input tiles (for a 3×3 U-meta-task). We unpack (and execute) U-meta-tasks in a column-major order $(U_1, \ldots, U_9)$. The first task that is begun has a tile-cache miss for tiles L_1 and H_1, the second, and third task each have a hit for tile H_1 and a miss for tiles L_2 and L_3. Task four will produce a tile-cache miss for tile H_2, and task seven will produce a tile-cache miss for tile H_3. This means, we can separate the subtasks into four groups: tasks U_5, U_6, U_8, and U_9 that only have a tile-cache miss for their respective A tile; tasks U_7, U_8 which have an additional tile-cache miss when reading tile H; tasks U_2 and U_3 which have a cache miss accessing tile L (and A); and task U_1 which has a tile-cache miss for each tile it accesses. Since all accessed tiles – apart from the three P tiles – are within the same meta-tile-column, there will be only three tile-cache misses that are not solvable through local memory.

The scheduling of meta-tasks follows two principal rules: meta-tasks are preferably scheduled to the node that stores the tile column they operate on (see Section 3.5); and meta-tasks are scheduled depending on their priority. Interestingly, this means that after a meta-task is finished the next meta-task that is scheduled to the node will often be within the same column as the previous meta-task. This is the case because two U-meta-tasks that operate on tiles of the same column during the same iteration have the same priority (see Section 4.1.3).

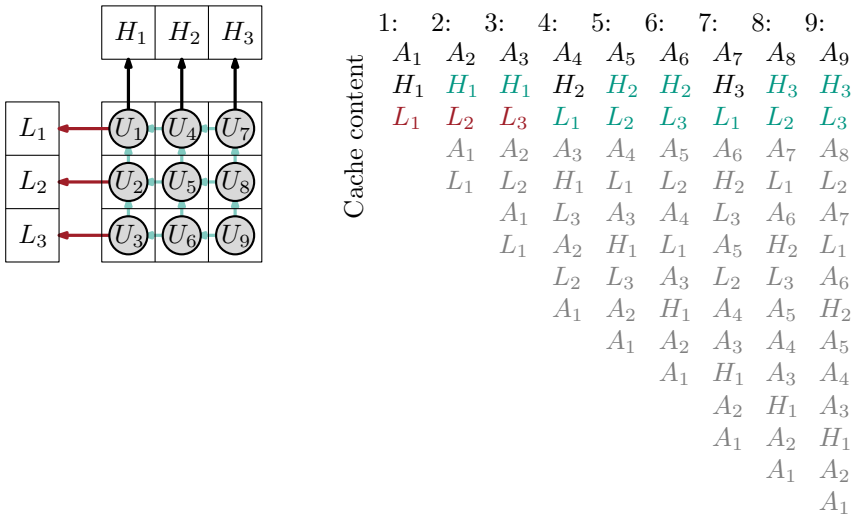


Figure 19: *Task execution order with cache status. Cache hits are highlighted green. Cache misses are highlighted black if they are locally solvable, or red if they are not (not generally local). The cache is ordered in a Least-Recently-Used ordering.*

Additionally, they become ready at the same time because they depend on the same T-task. They are also supposed to be scheduled to the same node as their input tiles are located on the same node.

If two consecutive meta-tasks do work on the same column and the tile-cache is sufficiently large, cached tiles can be shared between two meta-tasks. In Figure 19 we see, that after the execution of one meta-task, tile H_1 is at cache position 13 (there are 12 tiles read after the last use of H_1). If tile H_1 is still cached when the next meta-task is started, then the first subtask of that meta-task will have an additional cache hit. The same is true for tasks U_4 and U_7 if the corresponding tiles H_2 and H_3 are still cached.

Even if the scheduler is not optimized to exploit shared input tiles, the priority structure can influence the execution order such that tiles can be shared. As previously mentioned, two U-tasks within the same column have the same priority and become ready at the same time. Therefore, U-tasks of one column will be executed somewhat simultaneously even if they are scheduled by an unoptimized scheduler. Hence, the topmost tile H can be reused.

5 Implementation Details

5.1 Data Structures

There are three different data structures that our solution operates on: the scheduling data structures, the matrix, and the meta-task-graph.

In Section 5.1.1 we go into further detail about the data structures of our scheduler. The scheduler consists of multiple data structures that were initially introduced in Section 4.2.

As described in Section 2.1, the matrix is stored in a blocked data layout. In Section 5.1.2 we describe how we implemented the blocked data layout and how

we distribute the matrix between different nodes.

In Section 5.1.3 we describe the meta-task objects that make up the meta-task-graph. Note that these objects are stored in the scheduler data structures $\mathcal{H}$ and $\mathcal{Q}$.

5.1.1 Data Structures of the Scheduler

As described in Section 4.2, our scheduler works on two scheduling levels, the global level that consists of a hash $\mathcal{H}$ and the priority data structure $\mathcal{Q}$, and the local level which is node specific and consists of a FIFO-queue $\mathcal{LQ}_*$ (on each node). We use general `stl`-implementations (Standard Template Library) for each of these data structures. These implementations are generally not thread safe. Therefore, they have to be protected with locks such that only one thread can access each data structure at a time.

Task Hash $\mathcal{H}$ The task hash holds pointers to all unfinished meta-tasks. Each meta-task has an id which is used as its key. As already mentioned, we use a sequential implementation for this hash. Therefore, we have to use a lock to protect it. We call this lock the $\mathcal{H}$-lock. This lock is not only used to protect simple accesses to $\mathcal{H}$. In Section 5.2.1 we describe how this lock can be used to prevent race conditions that can occur when threads are working while tasks are still being generated/inserted into $\mathcal{H}$.

Priority Data Structure $\mathcal{Q}$ As described in Section 4.2, $\mathcal{Q}$ consists of multiple different priority queues that contain all meta-tasks that are ready to be executed. We use multiple queues to be able to schedule tasks in a NUMA-aware way ($\mathcal{Q}_{global}, \mathcal{Q}_0, ...$). Tasks are inserted into the queue that belongs to the node that they are preferably scheduled to.

Even though $\mathcal{Q}$ consists of multiple priority queues, we only use one lock to protect all priority queues simultaneously. We do this because all threads that take elements from $\mathcal{Q}$ access $\mathcal{Q}_{global}$. Therefore, protecting all queues together has the same effect as locking them individually. It is more advantageous to reduce the contention on this one lock. To do this, we ensure that there can be at most one thread per node that seeks to remove meta-tasks from $\mathcal{Q}$ at any given time.

Local FIFO-Queues $\mathcal{LQ}_*$ Since the local level is specific to each node, we will have one lock per local queue. In addition to protecting the local queue, we use this lock to ensure that there can be at most one thread (per node) looking for a new meta-task (see Section 5.3).

5.1.2 Practical Matrix Layout

Usually a matrix is stored in a continuous piece of memory. This is the case in LAPACK [2] for example, there the standard matrix layout is a column major format. As described in Section 2.1, we work with a blocked data layout. This means that the matrix is subdivided into tiles. One can still store all tiles back to back into a consecutive piece of memory (this is done in PLASMA).

We chose to truly separate all tiles, therefore, we store an $n \times n$ matrix with pointers to all tiles, which are themselves $b \times b$ matrices. Each tile is stored in a continuous piece of memory using a column major format, as is the matrix of

pointers. Different tiles are not stored together. We use this layout because it enables us to explicitly store matrix tiles on specific nodes of the system.

As described in Section 4.1.2, we distribute the matrix tiles in a meta-column-wise layout. This pattern is meant to combine the advantages from distributing the matrix tile-column-wise (described in Section 3.4) with the concept of meta-tiles (motivated in Section 3.3).

5.1.3 Meta-Task

As previously described in Section 4.1, a meta-task is a set of tasks that have been bundled together. On the global level, each individual task is only represented implicitly through its meta-task. From an implementation standpoint, a meta-task consists of many different attributes which are important for different steps of the scheduling process:

There are general attributes:

- `id`: uniquely identifies each meta-task. It is used synonymously to the whole meta-task, and at run-time a meta-task can be found through its `id` (within $\mathcal{H}$).
- `counter`: atomic counter, that we use for multiple steps of the scheduling process. Its uses will be described more thoroughly in Section 5.2.2.

Some attributes are responsible for describing the implicit tasks that are represented by each meta-task. As we do not store any tasks directly, we do have to store some information that we can use to create the represented tasks. Each task can be uniquely identified by its identifier $Y_{i,j}^{(k)}$ ($Y \in \{P, T, U, X\}$), therefore, we store:

- `type`: describes the class of the individual tasks (P-, T-, U- or X-tasks) within that meta-task (in our implementation there can be only one type, see Section 4.1.2).
- `k`: the number of the iteration that all tasks within the meta-task belong to.
- `position` and `size`: needed to specify the tiles that the individual tasks work on.

Other attributes are used in the global scheduling process to determine when a task becomes ready, and when and where the ready task should be scheduled:

- `dep_in`: a list of all meta-tasks that this meta-task is dependent on (stored as ids). With the help of this list it is possible to check if a meta-task is ready to be scheduled (see Section 5.2.1).
- `dep_out`: a list of all meta-tasks that are dependent on the meta-task (also stored as ids). We use this list to update all dependent meta-tasks once a task is finished.
- `priority`: important for the scheduling process.

- `preferred`: $\in \{global, 1, ..., |Nodes|\}$. This attribute is important for our NUMA-optimization. It stores the node id where this meta-task should be scheduled to optimally use the local memory (mt will be put into $\mathcal{Q}_{mt.\texttt{preferred}}$).

5.2 Description of a Meta-Task's Lifetime

Before we go into detail on the stages a meta-task goes through during the computation (see Section 5.2.2), we want to describe how meta-tasks are declared ready (see Section 5.2.1).

5.2.1 Readying a Meta-Task

Each meta-task has a counter. Before the meta-task is ready, we use this counter to count the number of finished dependencies that it has. After the creation of each meta-task its counter will be zero. Whenever a meta-task mt is finished, we find all meta-tasks mt_{out} that are dependent on mt and increment their counters. If mt_{out}'s counter equals the number of mt_{out}'s incoming dependencies, then mt_{out} is ready and it can be scheduled.

However, this method has one problem. It assumes that all meta-tasks are created before the first one is started. This is a problem because the the creation of the meta-task-graph is not parallelized. Only one core can work on the meta-task creation. To raise the efficiency of our implementation it must be possible for all other cores to work on the meta-tasks that have already been created. Thus, when the meta-task mt is finished, we cannot be sure that all dependent meta-tasks mt_{out} have already been created.

Therefore, if we finish the computation of mt, and look for mt_{out} in $\mathcal{H}$ to increase its counter, we might not find mt_{out}. If this is the case, then we cannot increase mt_{out}'s counter. This might mean that mt_{out} is never declared ready because its counter would be too small.

Our solution to this problem is that whenever a task mt is inserted into $\mathcal{H}$, we check all its dependencies to find the number of dependencies that have already been fulfilled (without being able to increase mt's counter). This can be done by searching all meta-tasks mt_{in} that mt depends on in $\mathcal{H}$. The order in which meta-task are created ensures that mt_{in} was created before mt. Therefore, if mt_{in} is not in $\mathcal{H}$, it must be finished already. Hence, we increase mt's counter for each meta-task mt_{in} that is not in $\mathcal{H}$.

This solution works, but it opens the possibility for a race condition. Imagine a meta-task mt_a is being finished, by a $core_a$, while $core_0$ inserts the dependent meta-task mt_b into $\mathcal{H}$. In this case mt_b's counter might be increased too often. For example, if $core_a$ removes mt_a from $\mathcal{H}$, then $core_0$ inserts mt_b and checks its dependencies. In this scenario, $core_0$ will increment the counter because mt_a is not in $\mathcal{H}$ anymore. While this happened, $core_a$ checks the meta-tasks that are dependent on mt_a. It finds mt_b in $\mathcal{H}$, and increment its counter a second time. Because of this race condition, we have to protect the insertion and the deletion of tasks through locks. We use the $\mathcal{H}$ lock that protects accesses to the task hash $\mathcal{H}$ to also protect the [insertion + check for readiness] and the [deletion + updating dependent tasks] (see Section 5.3.2 and Figure 20).

5.2.2 Overview

In this section we look at a meta-task mt and all the stages it goes through during the scheduling process. Each meta-task is generated by $core_0$ at the beginning of the execution. All meta-tasks are created by in the order described in Algorithm 2. The attributes are computed (id, position/size, incoming and outgoing dependencies, ...), and the counter is set to 0.

After its creation, mt is put into $\mathcal{H}$, which holds all unfinished tasks. At this point we check if mt is ready by searching each meta-task mt_{in} that it depends on in $\mathcal{H}$. The meta-task mt_{in} was created before mt. Hence, it has already been inserted to $\mathcal{H}$. If it is not there anymore, it must already be finished. For each finished dependency we increase mt's counter by one. If the counter is equal to the number of incoming dependencies, mt is ready to be scheduled. Otherwise mt will remain "not ready" until all its dependencies have been fulfilled.

When mt is ready, it will be inserted into the priority data structure $\mathcal{Q}$ (note that it is not removed from $\mathcal{H}$ as it is not finished yet). Depending on mt's preferred node h, mt is inserted into $\mathcal{Q}_h$. Once mt is the most important meta-task in its priority queue, it might be scheduled to a local scheduler (see Section 4.2.1, Figure 18, and 20).

When mt is scheduled to a node, we reset its counter to 0. Then we unpack its individual tasks $(t_1, \ldots, t_\ell)$. These are inserted into the local FIFO-queue. Whenever a task t_i is finished, we increase mt's counter by one. The core that increases the counter to ℓ is responsible for finishing mt (no race condition because the counter is atomic).

To finish mt, we remove it from $\mathcal{H}$. Then we go through the list of all outgoing dependencies. We look for each dependency mt_{out} in $\mathcal{H}$. If mt_{out} is in $\mathcal{H}$, we increase its counter to represent that one of its incoming dependencies has been fulfilled. This might make mt_{out} ready (in that case it is inserted into $\mathcal{Q}$). If mt_{out} is not in $\mathcal{H}$, then it has not been inserted yet (it cannot be finished because it cannot be ready). In this case we do nothing because when mt_{out} is inserted into $\mathcal{H}$, its dependencies will be checked, and it will be clear that mt has already been finished.

5.3 Work-Cycle of a Thread

The first thread (on $core_0$) is responsible for generating tasks. After it created all tasks, it begins working together with all other threads. In the following we describe the usual work cycle of a thread p which belongs to the node s. This work cycle can also be seen in Figure 20 which shows a flowchart representation of the procedure.

Whenever p needs a new task, it acquires the $\mathcal{LQ}_s$-lock. Then it accesses $\mathcal{LQ}_s$ and checks the number of tasks in $\mathcal{LQ}_s$. If the number of tasks is smaller than a limit c_ℓ ($\approx$ *number of cores on the node*), then the queue should be refilled with new meta-tasks. The process of filling the queue with new meta-tasks is described in Section 5.3.1.

When there are enough tasks in the local queue $\mathcal{LQ}_s$, then p removes the first task t and unlocks the $\mathcal{LQ}_s$-lock. Now, p executes the task t. After the execution we increase the counter of the meta-task that t belongs to. With the help of this

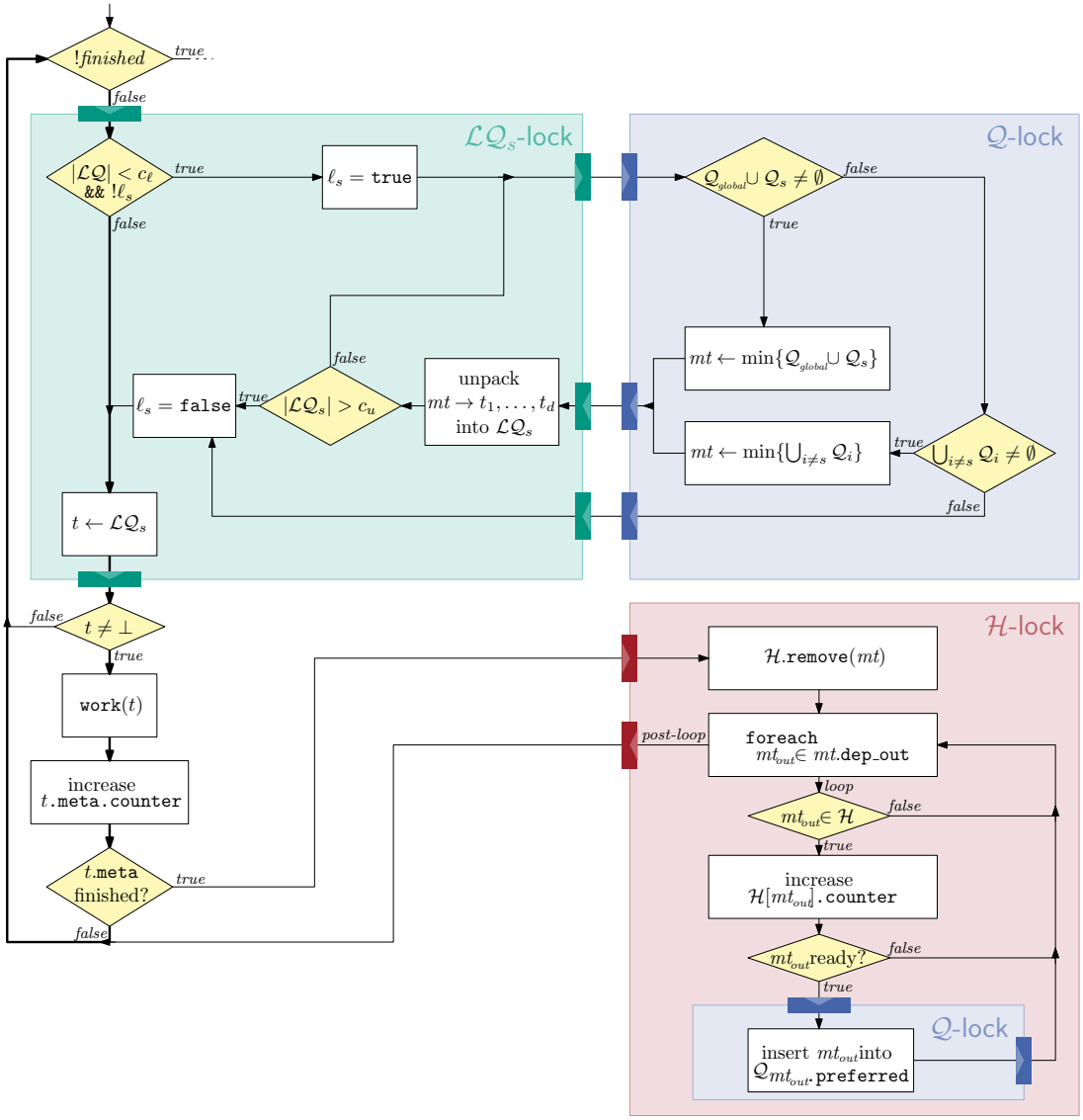


Figure 20: *Thread work cycle. This flowchart shows the work cycle of a thread on node s (main work cycle bold). The colored areas represent locks that the thread holds.*

counter, we can check if t was the last task that belonged to this meta-task. If t was the last task, then p is responsible for deleting the meta-task and updating dependent meta-tasks (see Section 5.3.2). Otherwise (or after the update) p is ready to work on another task and the work cycle begins anew.

5.3.1 Filling the Local Queue with new Meta-Tasks

The procedure of getting new meta-tasks can be seen in Figure 20 (green and blue area). To reduce the contention on global locks, it is important that there is at most one thread per node looking for new meta-tasks (at a time). Therefore, we introduce the ℓ_s flag, which signalizes that there is a thread from node s looking for new meta-tasks.

When p wants to get a new task, it first takes the $\mathcal{LQ}_s$-lock and checks if $\mathcal{LQ}_s$ contains enough elements ($\geq c_\ell \approx$ *number of cores on the node*). If this is not

the case and the ℓ_s flag is not set (no other thread is already looking), p will be looking for new meta-tasks. It sets ℓ_s while it still holds the $\mathcal{LQ}_s$-lock (to prevent race conditions). Then it unlocks the $\mathcal{LQ}_s$-lock so other threads can still access the remaining tasks (up to c_ℓ tasks are still available).

Now, p is looking for new meta-tasks on the global level. It begins with reserving the $\mathcal{Q}$-lock, which protects the global priority data structure. From all meta-tasks in $\mathcal{Q}$, p chooses a meta-task. As described in Section 4.2.1, p first checks the priority queues $\mathcal{Q}_{global}$ and $\mathcal{Q}_s$. From these, it takes the meta-task with the highest priority. If $\mathcal{Q}_{global}$ and $\mathcal{Q}_s$ are empty, p chooses the meta-task with the highest priority among all other priority queues $\mathcal{Q}_{i \neq s}$.

When p has obtained a meta-task mt, it releases the $\mathcal{Q}$-lock and reacquires the $\mathcal{LQ}_s$-lock before it unpacks mt into the individual subtasks $t_1, \ldots, t_\ell$. These are inserted into $\mathcal{LQ}_s$. Before p inserts them into the queue, it resets mt's counter to zero (this counter is now used to count the number of finished subtasks). After all tasks are inserted, p checks the number of tasks in the local queue $\mathcal{LQ}_s$ again. If there are still not enough tasks in the local queue ($\leq c_h$; usually $c_\ell \leq c_h \leq 2c_\ell$), then p repeats this procedure until there are enough tasks locally. When there are enough tasks, p resets the ℓ_s flag and resumes its work cycle. Notice that we use two different constants ($c_\ell \leq c_h$). Tasks are added when there are less then c_ℓ tasks until there are more then c_h tasks. With both parameters, we can regulate the frequency and size of the queue refills.

5.3.2 Finishing a Meta-Task

The procedure of finishing a meta-task can be seen in Figure 20 (red area). If p finished the last task of meta-task mt, then p is responsible for finalizing mt (p was the last task to increase mts atomic counter). To do this, p reserves the $\mathcal{H}$-lock. Then it removes mt from $\mathcal{H}$.

Now, p has to update all dependent meta-tasks. For each meta-task mt_{out} that depends on mt we check if that meta-task already exists within $\mathcal{H}$. If it does, we increment its counter. If it does not, we ignore mt_{out} because its counter will be updated when it is inserted into $\mathcal{H}$.

When the counter of mt_{out} equals the number of mt_{out}'s incoming dependencies, then mt_{out} is ready and has to be scheduled. In this case, p reserves the $\mathcal{Q}$-lock and inserts mt_{out} into the appropriate priority queue $\mathcal{Q}_{mt_{out}\texttt{.preferred}}$. After inserting mt_{out}, p unlocks the $\mathcal{Q}$-lock. This is the only instance where two different locks are held by one task. Once p updated all dependent subtasks, p also releases the $\mathcal{H}$ lock and continues working on other tasks.

5.4 Panel Algorithm

The panel algorithm (implementation of P-tasks) is central to any LU-decomposition algorithm. Even though submatrix updates (U-tasks) compose the majority of the work, panel updates (P-tasks) are very important because they are on the critical path of the computation, and every single P-task performs significantly more work than a single U-task.

Because a P-task performs that much work, it is very appealing to parallelize its implementation. Hence, we use a parallel panel algorithm (see Algorithm 3).

Algorithm 3: Panel Algorithm (parallel, recursive)

`Data`: This algorithm is executed in parallel on multiple cores (id =core id within the task). Each core works on a fixed set of tiles within the panel. All highlighted tasks (green) can be executed "independently".

```
1  Function recursive_LU(id, ℓ, r)
2      if r − ℓ > 1 then
3          mid = (r+l)/2                          // split into left and right part
4          recursive_LU(id, ℓ, mid)               // see Figure 21b
5          if id = 0 then
6              triangular solve on the top right  // ≈T-task see Figure 21c
7          synchronize
8          submatrix update on the right side     // ≈U-tasks see Figure 21c
9          recursive_LU(id, mid, r)               // see Figure 21d
10     else                                       // r − l = 1
11         (ℓ_id, row_id) ← local maximum, and its row   // see Figure 22b
12         (pivot, q_ℓ) ← exchange_max(ℓ_id, row_id)     // see Figure 22c
13         if id = 0 then
14             row_exchange(q_ℓ, ℓ)               // see Figure 22d
15         compute values in column ℓ  // a ← a/pivot see lines 4,5 in Algorithm 1
```

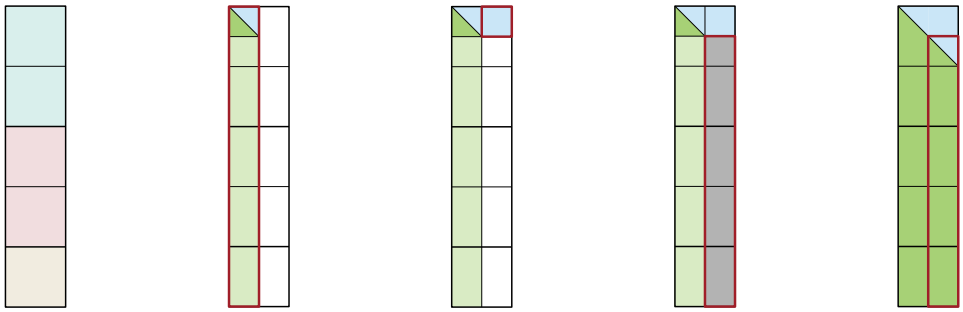
`Output`: LU-decomposition of the panel with pivoting elements $q_l, \ldots, q_r$

This algorithm was also used by Dongarra et al. [10]. It distributes the panel tiles between all participating cores (see Figure 21a). Every core is responsible for a fixed subset of panel tiles.

The panel is factorized cooperatively by splitting it into a left and a right part. The two parts are solved recursively. At first the left side is factorized (line 4 and Figure 21b). Then, the right side of the split has to be updated with the row operations made on the left side. This is comparable to an iteration of the tiled algorithm (described in Section 2.2.3). First, the core that is responsible for the first panel tile performs a triangular solve on the top square of the right side (line 6 and Figure 21c) – comparable to a T-task. Then, all cores perform a submatrix update (line 7 and Figure 21d) – comparable to a U-task – on the tiles that they are responsible for. After the right side is updated, it can be solved using the same recursive algorithm (line 9 and Figure 21e).

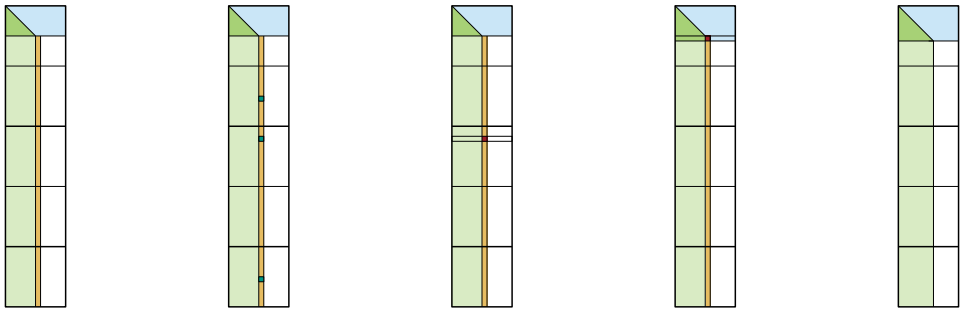
After a number of splits there is only one column left (line 10 and Figure 22a). At this point, all cores find the maximal value in their part of this column (line 11 and Figure 22b). These local maxima are exchanged to find the global maximum (line 12 and Figure 22c). The row with the global maximum will be the pivot row. It is exchanged to the top tile of the panel (line 13, 15, and Figure 22d). The pivot element is used to compute all values of the column (line 15 Figure 22e) by dividing them with the pivot element – comparable to lines 4 and 5 in Algorithm 1.

In theory this algorithm can be used to compute a complete LU-decomposition (not only the panel) but it does not scale very well with the number of processors. The problem of this algorithm is that there are frequent synchronization points



(a) *work distribution* (b) *recursion on the left* (c) *triangular solve* (d) *submatrix update* (e) *recursion on the right*

Figure 21: *Panel algorithm recursion. The panel algorithm is executed in parallel. Each core works on a subset of tiles depending on its id within the P-task. In this case there are three cores working on five tiles.*



(a) *single tile instance* (b) *find local maxima* (c) *find the global pivot* (d) *swap pivot row to top* (e) *compute column*

Figure 22: *Panel algorithm terminating case. Once the recursion arrives at single column instances, the cores search for a pivot (together). The pivot row is exchanged to the top row and all L values of the column are computed.*

between cores. Because of this, we chose to reduce the number of cores that work on each panel compared to PLASMA. In PLASMA it is possible that all cores (up to 40) cooperate on one P-task. In our implementation we reduced the maximum number of cores that cooperate of one P-task to the number of cores on the node where the P-task is executed.

One reason why reducing the number of processors within one P-task is beneficial is that after the first P-task there are a lot of tasks that become ready. Working on these tasks is very efficient compared to the suboptimal parallelization of P-tasks. As long as there are other more efficient tasks to work on, P-tasks should not be executed by more cores than necessary. In our experiments we found that only using cores of one node did not lead to a situation where cores were waiting idly, for the P-task to finish. Using only cores from one node can also accelerate the communication and synchronization overheads because all cores have access to the same caches and local memory.

All cores that start working on one P-task have to wait for all other cores that also work on this P-task until they can get started. They will wait at the first synchronization point until every core reached that point. Therefore,

it is beneficial to not only reduce the number of processors per P-task but to ensure that all processors enter the P-task at approximately the same time. Fortunately, this can easily be enforced within the architecture of our scheduler, by creating one P-meta-task that is unpacked into x smaller subtasks (x is the number of cooperating threads). Through the local scheduling structure with a FIFO-queue, we can guarantee that no other task will be started before the P-task is sufficiently attended. This guarantees that, as soon as the first thread enters a P-task, the next $x - 1$ threads (of the same node), that are looking for a task, begin working on the same P-task.

6 Experiments

In this section, we describe in detail how we tested our implementation against the current state of the art. Within our evaluation, we look closely at each of our optimizations. To gain more insight into the influences of our optimizations we not only measured the running time of the executions. We closely measured the running times of each subtask that was executed within the computation.

Within this section, we first describe the surroundings of our measurements. In Section 6.1 we describe the test systems and their software setups. Then in Section 6.2, we describe the test instances and sizes. Section 6.3 gives an overview of our results, and the advantages of our implementation compared to the current state of the art (PLASMA). After this overview, we use Section 6.4 to go into further detail about the performance of our implementation. There we analyze the influences of each optimization on the running time of subtasks.

Then, we analyze the schedule from a theoretical point of view by analyzing the measurements with the machine model that we developed in Section 3.1. The results of this analysis can be seen in Section 6.5. At the end of this section – in Section 6.6 – we review our implementation with consideration of different performance metrics. There we will analyze the energy consumption and the efficiency with which we use the systems resources.

6.1 Hardware and Environment

We ran our experiments on three different machines. These three machines combined are representative for most (compute-)server architectures used today. All of our systems use an Ubuntu Linux operating system and have speed-stepping enabled.

The main system, that we could extensively test our implementation with, is a system consisting of four Sandy Bridge (Intel Xeon E5-4640) processors with eight cores each. The cores are running at $2.4 GHz$ (base frequency). We call this machine SandyBridge32. We use SandyBridge32 as the main system for our testing because it is the biggest system that we could regularly test on. If not explicitly mentioned otherwise, all measurements used in the text and in images are taken on SandyBridge32.

SandyBridge32 has $512 GB$ of main memory distributed evenly between all four sockets ($128 GB$ each). Additionally, all cores of a single processor share a common L3 cache with $20 MB$. The processors support AVX instructions to

accelerate floating point computations. This leads to a theoretical peak performance of $614.4\frac{Gflop}{s}$ (double precision), though this could be surpassed using speed-stepping. SandyBridge32 runs on kernel version 3.2.0-75-generic.

The second machine, that we regularly tested on, is a two socket system with two Ivy Bridge (Intel Xeon E5-2650 v2) eight core processors (very similar to those on SandyBridge32) each running at $2.6GHz$ (base frequency). We call this system IvyBridge16. IvyBridge16 has $128GB$ of main memory which is distributed between both sockets ($64GB$ each). Comparable to SandyBridge32, each processor on IvyBridge16 has a $20MB$ L3 cache that is shared between all its cores. Since IvyBridge16 also supports AVX instructions, it has a theoretical peak performance of $332.8\frac{Gflop}{s}$. IvyBridge16 runs on kernel version 3.13.0-36-generic.

The newest machine, that we were able to use for testing, is a 2-socket Haswell (Intel(R) Xeon(R) CPU E5-2670 v3) system with 24 cores that have $2.3GHz$ each. We configured this system to distribute its cores into two NUMA-nodes, one per socket (alternatively, one could separate the 12 cores per socket into two NUMA-nodes). We call this system Haswell24. Haswell24 is the only machine in our test set that supports the new AVX2 instructions. Haswell24 has $128GB$ of main memory ($64GB$ per node) and $30MB$ of shared L3 caches per node. It runs kernel version 3.13.0-45-generic.

To compare our implementation with the current state of the art we used the PLASMA library. PLASMA is a library for numerical linear algebra problems, that is freely available. Dongarra et al. [10] have shown that it has a significantly better performance than LAPACK and intel's MKL (Math Kernel Library). In all our tests, we used PLASMA version 2.6.0, which was released in December of 2013.

Comparable to our implementation, PLASMA uses a BLAS (Basic Linear Algebra Subprogram) library for its low-level matrix manipulations. BLAS libraries offer highly tuned matrix computations that can be used to circumvent the need to reimplement and retune simple functionality. For our tests, we used the MKL's BLAS implementation. All tests were made with MKL version 11.0 (update 5).

We also use *PAPI*[16] (Performance Application Programming Interface), which is a library that can be used to read hardware performance counter. In our tests, we use these performance counter to measure detailed performance characteristics like the amount of L3 cache misses that happen during subtask executions. We used PAPI version 5.3.2.0. All test programs were compiled with version 13.1.3 of the Intel compiler suite (icc/icpc). During the writing of this thesis Intel released new versions of their compiler (15.0.0), and the MKL (11.2 update 1). Sample tests with the new versions have shown no changes.

6.2 Tested Variations and Inputs

During our experiments, we tested the following variations and scheduling schemes:

PLASMA: (sometimes called P or P$\langle b \rangle$) within PLASMA, we used the function called `dgetrf_tiled`, which is the standard algorithm for the LU-decomposition of tiled matrices. In our preliminary testing this function was the fastest algorithm for LU-decomposition within PLASMA. As advised by the PLASMA users

guide [11] we ran all tests made with PLASMA on interleaved memory to reduce NUMA-effects (`numactl --interleave=all`).

M⟨m⟩col: our scheduler with all optimizations as they are described in Section 3.3 and 3.4. The meta-tile size is set to $m \times m$. For most measurements on SandyBridge32 we use $m = 3$, this has generally been an optimum in our experiments. On IvyBridge16 and Haswell24 a meta-tile size of $m = 5$ has achieves the best performance.

M⟨m⟩nN: our scheduler with a meta-tile size of $m \times m$ but without the NUMA-optimizations. In this variation there is no care taken where matrix tiles are stored and where meta-tasks are scheduled to. Each matrix tile is still stored on one NUMA-node but that NUMA-node is chosen at random, and not all tiles from one meta-tile are stored on the same node (not usually). To keep all cache optimizations intact, subtasks belonging to one meta-tasks are still scheduled to one node.

M1nN: our scheduler with a meta-tile size of 1×1 and without NUMA-optimization. This is a variation of our scheduling scheme without any optimizations described in Section 3. The meta-tile size of 1×1 effectively makes meta-tasks equivalent to single subtasks, thus, eliminating the cache optimization. We use this variation as a reference point for our comparisons. Using M1nN we can analyze how much performance can be gained through each optimization.

We tested all these variations on matrices with between 4096 and 32768 elements per side and with varying tile sizes. For PLASMA the optimal tile size strongly depends on the matrix size. Therefore, we tested a wide range of tile sizes between 240 and 512 (240, 256, 280, 288, 320, 352, 384, 416, 448, 480, and 512). Whenever we mention PLASMA with a fixed tile size, we explicitly choose the best tile size for that matrix size (if not explicitly stated otherwise). Because the running times of our implementation are less dependent on the right combination of matrix- and tile size, we only tested a smaller sample of tile sizes (240, 256, 288, 320, 384, 448).

To control the amount of variance within our tests we repeated every test instance five times. With five running times per test instance we can compute the average time that each decomposition takes. We can also compute the standard deviation between the different iterations. It is important to consider the deviation of a measurement to determine if results are significant. Especially on small matrices running times can vary a lot.

To ensure the comparability of all measurements, we use the same matrix generator for all measurements. All matrices are generated with the matrix generator that was used in PLASMA's timing subprograms. We reimplemented that matrix generator for our tests.

In addition to all running times, we also want some more detailed information on the performance of all subtasks, therefore, we log each execution of a subtask. For each subtask, we measure the time as well as the amount of cache misses that the subtask caused during its execution. To count the number of cache misses, we use hardware counters which we access through the PAPI library. With the help of these detailed performance logs, we are able to analyze the effect that even small scheduling changes have on the overall performance of subtasks.

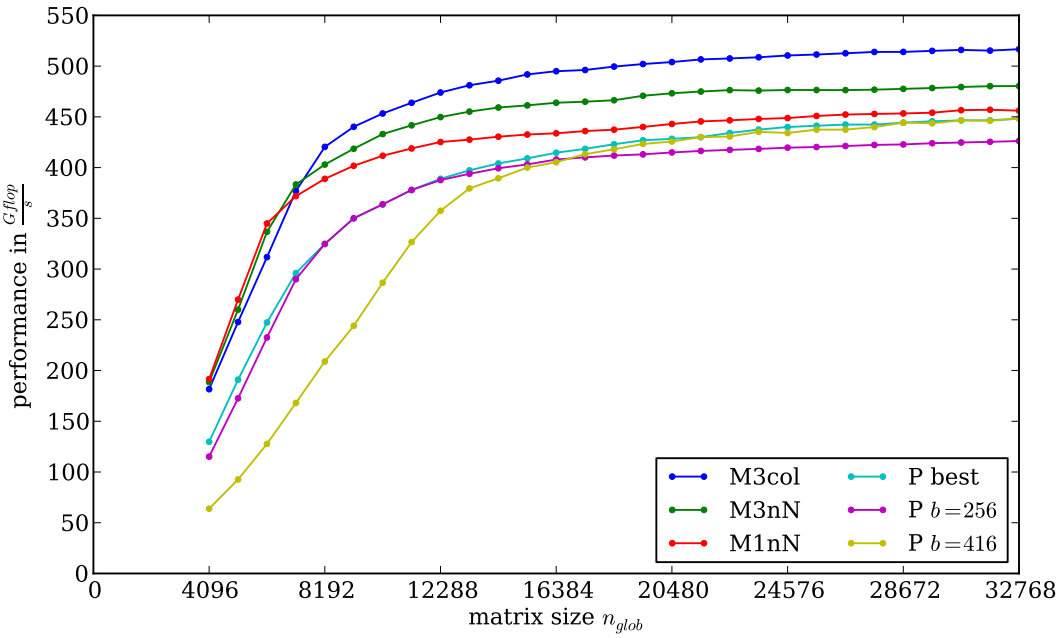


Figure 23: *Main performance plot. This plot shows the performance ($\frac{Gflop}{s}$) of different scheduling variations over different matrix sizes (n_{glob}).*

6.3 Overview of our Test Results

The diagram in Figure 23 shows the performance of different scheduling variations on matrices of varying sizes. The performance is measured in Gflop per second. It is computed by dividing number of Gflop that are theoretically needed to compute the LU-decomposition (lower bound for LU-decomposition: $\frac{2}{3}n_{glob}^3 - \frac{1}{2}n_{glob}^2 + \frac{5}{6}n_{glob}$) through the execution time of the LU-decomposition. The diagram clearly shows that our scheduling optimizations improved the overall performance of the numerical algorithm significantly. It highlights two distinct effects that separate our implementation from that of our competitors. We achieve a 15% better peak performance ($517\frac{Gflop}{s}$ with M3col against $448\frac{Gflop}{s}$ for PLASMA with tile size 416). And we achieve this peak performance even on smaller matrices, for example, we reach a 29% higher performance on a matrix with $n_{glob} = 8192$ ($420\frac{Gflop}{s}$ with M3col against $325\frac{Gflop}{s}$ PLASMA with $b = 256$).

If we compare different variations of our scheduling scheme, it becomes clear that our combination of cache- and NUMA-optimizations is responsible for the increased peak performance. The variation M1nN shows that without any optimizations the peak performance is comparable to that of PLASMA ($456\frac{Gflop}{s}$ with M1nN against $448\frac{Gflop}{s}$ with PLASMA $b = 416$). This is not surprising considering that we used the PLASMA implementation as a starting of point for our implementation. The experiments with M3nN show that the cache-optimization by itself improves the peak performance significantly (to $480\frac{Gflop}{s}$). But only the combination of both optimizations results in the best peak performance ($517\frac{Gflop}{s}$).

The increased performance of our implementation on smaller matrices is nearly independent from the scheduling variation. This could indicate that the task

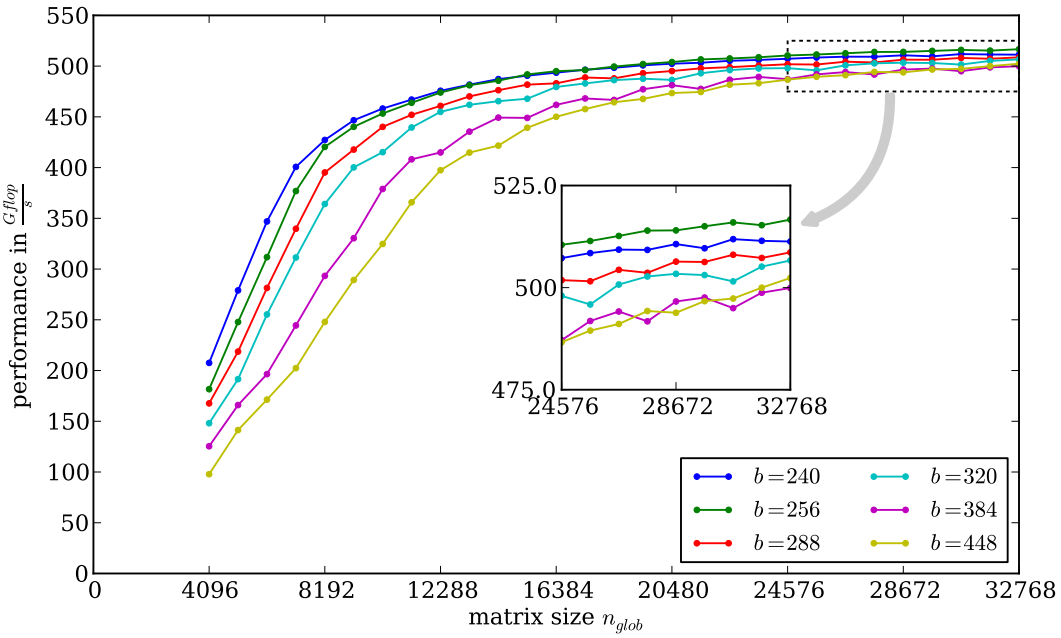


Figure 24: *Varying tile sizes. This measurement shows the influence of different tile sizes on the performance of our implementation (M3col executed on SandyBridge32).*

generation of our scheduler is more effective. It is designed to generate tasks as quickly as possible and to schedule tasks even before the complete task-DAG is generated. Thus, it has some distinct advantages over PLASMA which is a general library that is not as specialized on the LU-decomposition. These advantages are somewhat negligible on large matrices where the task generation does take less time relative to the computation. Contrary to the performance on big matrices, it can even be beneficial to turn off certain optimizations when working on small matrices. This might be an additional sign that on smaller matrices it is important to quickly generate and schedule all subtasks. Instead of worrying about correct NUMA-placement and smart grouping of tasks.

6.3.1 Influence of the Tile Size

In Figure 24 we can see that bigger tile sizes lead to worse performance on small matrix sizes. The reason for this is that bigger tile sizes on small matrices lead to less overall tasks, which leads to several problems. Less tasks means that there are times where not all cores are used at full capacity. It also means that the ratio of U-tasks towards the other task classes is slightly decreased, which is problematic since U-tasks are the most efficient class of subtasks ($\frac{Gflop}{s}$-wise).

Additionally, big tiles can reduce the effectiveness of our NUMA-optimization. Our NUMA-optimization works by distributing the responsibility for these meta-tile-columns between NUMA-nodes (see Section 3.5). Bigger tiles lead to a reduced number of meta-tile-columns, which can lead to a work imbalance between nodes. Especially on smaller matrices, these imbalances can be significant.

On bigger matrices the tile size loses some of its influence. Even the worst

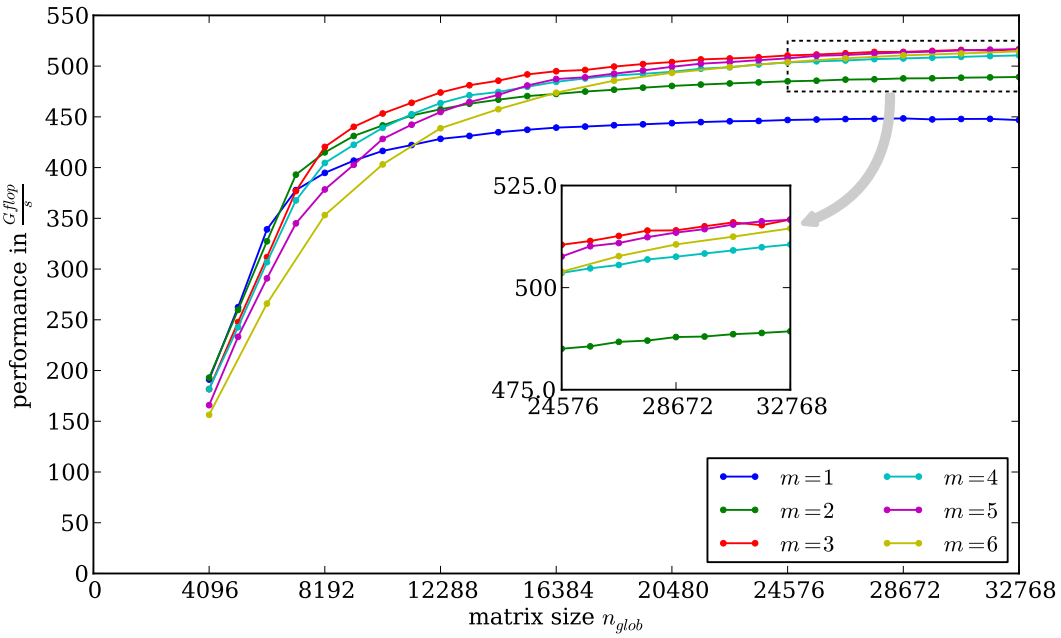


Figure 25: *Varying meta-tile sizes. This measurement shows the influence of different meta-tile sizes on the performance of our implementation ($b = 256$ on SandyBridge32).*

tile size for a matrix with $n_{glob} = 32768$ still achieves around 97% of the optimal performance ($500\frac{Gflop}{s}$ with $b = 384$ against $517\frac{Gflop}{s}$ with $b = 256$).

6.3.2 Influence of the Meta-Tile Size

In Figure 25 we see that for small matrices big meta-tile sizes have the same effect as big tile sizes. They reduce the number of available meta-tasks that can be scheduled to nodes. Hence, it is possible that some cores idle needlessly. Meta-tasks with a size of 4×4 or greater have significantly more tasks than a node has cores. When such a big meta-task is scheduled to a node then there are tasks that are not started directly. Therefore, it is possible that cores idle because there are no more meta-tasks ready to be scheduled, while other nodes still have ready tasks that are not executed yet.

On bigger matrices, bigger meta-tile sizes achieve a better performance than smaller meta-tile sizes. Especially M1col and M2col are quickly outclassed. M3col seems to be the best meta-tile size. It is the optimal meta-tile size for nearly all matrix sizes, only rivaled by M5col (and M1col/M2col for very small matrices).

Interestingly, M4col seems to be significantly slower than other comparable meta-tile sizes. One possible reason for this is that the number of tasks within a 4×4 U-meta-task (16) is a multiple of the number of cores that SandyBridge32 has per node. Hence, there can be a synchronization between multiple cores (of the same node). This could repeatedly lead to a situation where two processors simultaneously read the same input tiles. In this situation, both processors have to wait for the cache misses to resolve. Thus, both core cannot profit from fast

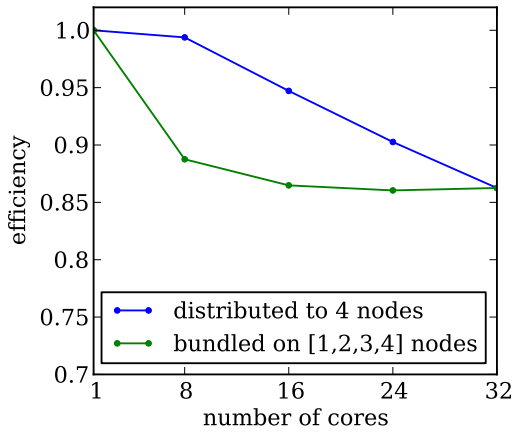


Figure 26: *Relative efficiency. All measurements were taken on a matrix with* $n_{glob} = 32768$. *There are two options when reducing the number of cores: the first option is to reduce the number of cores per node and the second option is to reduce the number of nodes.*

access times of the shared L3 cache because the accessed memory sections are not loaded yet. For a more detailed analysis of this effect see Section A.1.

Another interesting point is that M1col performs worse than the unoptimized version M1nN ($447\frac{Gflop}{s}$ with M1col against $456\frac{Gflop}{s}$ with M1nN). The reason for this is probably that our NUMA-optimization synchronizes accesses to the panel tiles (which are not stored locally). It is probable, that no two threads access the same panel tile, so all panel tiles have to be loaded separately. This is a problem since all panel tiles, of one iteration, are stored on the same node, creating a potential bandwidth problem on the QPI links.

6.3.3 Scalability

In this section, we want to explore how efficient our solution adapts to different machine setups. We demonstrate how our solution scales to a varying number of cores and we present running times of our implementation on IvyBridge16 and Haswell24.

Varying Number of Cores on SandyBridge32 Figure 26 shows the efficiency of our implementation on SandyBridge32. The efficiency of our implementation never drops below 86%. Interestingly, using all cores of one node reduces the efficiency drastically, compared to spreading the used cores onto multiple different nodes. The biggest difference can be seen when we use eight cores. If these eight cores are spread out between all four nodes, then we achieve a nearly perfect efficiency of 99% but using all eight cores of one node only reaches an efficiency of 89%. This is a reduction by 10% similar efficiency differences can be seen for 16 and 24 cores (8%, and 4%).

One reason for this is the multiplication of memory resources. If only two out of eight cores work on each node, they still have access to the whole L3 cache and memory bandwidth. Both of these resources have to be shared eight ways when all cores of one node are active. This is another indicator that cache and memory resources can be a limiting factor for the execution of the LU-

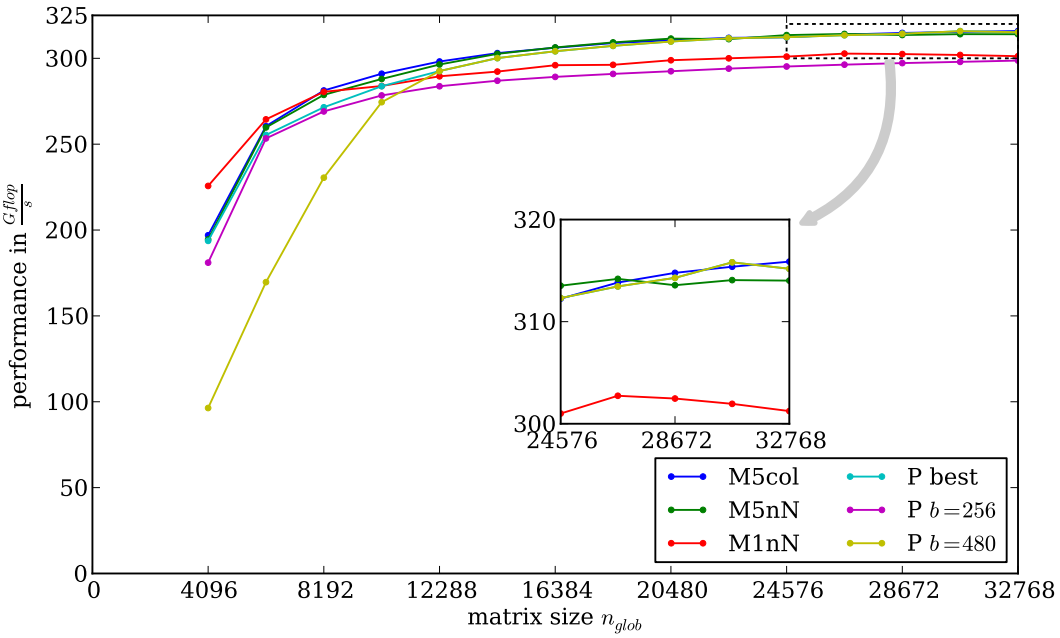


Figure 27: *Performance on IvyBridge16. Comparable, to the measurements on SandyBridge32 we use* $b = 256$.

decomposition. Another reason could be the speed stepping of the CPU. Speed stepping can overclock the CPU as long as it is sufficiently cold. The spread of working cores reduces the temperature development, therefore, enabling higher CPU frequencies.

IvyBridge16 In Figure 27 we can see multiple interesting effects. Sadly especially on big matrices our implementation does not achieve any significant speed ups compared to PLASMA. Both implementations perform around $315\frac{Gflop}{s}$ ($315\frac{Gflop}{s}$ for PLASMA with $b = 480$ and $316\frac{Gflop}{s}$ for M5col). Especially the NUMA-optimization does not change the performance significantly. Accesses to non-local NUMA-nodes seem to be significantly faster on IvyBridge16 than they are on SandyBridge32. The reason for this is that IvyBridge16 consists of two sockets connected by two parallel QPI links. This makes the communication between nodes significantly faster than the communication between the four QPI nodes on SandyBridge32, that are only connected by a ring of QPI links. Therefore, NUMA-effects seem to have no influence on memory access times, making average memory accesses much cheaper.

Cheap memory accesses are probably also the reason why our cache optimizations cannot improve the running time as significantly as on SandyBridge32. A U-task without cache optimization (M1nN) takes an average of $1.69ms$. We could only improve this to $1.63ms$ through our cache optimizations (M5col), even though the number of cache misses per U-task was nearly cut in half (from ~ 9593 with M1nN to ~ 4809 M5col). For a breakdown of the U-task performance on IvyBridge16 see Section A.2.

Haswell24 Even though Haswell24 is a two socket machine like IvyBridge16, we achieve significant speedups on Haswell24 (see Figure 28). On big matrices ($n_{glob} = 32768$) we achieve an improvement of 8% from PLASMA ($b = 320$)

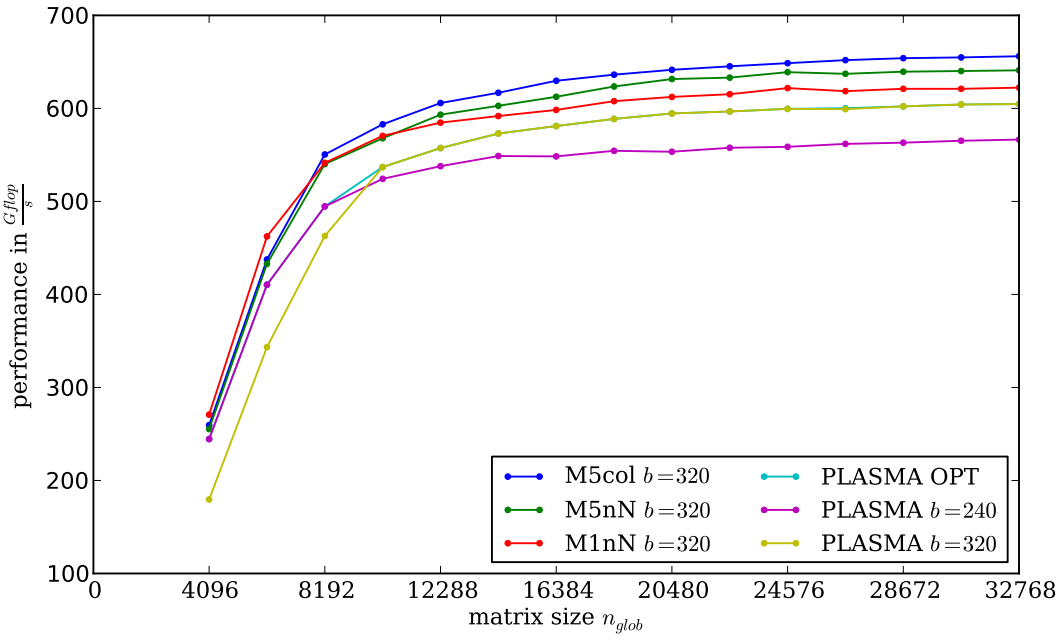


Figure 28: *Performance on Haswell24. On Haswell24 the best parametrization of our implementation is M5col with a tile size of* $b = 320$.

with $605\frac{Gflop}{s}$ to M5col with $656\frac{Gflop}{s}$. The reason for this seems to be that memory accesses and NUMA-effects are expensive relative to floating point operations. Therefore, our optimizations can significantly improve the running time of the decomposition, by reducing the number of memory loads. Haswell24 is the newest and strongest system that we could run tests on. We believe that this performance increase is representative for many future two socket systems with AVX2. Through the increased performance on floating point operations memory accesses become a bottleneck even on two socket machines. For a breakdown of the U-task performance on Haswell24 see Section A.3.

Everything considered, our solution scales well to different hardware and machine sizes. The gains over our competitors depend on the hardware setting. Our gains are significantly larger on machines where memory accesses are expensive compared to floating point operations. This is especially the case on four socket systems and on systems with AVX2.

6.4 Breakdown of our Improvements

In this section, we want to go more in depth on the use of all our improvements. Here, we use the information that we gathered by logging all executions of subtasks.

In Section 6.4.1 we analyze the influence of different task classes on the overall running time, and how the running time composition changes between different scheduling schemes. In Section 6.4.2 and 6.4.3, we analyze the influence of our optimizations on the running time and the number of cache misses of single tasks.

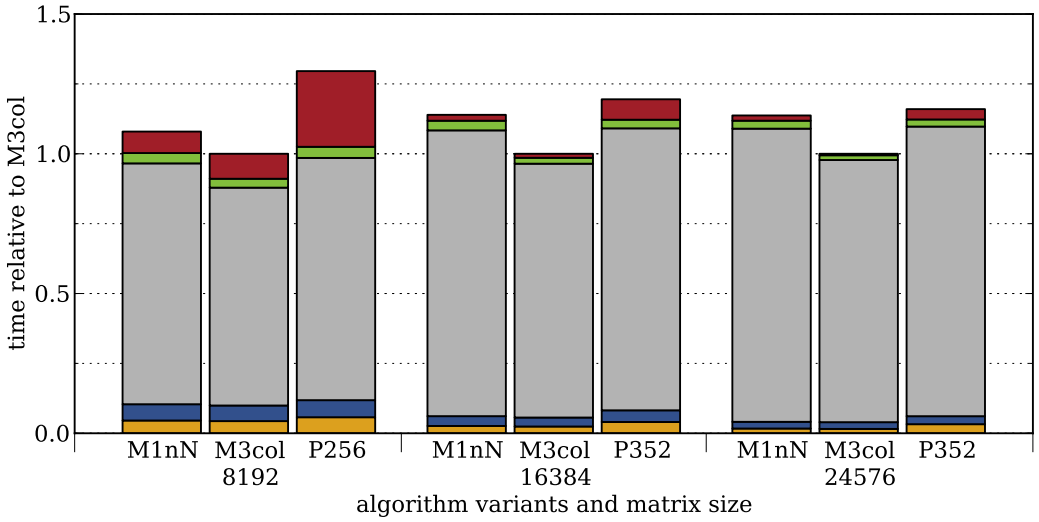


Figure 29: *Relative running time ratios over different matrix sizes (on SandyBridge32). This plot shows the running time of different schedules variations relative to the performance of M3col. It also shows the distribution of the running time in different subtask classes (P-tasks orange, T-tasks blue, U-tasks gray, X-tasks green, and scheduling red).*

6.4.1 The Influence of Different Task Classes

In Section 2.2.3 we stated that U-tasks are the only class of tasks that performs $\mathcal{O}\left(n_{glob}^3\right)$ floating point operations, while P-, T- and X-tasks perform significantly less floating point operations. This was the reason why we focused our improvements on the efficient execution of U-tasks (as stated in Section 3).

Now we want to analyze the running time percentages of each task class. With these percentages, we can prove the hypothesis that the majority of running time is spent executing U-tasks. To compute the running time percentages, we summed up the execution times of all different task classes (weighted with the number of participating cores) and computed their influence on the total execution time. The result of this analysis can be seen in Figure 29. Especially when decomposing big matrices, U-tasks clearly dominate the running time. On a matrix with $n_{glob} = 32768$, U-tasks make up around 95% of the execution time (see Table 2). Even on small matrices, U-tasks contribute more than 75% of the total running time (e.g. 77% for M3col on $n_{glob} = 8192$). This validates our decision to target U-tasks with our optimizations.

Figure 30 and Table 2 show a compilation of different scheduling variations on a constant matrix size of 32768. With the help of this diagram and the corresponding table, we can analyze the influence of our optimizations on different task classes. One can see that our implementation uses less time for the panel factorization (P-tasks). We believe the reason for this is that we use less processors for the panel factorization. This reduces the high synchronization overheads that are necessary for the parallel panel algorithm. As described in Section 5.4, this also reduces the time that is spent inside a panel task waiting for all other participating processors to begin working on that panel task. Using less processors has no impact on the utilization of cores because there are enough

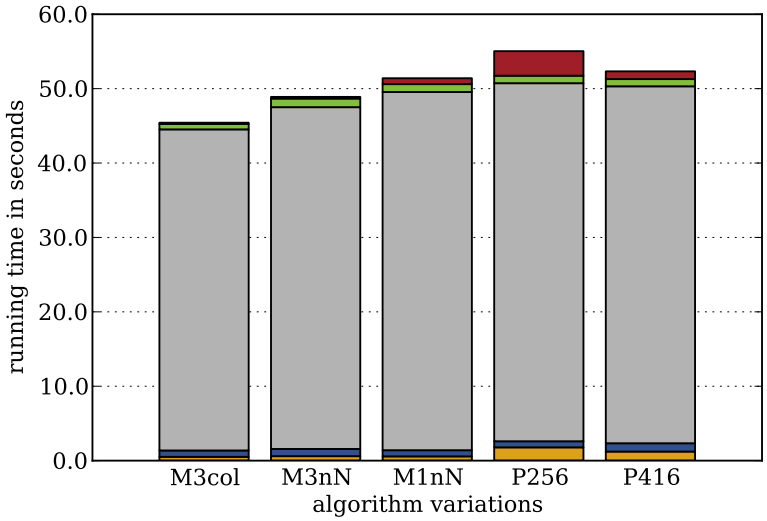


Figure 30: *Running time percentages of different task classes (on SandyBridge32). This plot shows the total running time of different variations on a matrix with $n_{glob} = 32768$ ($b = 416$ is PLASMA's optimum) and its distribution on different task classes (P-tasks orange, T-tasks blue, U-tasks gray, X-tasks green, and scheduling red).*

tasks such that each processor that does not participate in panel tasks can still work on other tasks. It is also recognizable that X-tasks profit from the NUMA-optimizations made in M3col. They are accelerated by over 30% (from 1.06s for M1nN to 0.73s for M3col). X-tasks exchange rows of the matrix, therefore, they access "random" lines within one column. Our NUMA-optimization ensures that all lines can be read from the local memory, thus accelerating their access times. It is interesting that T-tasks do not profit from NUMA-optimizations as much as X-tasks, even though they also exchange matrix lines within one column. Additionally to the improved subtask executions, meta-tasks can also reduce the scheduling overhead significantly. The time spent scheduling tasks is reduced by nearly 80% (from 0.79s for M1nN to 0.16s for M3col).

The most visible increase in performance, however, was achieved in the execution of U-tasks. While the relative performance increase is only around 10% (48.1s for M1nN to 43.2s for M3col) the absolute reduction in running time is greater than the complete running time of all other task classes combined. This performance increase is the core of this thesis. Therefore, we use the following sections to analyze the influence of our optimizations on the running time of single U-tasks.

6.4.2 Cache Performance

In this section, we want to analyze the effectiveness of the cache optimization that we described in Section 3.3. Its purpose is to reduce the number of expected cache misses during the execution of U-tasks. To be able to make predictions on the number of cache misses, we designed a theoretical tile-cache model (see Section 3.1). Within this model, we achieved significant cache benefits by grouping tasks according to their input tiles and executing all tasks of one group on

	P-tasks	T-tasks	U-tasks	X-tasks	scheduling	total
M3col	0.51s 100%	0.85s 100%	43.16s 100%	0.73s 100%	0.16s 100%	45.42s 100%
M3nN	0.60s 116%	0.97s 113%	45.94s 106%	1.16s 158%	0.22s 134%	48.89s 107%
M1nN	0.57s 111%	0.83s 97%	48.14s 111%	1.06s 144%	0.79s 486%	51.40s 113%
PLASMA ($b = 256$)	1.78s 347%	0.82s 96%	48.13s 111%	1.01s 137%	3.30s 2022%	55.04s 121%
PLASMA ($b = 416$)	1.22s 237%	1.11s 130%	47.98s 111%	0.97s 132%	1.04s 639%	52.33s 115%

Table 2: *Breakdown of the running times of different task classes on a matrix with $n_{glob} = 32768$ ($b = 416$ is the optimal tile size for PLASMA on this size). We computed the running time percentages by adding all execution times of single tasks and dividing that number by 32 (the number of cores on SandyBridge32). The scheduling time is the total time used for the computation minus the time of all task executions. The percentage number is computed relative to the corresponding times of M3col.*

the same node at approximately the same time. In Section 3.5, we decided to group tasks which work on one "meta-tile" of the matrix. This way, input tiles can be shared through the L3 cache. In Section 3.3, we describe that executing a 3×3 U-meta-task would only read 15 different input tiles. This is a reduction by 12 tiles from approximately 27 tiles that have to be read when executing 9 unrelated tasks (a reduction by $\frac{4}{9}$).

Now, we want to see how this optimization holds up during practical experiments. In Figure 31 we plot the number of cache misses that happend during a U-tasks execution against its running time (for different scheduling variants). We do this with a 2D-histogram plot. In this plot, tasks are distributed on bins depending on the number of cache faults and their execution time. Each bin is 100 cache misses wide and 0.01 milliseconds tall. The color of a point (x, y) represents how many U-tasks had x cache misses and took y milliseconds.

These plots show that during the cache optimized scheduling variants (M3nN and M3col in Figure 31) U-tasks cause significantly less cache misses than U-tasks of the unoptimized version (M1nN in Figure 31). One can also see that the U-tasks of the cache optimized versions can be categorized into two general groups. There is one group of U-tasks – consisting of approximately $\frac{2}{3}$ of all U-tasks. All U-tasks of this group cause less than 8000 cache misses and most of them cause around 2000-4000 cache faults.

The U-tasks of the other group cause approximately as many cache misses as U-tasks during the unoptimized variant M1nN. The reason for this is that when a node begins executing the tasks of one meta-task, then the tiles that are read by the first task might not already be in the cache. Therefore, this task does not profit from any cache optimizations. It likely causes the same amount of cache hits and misses as an unoptimized U-task. Only the later tasks within one meta-task are able to profit from our optimizations by reading tiles that

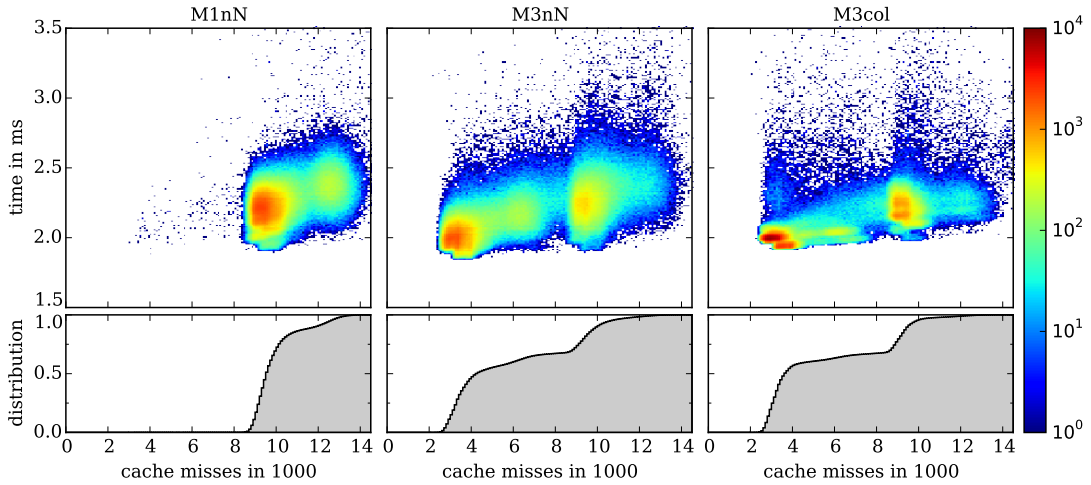


Figure 31: *U-task cache misses versus execution time as density map. All U-task executions during the factorization of a matrix with $n_{glob} = 32768$ (690880 U-task executions). Each bin is 100 cache misses wide and 0.01 milliseconds tall.*

M3col	1.999ms	5475
M3nN	2.128ms	5901
M1nN	2.230ms	9941
PLASMA ($b = 256$)	2.229ms	9274
PLASMA ($b = 416$)	9.522ms	53373
(norm.)	2.219ms	12438

Table 3: *Average running time of one U-task compared to its average number of cache misses (averaged over all U-task executions on a matrix with $n_{glob} = 32768$). For PLASMA with $b = 416$ we computed the normalized time and cache misses (by computing the amount of time/cache misses per Gflop)*

preceding tasks already loaded into the cache.

We can also see from these plots that the running time of a U-task depends on its number of cache misses. A U-task out of the first group (less than 8000 cache misses) takes about $0.2ms$ less time to compute than a U-task of the second group (M3col: $0.21ms = 2.14ms - 1.93ms$ and M3nN: $0.23ms = 2.28ms - 2.05ms$). In Section 6.5, we try to explain this effect, using the theoretical cache model.

6.4.3 NUMA-Optimization

In this section, we analyze the influence of our NUMA-optimization on the execution time of U-tasks. Our NUMA-optimization consists of storing the matrix distributed meta-column wise on all NUMA-nodes and executing meta-tasks preferably on the node that holds the corresponding column (see Section 3.4 and 3.5). In Figure 30 and in Table 2, we can see that U-tasks profit from being executed that way.

In Table 3 and in Figure 31, we see that the number of cache misses does not change significantly through the NUMA-optimization (compare M3nN and

M3col). Instead, the influence of a cache miss on the running time of a U-task is shrinking between M3nN and M3col. For M3col in Figure 31 there are areas where the influence of cache misses on the average running time is very weak (nearly constant running time for increasing number of cache misses).

Another effect that can be witnessed, looking at the execution of U-tasks during M3col, is that explicit NUMA-scheduling can reduce the variance in running times. Since every execution happens on the NUMA-node that stores two of three input tiles, there is at most one tile that has to be loaded from another NUMA-node. Interestingly, the running time difference between U-tasks that profit from the cache optimization and those that do not remains. Even the absolute running time difference between the first and the second group is nearly the same in M3nN as in M3col.

6.5 Using the Tile-Cache Model to Analyze Experiments

In Section 3.1 we described the tile-cache model that we designed to represent the way shared caches work during the computation. This cache model was motivated by the assumption that L3 hardware caches store full matrix tiles, which simplifies cache behavior significantly.

The tile-cache model was the basis for our optimizations. In Section 6.4.2 we proved that our cache optimization significantly reduced the number of cache misses during the execution of U-tasks. This is a first indication for the accuracy of our tile-cache model. In this section, we want to support the tile-cache model with additional experimental data. Then, we will try to interpret the experimental data with the help of the tile-cache model.

Evaluation of the Tile-Cache Model Before analyzing experimental data with the theoretical tile-cache model, it is important to know how many tiles can fit into the shared tile-cache. Therefore, we have to analyze the physical cache memory of the machine. On SandyBridge32 the L3 hardware caches have a size of $20MB$ (L1 and L2 caches are not shared; Additionally, L1 and L2 are inclusive which means that they cannot store anything that is not already stored in the L3 cache). The matrix is distributed in tiles of 256×256 double precision floating-point numbers, which have $512KB = \frac{1}{2}MB$. Therefore, in theory, 40 tiles can fit into the shared cache simultaneously.

By analyzing the MKL's memory allocations, we found that each core executing MKL BLAS routines reserves enough memory to store two whole matrix tiles. We believe that this memory is used to copy input tiles and to store intermediary results. As it is constantly used, it remains cached and reduces the memory that is available for data sharing between cores. Thus, in reality the maximum number of tiles within each tile-cache is lower than 40. We estimate that the shared tile-cache on SandyBridge32 can hold approximately 24 full tiles.

To analyze experimental data we programmed a simulator for our tile-cache model. This simulator stores the contents of all tile caches (one per node). Given a task and the corresponding core/node the simulation can check if the tile accesses inside the task would have been cache hits. Afterwards, the simulation computes the changes that the task would have on the contents of the tile-cache. To analyze the practical execution, one simulates all task executions in the order given by the executions log (ordered by starting time). With the help of this

simulation we find out which input tiles of each task execution would have been tile-cache hits or misses (within the theoretical model).

With this data we first analyze the validity of our model. We believe that our model can predict the number of actual cache misses and the running time of a U-task by analyzing which tasks were executed before it. Therefore, we use the simulation data to categorize the U-tasks depending on the expected tile-cache hits that they caused during our simulation. Each U-task $U_{ij}^{(k)}$ computes $A_{ij}^{(k)} = A_{ij}^{(k-1)} - A_{ik}^{(k)} \cdot A_{kj}^{(k)}$, where $A_{ik}^{(k)}$ $(= L)$ is a tile within the panel, and $A_{kj}^{(k)}$ $(= H)$ is a tile at the top of the matrix. We distribute all U-tasks into the following categories (represented in Figure 32): (1) as the U-task begins to be executed, both, the corresponding panel tile and the corresponding tile at the top of the matrix are already loaded into the tile-cache (see top-middle), (2) only the panel tile L is cached (see top-right), (3) only tile H at the top of the matrix is cached (see bottom-left), and (4) neither of both tiles is cached within the simulated tile-cache (see bottom-right). The tile $A_{ij}^{(k-1)}$ is not considered for this categorization because even in the cache optimized version there are practically no tile-cache hits for this tile (only one U-task per iteration works on any specific tile).

In Figure 32 one can clearly see that these four groups are discernible by their position within the density map. This means that our theoretical cache model can "predict" the running time and the number of physical cache misses of a U-task by analyzing preceding task executions. This is a strong sign that our cache model represents the behavior of the L3 cache during the execution of the algorithm.

Using the Tile-Cache Model to Explain Running Time Through the distribution of U-task executions into four groups we can explain some specific effects. In Section 6.4.2 we pointed out that we can divide all U-tasks into two groups, one group with all U-tasks having less than 8000 cache misses and the other with all U-tasks having more than 8000 cache misses. Now we can see that this categorization effectively separates groups (1) and (2), from groups (3) and (4). Which means that practically all tasks below 8000 cache misses operate on a cached panel tile L. We can discern that the number of cache misses and the running time of a U-task is much more dependent on the cache status of L than the status of H. We believe the reason for this is the way in which tile L is accessed. It is the first tile of the multiplication, therefore, it is generally accessed row-wise. Row-wise access is bad considering that the tile is stored in a column major format. Reading a row equates to reading every 256-th value of an array $(b = 256)$. Therefore, one only reads one element per cache line which makes it hard for the prefetcher to keep up.

Another reason why tile-cache misses for tile L are worse for the running time than tile-cache misses for tile H lies in the NUMA-optimization. In version M3col, tile H can always be read locally while tile L is not guaranteed to be local (only $\sim \frac{1}{4}$ of all accessed L tiles is local). This is the reason why in M3col group (2) is only 0.06ms slower than group (1), while in M3nN group (2) is 0.15ms slower than group (1) (0.07ms between group (3) and group (4) in M3col, and 0.13ms between group (3) and group (4) in M3nN)

An interesting fact that can be observed through the analysis of the tile-cache

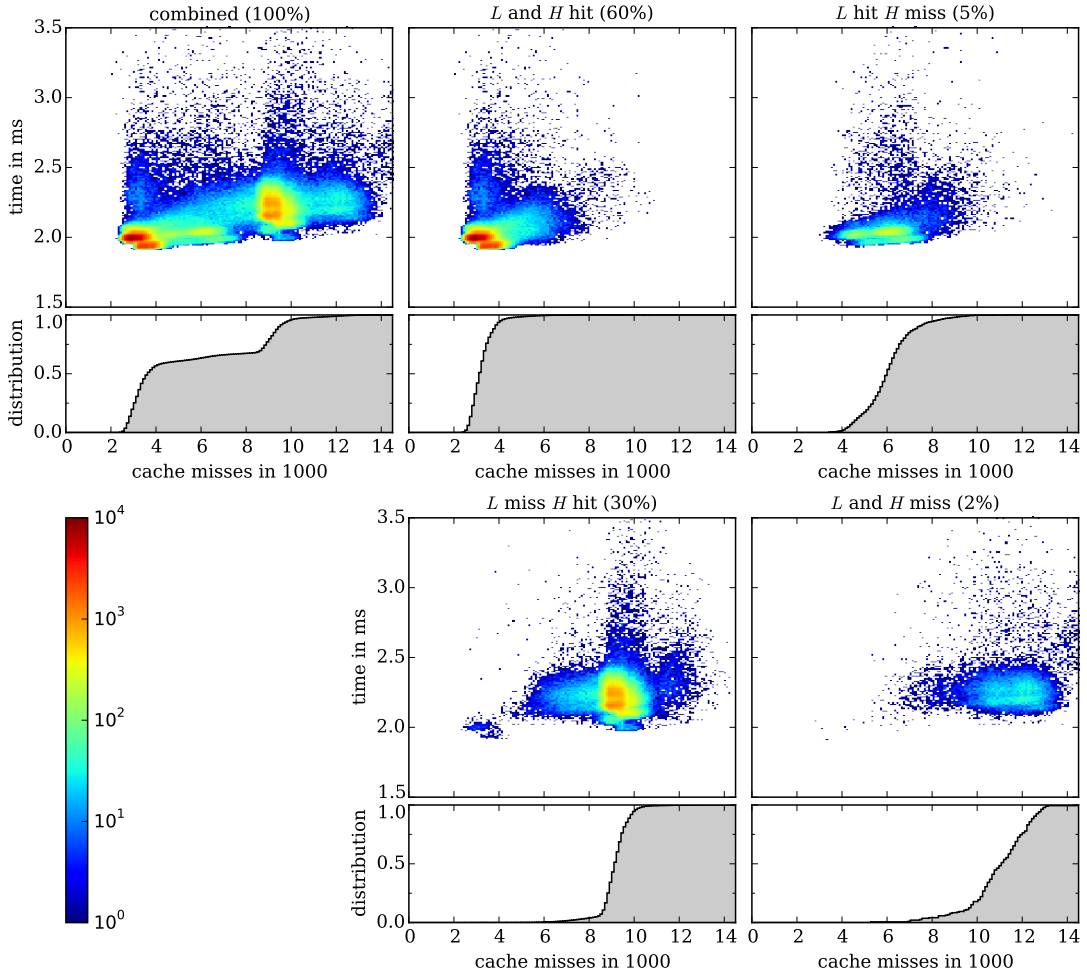
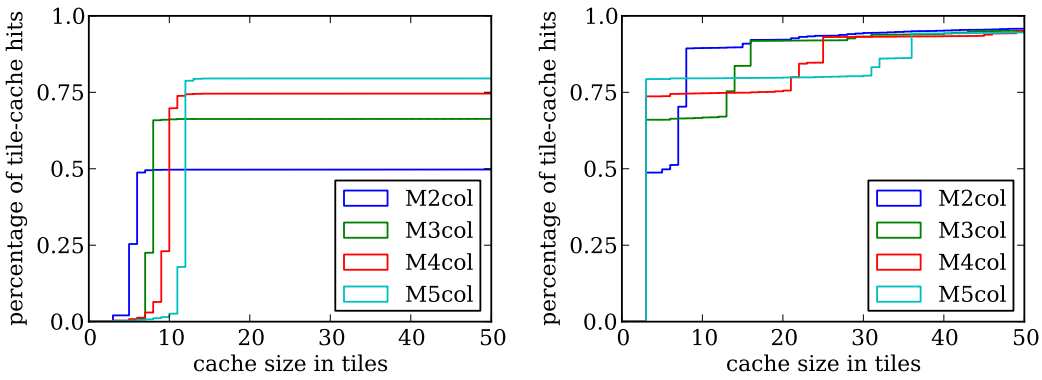


Figure 32: *U-tasks categorized with the tile-cache model. For this figure we categorized all U-tasks of one M3col execution ($n_{glob} = 32768$) into four groups depending on the position of their input sizes within our cache simulation (expected tile-cache size 24 tiles).*

model is that there are a lot more tile-cache hits for tile H than there are tile-cache hits for tile U. The tasks within one meta-task are unpacked in a column major order. In Section 4.3 we analyzed that during the execution of a solitary U-meta-task there should be four tasks in group (1), two tasks in groups (2) and (3) respectively, and one task in group (4). For the access on tile L this prediction is correct. About $\frac{2}{3}$ of all accesses to the panel tile are cache hits. But for the access to tile H we reach a $\frac{9}{10}$ hit rate. The reason for this is that tiles can be shared between meta-tasks. If two subsequent meta-task are somewhere within the same meta-tile-column, the corresponding H tiles could still be in the tile-cache and, thus, cause tile-cache hits. As we mentioned in Section 4.3, the probability that two consecutive meta-tasks operate on the same meta-tile column is reasonably high because two U-meta-tasks that operate on the same column (during the same iteration) have the same priority and become ready simultaneously. This maximizes the chance that they are executed consecutively. During the NUMA-optimized variant they are also scheduled to the same node (preferably), further increasing the chance for possible tile reuses.



(a) *Tile-cache hits accessing L* (b) *Tile-cache hits accessing H*

Figure 33: *Number of tile-cache hits on varying tile-cache sizes. Interestingly, this plot is nearly independent from the tile/matrix size, as the actual tile size only influences the number of tiles that can be cached.*

Optimizing Meta-Tile Size For data reuse to work, the cache has to be big enough to hold all tiles that are read between the reading of a tile and its reuse. In Figure 33 we show how many tile accesses are considered a cache hit over a varying tile-cache size. To compute these plots we used an execution order taken from the experiments measured on a matrix with $n_{glob} = 32768$ and computed the number of tile-cache hits/misses with the help of our simulation.

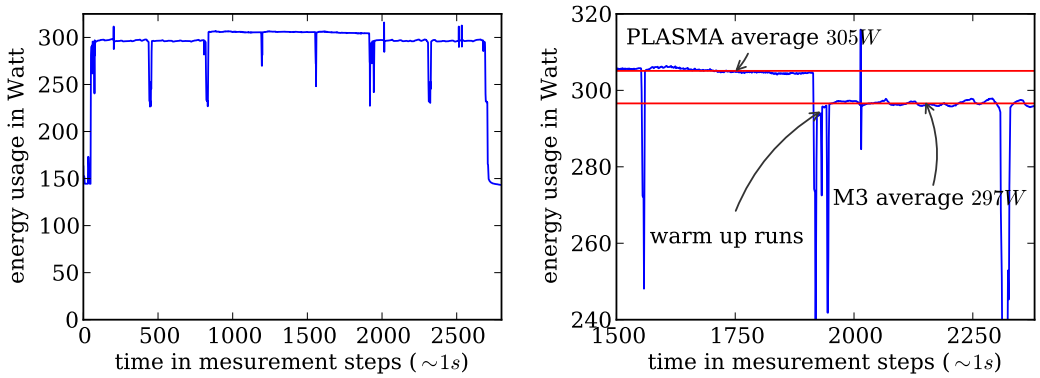
From these plots one can extrapolate how many tiles the cache should be able to hold to properly use certain meta-tile sizes. In Figure 33a there is only one big gain in the number of tile-cache hits. This gain is around $\textit{cache-size} = 2 \cdot m + 1$. From that cache size on around $\frac{m-1}{m}$ of all accesses to tile L are cache hits. In Figure 33b, there are two distinct gains of tile-cache hits. The first rise happens very early at a tile-cache size of three tiles (independent from the meta-tile size). The reason for this is that two consecutive subtasks within one meta-task share the same H tile. This rise is also around $\frac{m-1}{m}$ of all executions high. The second rise happens when the tile-cache is big enough to hold all tiles that are read during the execution of one meta-task. This is the point at which tiles can be shared between different meta-tasks. Interestingly, the number of tile-cache hits after that second rise is practically independent from the meta-tile size.

6.6 Different Performance Metrics

In this section, we want to analyze the performance of our implementation with other performance metrics (apart from execution time and cache behavior). We already saw that our optimizations achieves improved running times and that it greatly reduces the number of cache faults. In this section, we want to talk about the power usage and about the CPU utilization.

6.6.1 Power Consumption

Programmers have very little influence on the power usage of algorithms. Usually, there are only two ways to influence the power needed for a computation. The first option is to adapt the algorithm to a power efficient setup. For example, by



(a) *Complete measurement* (b) *Close up on two measurements*

Figure 34: *Power measurements. To make sure that the power usage is not heat (time) dependent, we measured two factorizations with M3col, then three with PLASMA* $b = 512$ *(one as warm up), and then another two factorizations with M3col. For this measurement, we factorize huge matrices (*$n_{glob} = 65536$*) to allow the energy usage to settle.*

using a machine that has more energy efficient processors or a different cooling system. The second option is to increase the performance, thus, reducing the time needed for the computation instead of the power usage per time.

We want to show that the power consumption of a computer can be changed by the computation that is executed. We believe that the power consumption can be reduced by reducing the number of cache misses.

To measure the amount of power that is needed for the computation, we use a "watt's up? PRO" power measurement tool. This tool is plugged in between the electrical outlet and the power supply unit of the computer. It logs the power usage in regular intervals ($\sim$ 1 second). For the power usage to settle, we used big input matrices with $n_{glob} = 65536$. On SandyBridge32 we had some problems with the fan regulation. During the computation, the fans would accelerate and disrupt the measurement. To reduce this problem, we turned the fans to maximum ventilation for the whole duration of the computation. With these settings, we measured a reduction of around $30W$ (4% ; $820W$ PLASMA with $b = 512$ and $790W$ for M3col). The results on IvyBridge16 can be seen in Figure 34. On Ivybridge, we measured a power reduction by $8W$ (3%; $305W$ PLASMA $b = 512$ and $207W$ M3col) between M3col and PLASMA.

It is interesting that even on IvyBridge16 – where we could not achieve better running times through our cache optimizations – we can save power by scheduling tasks in a way that reduces cache misses.

6.6.2 Theoretical Upper Bounds

Theoretically, SandyBridge32 has a peak performance of $614.4\frac{Gflop}{s}$. With our solution we achieve $517\frac{Gflop}{s}$, which is 84% of the peak performance of SandyBridge32 (Theoretically, even more than 100% might be possible through speed-stepping). The question is how much more performance is possible to achieve with this numerical algorithm and the BLAS libraries available. A first estimate

could be the performance of a single core with the same algorithm. A single core on SandyBridge32 has $2.4 GHz$ and can execute 4 additions and 4 multiplications per cycle (through AVX). Therefore, it has a peak performance of $19.2 \frac{Gflop}{s}$. In our scalability tests a single core reached $18.7 \frac{Gflop}{s}$ which is exactly 97.5% of its peak performance.

Estimating from one core to 32 cores can be problematic since there are many resources that are shared between cores. For example, the memory bandwidth, and the cache system, but also the cooling capacity. A single core can profit much more from the turbo mode's overclocking than all cores working together. Therefore, we want to estimate the theoretical performance peak through a measurement made within a bigger, more realistic computation. Hence, we look at the performance of U-tasks. A U-task that has a tile-cache hit on both of its input tiles (see Section 6.5) has an average running time of $1.92 ms$. It computes $256^3 \cdot 2 flop$ (256^3 multiplications and additions). Therefore, it has a performance of $17.4 \frac{Gflop}{s}$ which is around 90.6% of the peak performance. Considering that there are also other kinds of (less efficient) subtasks and that in praxis not every U-task can have a cache hit, we believe 84% is a satisfactory result.

7 Conclusion

7.1 Overview

Our goal was to use scheduling techniques to improve the performance of the LU-decomposition. The submatrix update, which is the most work intensive step during the LU-decomposition, has long been considered to be compute bound. But modern processors have become faster and faster. With AVX and AVX2, processors can compute many computations per cycle. The improvements in memory bandwidth and latency on the other hand have not been as drastic. Because of NUMA-effects, memory accesses have the potential to be even more costly. Through all these effects the computation of the LU-decomposition can become memory bound, especially on four socket machines. To counteract this problem and reduce the influence that memory accesses have on the running time of computations, manufacturers build bigger and bigger cache hierarchies. Optimally using these cache hierarchies has, therefore, become an important and difficult task for programmers.

Our approach is to group tasks that share common input tiles into meta-tasks. Then we schedule these meta-tasks to nodes such that the common input tiles can be shared through the shared L3 cache. To do this, we use a two level hierarchical scheduler. On the global level, we schedule the meta-tasks to nodes. On the local level, we unpack the grouped subtasks and schedule them with a local FIFO queue. This way, we ensure that grouped tasks are executed in parallel. Each input tile is loaded by the first subtask accessing it. Afterwards, all subsequent tasks can read the already loaded tile from the shared cache. Using this method we can nearly cut in halve the number of cache misses during the computation (from 9941 per U-task on average using M1nN to 5475 per U-task on average using M3col).

Additionally, we propose a NUMA-optimization. For this optimization, we

distribute the matrix between all NUMA-nodes meta-column wise. Then we schedule meta-tasks such that each node preferably works on tasks whose input tiles are stored in the local memory, accelerating loading times of tiles that are not already cached. This optimization is especially effective on four socket machines like SandyBridge32, where it improves the average running time of U-tasks even more than the cache optimization (from 2.13ms with M3nN to 2.00ms with M3col).

Through the combination of these techniques we improved the performance of the current state of the art implementation by up to 29% (on small matrices $n_{glob} = 8192$) and by at least 15% (on big matrices with $n_{glob} = 32768$). This performance increase is very significant, considering that we perform the very same numerical operations as the implementation of our competitors, which is already very optimized. Our implementation reaches up to $517\frac{Gflop}{s}$, which is over 84% of our machines peak performance.

7.2 Future Work

In the beginning of this thesis we chose the LU-decomposition for our research because it is characteristic for many numerical workloads. Hence, it is logical to generalize the concepts developed within this thesis to other similar computations. Especially the tile-cache model and the our cache optimization can easily be adapted for other numerical algorithms. The general concept of grouping subtasks by their data dependencies and scheduling them accordingly, has applications in many other scheduling scenarios.

As we have seen in Section 6.6.2, there is still a little room for improvement within the LU-decomposition. Some approaches that might have potential for further improvements are non-square meta-tasks and other NUMA-distributions. Especially, if the LU-decomposition is only one algorithm within a bigger application, it might be important to fit the NUMA-distribution to the needs of other parts of the application.

References

[1] Emmanuel Agullo, James Demmel, Jack J. Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series*, 180(1):012037, 2009.

[2] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammerling, Alan McKenney, and Danny Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] Michael Bader and Christoph Zenger. Cache oblivious matrix multiplication using an element ordering based on a peano curve. *Linear Algebra and its Applications*, 417(2–3):301 – 313, 2006. Special Issue in honor of Friedrich Ludwig Bauer.

[4] Susan Blackford and Jack J. Dongarra. Lapack working note 41 installation guide for lapack. 1999.

[5] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack J. Dongarra. Dague: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1–2):37 – 51, 2012. Extensions for Next-Generation Parallel Programming Models.

[6] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack J. Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1432–1441, May 2011.

[7] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009.

[8] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 105–115, New York, NY, USA, 2007. ACM.

[9] Jack J. Dongarra. Performance of various computers using standard linear equations software. 2014.

[10] Jack J. Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Achieving numerical accuracy and high performance using recursive tile lu

factorization with partial pivoting. *Concurrency and Computation: Practice and Experience*, 2013.

[11] Jack J. Dongarra, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, Wesley Alvaro, Mathieu Faverge, Azzam Haidar, Joshua Hoffman, Emmanuel Agullo, Alfredo Buttari, and Bilel Hadri. Plasma version 2.6.0 user guide. http://icl.cs.utk.edu/plasma. 2013.

[12] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[13] Alexander Heinecke and Michael Bader. Parallel matrix multiplication based on space-filling curves on shared memory multicore platforms. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, MAW '08, pages 385–392, New York, NY, USA, 2008. ACM.

[14] Jakub Kurzak, Hatem Ltaief, Jack J. Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010.

[15] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999.

[16] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[17] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. Van De Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3):14:1–14:26, July 2009.

[18] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 47–58, New York, NY, USA, 2007. ACM.

A Density Maps for other Parametrizations

A.1 SandyBridge32

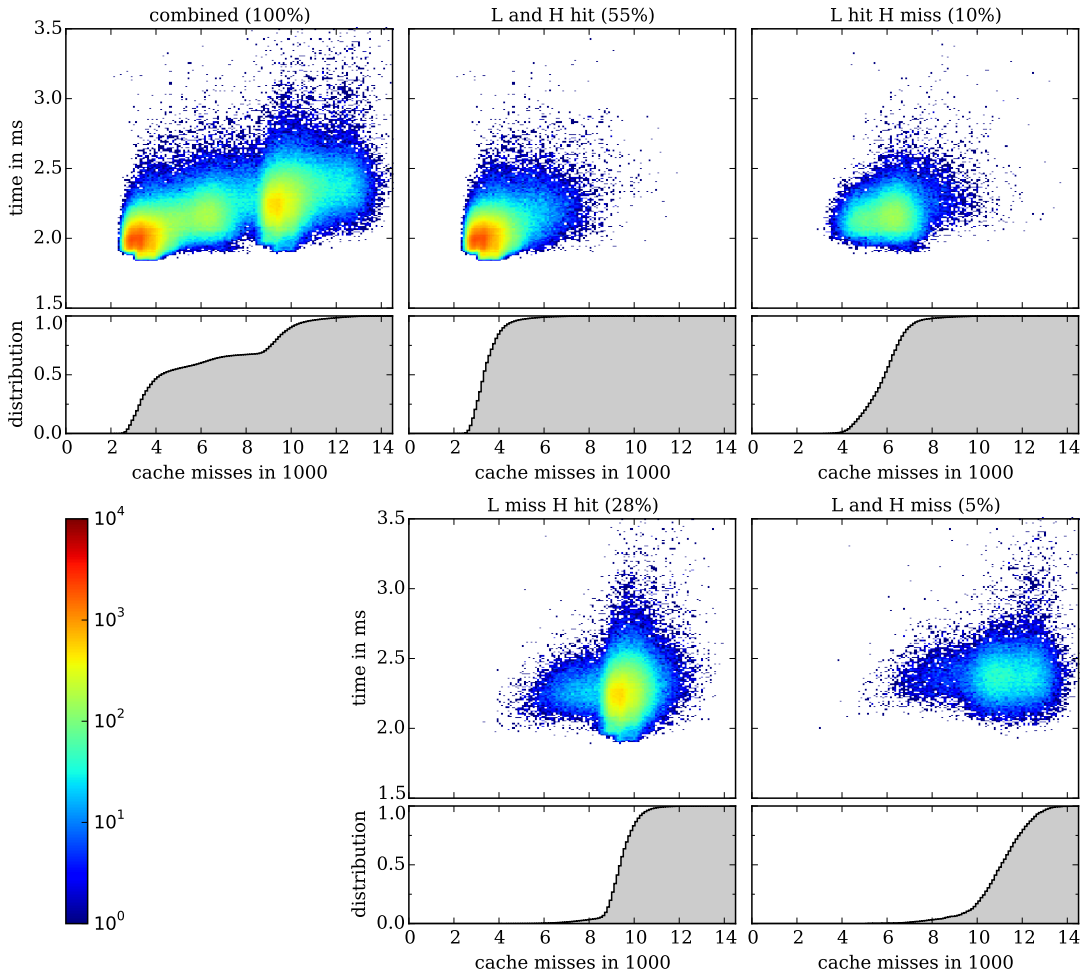


Figure 35: *Density map for M3nN. This plot shows all U-tasks of one M3nN execution on a matrix with* $n_{glob} = 32768$ *and* $b = 256$*, separated with a simulated cache size of 24 tiles.*

Figure 35 shows that using the simulation, described in Section 6.5, we can separate all U-tasks, during the execution of M3nN, into four groups, according to their tile-cache hits. Comparable to M3col these groups each have a distinct position within the density map (distinct combination of cache misses and running time).

When we compare the plots between M3col and M3nN we see:

- the running time variance is greater in M3nN
- the number of cache misses per group is comparable
- in M3nN tile-cache misses accessing the H tile are worse, than they are during M3col

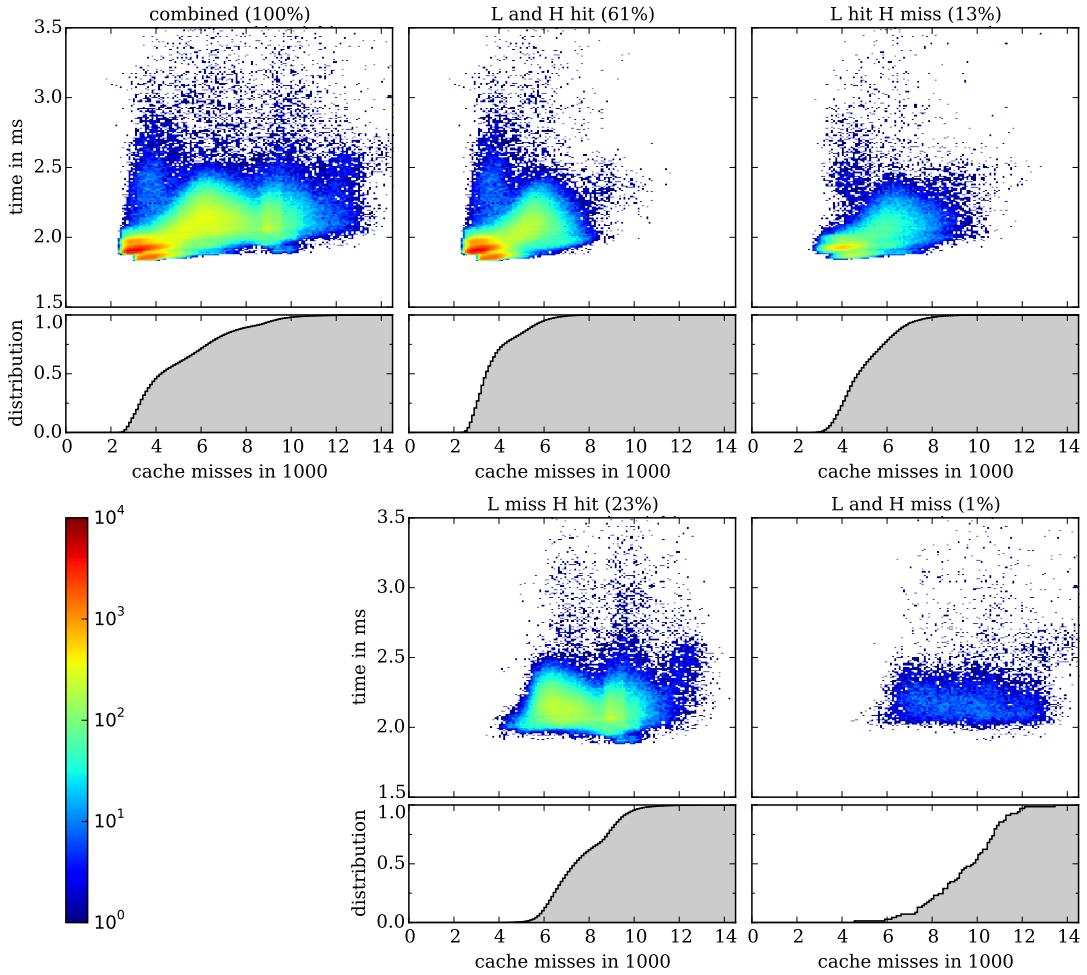


Figure 36: *Density map for M4col. This plot shows all U-tasks of one M4col execution on a matrix with* $n_{glob} = 32768$ *and* $b = 256$*, separated with a simulated cache size of 24 tiles.*

In Section 6.3.2 we discovered that a meta-tile size of four leads to an unusually bad running time. Interestingly, Figure 36 shows that the classification computed with our simulation is not distinct. There is a number of U-tasks that has around 6000 cache misses and takes around $2.2ms$. These U-tasks are distributed onto multiple different groups.

Our assumption is that two tasks, within one meta-task, that share a common L tile are executed nearly simultaneously. Our simulation discretizes the execution, therefore, we assume that the first U-task produces a tile-cache miss, and the second U-task produces a tile-cache hit. In reality this might not be the case. The task that is started "later" might overtake the original task, thus, the two tasks are sharing their cache misses. Both tasks have to wait for the task, to be loaded into the L3 cache.

To verify this thesis, we looked into the logs of the execution and found, only tasks within the first and second tile-column of a meta-tile belong to this group. All tasks within the third and fourth tile columns have regular tile-cache hits on tile L. We also found that whenever a task belongs to this group its neighbor there is a neighbor, that also belongs to this group.

It is interesting that this synchronization happens only on 4×4 meta-tasks. The reason for this might be that in this scenario the number of tasks (16) within one meta-task is a multiple, of the number of cores on one node. A sign for this theory might be that this effect does also happen on IvyBridge16, where there are also 8 cores per node, but the effect does not happen on Haswell24, where there are 12 cores per node.

A.2 IvyBridge16

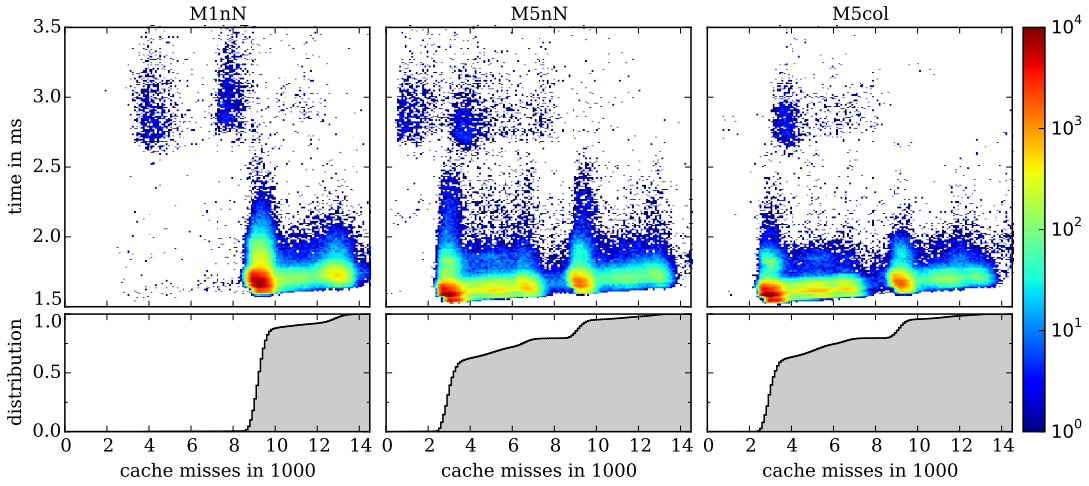


Figure 37: *Density map for different variations on IvyBridge16*

In Figure 37 we see that the execution time of a U-task is practically independent from the number of its cache misses (nearly no slope). Comparable to our tests on SandyBridge32 there is a group of U-tasks that cause less than 8000 cache misses. Thus, the cache optimization seems to work as expected. As on SandyBridge32 this group consists of all U-tasks that have a tile-cache hit when accessing the panel tile (L).

Interestingly, on IvyBridge16 there is a small number of U-tasks, that take significantly longer to execute than an average U-task, even though they have even less cache misses. This has not been the case on other machines.

A.3 Haswell24

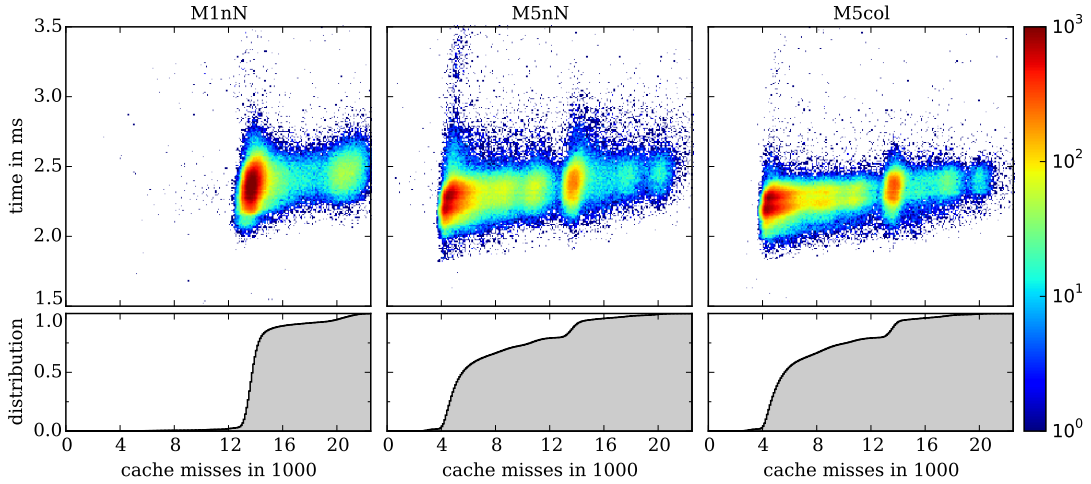


Figure 38: *Density map for different variations on Haswell24 (with $n_{glob} = 32768$ and $b = 320$). As 32768 is no multiple of 320, there are cutoffs on the rightmost tile-column and the bottom tile-row. Within the plot we omitted U-tasks that operated on tiles with cutoffs to show the running time of average U-tasks.*

In Figure 38 we see that the cache optimization also works on Haswell24. It reduces the number of cache faults significantly. Comparable to IvyBridge16, the correlation between cache misses and running time is relatively weak. Haswell24 has more cache misses per task, because the loaded tiles are significantly larger ($b = 320$) compared to the tiles used on SandyBridge32 and IvyBridge16.

The NUMA-optimization does not change the running times of U-tasks significantly. It only improves the variance of running times.