Node-Level Performance Modeling of Sparse Factorization Solver

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Radim Janalík

under the supervision of
Prof. Olaf Schenk

August 2021

Dissertation Committee

Prof. Illia Horenko	Università della Svizzera italiana, Switzerland
Prof. Igor Pivkin	Università della Svizzera italiana, Switzerland
Prof. Harald Köstler	Friedrich-Alexander University Erlangen-Nurnberg, Germany
Prof. Tomáš Kozubek	Technical University of Ostrava, Czech Republic

Dissertation accepted on 4 August 2021

Research Advisor	PhD Program Director
Prof. Olaf Schenk	**Prof. Walter Binder** **Prof. Silvia Santini**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Radim Janalík
Lugano, 4 August 2021

Abstract

Solving large sparse linear systems is at the heart of many application problems arising from scientific and engineering problems. These systems are often solved by direct factorization solvers, especially when the system needs to be solved for multiple right-hand sides or when a high numerical precision is required. Direct solvers are based on matrix factorization, which is then followed by forward and backward substitution to obtain a precise solution. The factorization is the most computationally intensive step, but it has to be computed only once for a given matrix. Then the system is solved with forward and backward substitution for every right-hand side. Performance modeling of algorithms involved in solving these linear systems reveals the computational bottlenecks, which can guide node-level performance optimizations and shows the best performance that can be achieved on given architecture.

In this thesis we investigate and analyze the performance of the forward/backward solution process of the PARDISO direct sparse solver and present a detailed performance analysis for its sparse solver kernel. This analysis is based on the Berkeley roofline model, a model that is widely used to predict the upper bound of a code based on processor peak performance and memory bandwidth. We establish a modified roofline model that captures the serial and parallel execution phases which allows us to predict the in-socket scaling over the processor cores. The distinction of serial and parallel execution is important as the amount of the serial fraction depends on the matrix used and can have a significant negative impact on performance. We compared the roofline model with an alternative Erlangen ECM model and provide discussion on usability and modeling capabilities of both models. The model predictions are compared with various measurements for a representative set of sparse matrices on different x86_64 processors. The performance analysis and modeling performed in this work are limited to a single node, however, the code considered here is also a building block for the MPI parallel version. Hence, also the distributed memory implementation of PARDISO will profit from any enhancement achieved.

Acknowledgements

First of all, I would like to thank my advisor, Prof. Olaf Schenk, for all his guidance, invaluable support, and patience. I am also very grateful to my friend Dr. Juraj Kardoš for his valuable advice and help. Without them, the completion of my dissertation would not have been possible.

Thanks to all the committee members for their review and suggestions to make this work better. Thanks to Dr. Georg Hager, from whom I learned a lot, and people from IT4Innovations for their hospitality during my stay. Special thanks to organizers of the CSCS Summer School. I was part of it since the beginning of my doctoral studies and it was always an interesting experience.

This research was partially supported by Swiss National Science Foundation (SNSF) through the German Priority Programme (DFG) 1648 "Software for Exascale Computing" (SPPEXA) under WE-5289/1-2 project EXASTEEL-II.

Contents

Contents **vii**

1 Overview of state-of-the-art sparse direct solvers **1**
1.1 Maximum weight matching to enable static data structures 2
1.2 Symbolic reordering techniques based on nested dissection 8
1.3 Sparse LU factorization 12
1.4 Supernodal data structures 18
1.5 Sparse linear factorization solvers 24

2 Performance modeling of sparse factorization solvers **29**
2.1 Memory hierarchies, autotuning, and solvers 30
2.2 Computer architecture 31
2.3 Performance modeling using the LIKWID tools 35
2.4 Berkeley roofline model 37
2.5 Erlangen Execution-cache-memory model 42
2.6 ECM model application 45

3 Node-level performance modeling of sparse triangular solve **55**
3.1 Algorithm and data structures of sparse triangular solve 57
3.2 Performance analysis of sparse triangular solve 58
3.3 Application of modified Berkeley roofline model 62
3.4 Application of Erlangen ECM model on sparse triangular solve . . 64
3.5 Experimental testbed for the performance evaluation 65

4 Performance evaluation and analysis **71**
4.1 Single core performance modeling, evaluation, and analysis 71
4.2 Multiple core performance modeling, evaluation, and analysis . . 74

5 Conclusion **79**

Bibliography **81**

Chapter 1

Overview of state-of-the-art sparse direct solvers

This chapter presents an overview of combinatorial algorithms in sparse elimination methods. Beside well-established techniques that have been developed in the last twenty years, a modern viewpoint of sparse LU and LDL^T decomposition is presented that illustrates how the evolution of techniques in the last decade improved the performance of sparse direct solvers by three to four orders of magnitude. Some parts of this chapter have been published as an invited overview paper on parallel sparse direct methods in Bollhöfer et al. [2020].

Solving large sparse linear systems is at the heart of many application problems arising from computational science and engineering applications. Advances in combinatorial methods in combination with modern computer architectures have massively influenced the design of state-of-the-art direct solvers that are feasible for solving larger systems efficiently in a computational environment with rapidly increasing memory resources and cores. Among these advances are novel combinatorial algorithms for improving diagonal dominance which pave the way to a static pivoting approach, thus improving the efficiency of the factorization phase dramatically. Besides, partitioning and reordering the system such that a high level of concurrency is achieved, the objective is to simultaneously achieve the reduction of fill-in and the parallel concurrency. While these achievements already significantly improve the factorization phase, modern computer architectures require one to compute as many operations as possible in the cache of the CPU. This in turn can be achieved when dense subblocks that show up during the factorization can be grouped together into dense submatrices which are handled by multithreaded and cache-optimized dense matrix kernels using level-3 BLAS and LAPACK (Anderson et al. [1999]).

This chapter reviews some of the basic technologies together with the latest developments for sparse direct solution methods that have led to state-of-the-art LU decomposition methods. The chapter is organized as follows. In Section 1.1 maximum weighted matching is introduced which is one of the key tools in combinatorial optimization to dramatically improve the diagonal dominance of the underlying system. Next, Section 1.2 reviews multilevel nested dissection as a combinatorial method to reorder a system symmetrically such that fill-in and parallelization are improved simultaneously, once pivoting can be more or less ignored. After that, established graph-theoretical approaches are reviewed in Section 1.3, in particular, the elimination tree, from which most of the properties of the LU factorization can be concluded. Among these properties is the prediction of dense submatrices in the factorization. In this way several subsequent columns of the factors L and U^T are collected in a single dense block. This is the basis for the use of dense matrix kernels, using optimized level-3 BLAS as well, to exploit fast computation using the cache hierarchy which is discussed in Section 1.4. We assume that the reader is familiar with some elementary knowledge from graph theory; see, e.g., Davis [2006] and some simple computational algorithms based on graphs in Aho et al. [1983]. Finally Section 1.5 shows how the building blocks described in previous sections are used to implement the efficient sparse direct solver on modern hardware. Solving a sparse linear system is a computational kernel of many applications in science and engineering, thus an efficient solver is a key component of such applications.

1.1 Maximum weight matching to enable static data structures

In modern sparse elimination methods the key to success is the ability to work with efficient data structures and their underlying numerical templates. If we can increase the size of the diagonal entries as much as possible in advance, pivoting during Gaussian elimination can often be bypassed and we may work with static data structures and the numerical method will be significantly accelerated. A popular method to achieve this goal is the maximum weight matching method (Duff and Koster [1999]; Olschowka and Neumaier [1996]) which permutes, e.g., the rows of a given nonsingular matrix $A \in \mathbb{R}^{n,n}$ by a permutation matrix $\Pi \in \mathbb{R}^{n,n}$ such that $\Pi^T A$ has a nonzero diagonal. Moreover, it maximizes the product of the absolute diagonal values and yields diagonal scaling matrices $D_r, D_c \in \mathbb{R}^{n,n}$, such that $\tilde{A} = \Pi^T D_r A D_c$ satisfies $|\tilde{a}_{ij}| \leqslant 1$ and $|\tilde{a}_{ii}| = 1$ for all

$i, j = 1, \ldots, n$. The original idea on which these nonsymmetric permutations and scalings are based is to find a maximum weighted matching of a bipartite graph. Finding a maximum weighted matching is a well known assignment problem in operations research and combinatorial analysis.

Definition 1.1.1 *A graph $G = (V, E)$ with vertices V and edges $E \subset V^2$ is called bipartite if V can be partitioned into two sets V_r and V_c, such that no edge $e = (v_1, v_2) \in E$ has both ends v_1, v_2 in V_r or both ends v_1, v_2 in V_c. In this case we denote G by $G_b = (V_r, V_c, E)$.*

Definition 1.1.2 *Given a matrix A, then we can associate with it a canonical bipartite graph $G_b(A) = (V_r, V_c, E)$ by assigning the labels of $V_r = \{r_1, \ldots, r_n\}$ with the row indices of A and $V_c = \{c_1, \ldots, c_n\}$ being labeled by the column indices. In this case E is defined via $E = \{(r_i, c_j) |\ a_{ij} \neq 0\}$.*

For the bipartite graph $G_b(A)$ we see immediately that If $a_{ij} \neq 0$, then we have that $r_i \in V_r$ from the row set is connected by an edge $(r_i, c_j) \in E$ to the column $c_j \in V_c$, but neither row is connected to each other nor do the columns have interconnections.

Definition 1.1.3 *A matching $\mathcal{M}$ of a given graph $G = (V, E)$ is a subset of edges $e \in E$ such that no two of which share the same vertex.*

If $\mathcal{M}$ is a matching of a bipartite graph $G_b(A)$, then each edge $e = (r_i, c_j) \in \mathcal{M}$ corresponds to a row i and a column j and there exists no other edge $\hat{e} = (r_k, c_l) \in \mathcal{M}$ that has the same vertices, neither $r_k = r_i$ nor $c_l = c_j$.

Definition 1.1.4 *A matching $\mathcal{M}$ of $G = (V, E)$ is called maximal, if no other edge from E can be added to $\mathcal{M}$.*

If, for an $n \times n$ matrix A a matching $\mathcal{M}$ of $G_b(A)$ with maximum cardinality n is found, then by definition the edges must be $(i_1, 1), \ldots, (i_n, n)$ with $i_1, \ldots, i_n$ being the numbers $1, \ldots, n$ in a suitable order and therefore we obtain $a_{i_1,1} \neq 0, \ldots a_{i_n,n} \neq 0$. In this case we have established that the matrix A is at least structurally nonsingular and we can use a row permutation matrix Π^T associated with row ordering $i_1, \ldots, i_n$ to place a nonzero entry on each diagonal location of $\Pi^T A$.

Definition 1.1.5 *A perfect matching is a maximal matching with cardinality n.*

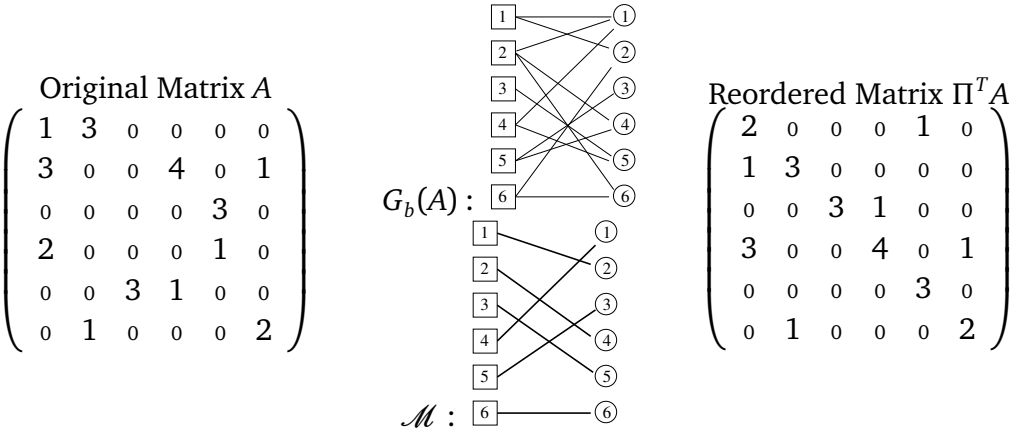


Figure 1.1. Perfect matching. Left side: original matrix A. Middle: bipartite representation $G_b(A) = (V_r, V_c, E)$ of the matrix A and perfect matching $\mathscr{M}$. Right side: permuted matrix $\Pi^T A$.

It can be shown that for a structurally nonsingular matrix A there always exists a perfect matching $\mathscr{M}$.

Perfect matching

In Figure 1.1, the set of edges $\mathscr{M} = \{(1,2),(2,4),(3,5),(4,1),(5,3),(6,6)\}$ represents a perfect maximum matching of the bipartite graph $G_b(A)$.

The most efficient combinatorial methods for finding maximum matchings in bipartite graphs make use of an augmenting path. We will introduce some graph terminology for the construction of perfect matchings.

Definition 1.1.6 *If an edge $e = (u, v)$ in a graph $G = (V, E)$ joins vertices $u, v \in V$, then we denote it as uv. A path then consists of edges $u_1u_2, u_2u_3, u_3u_4 \ldots, u_{k-1}u_k$, where each $(u_i, u_{i+1}) \in E$, $i = 1, \ldots, k-1$.*

If $G_b = (V_r, V_c, E)$ is a bipartite graph, then by the definition of a path, any path is alternating between the vertices of V_r and V_c, e.g., paths in G_b could be such as $r_1c_2, c_2r_3, r_3c_4, \ldots$.

Definition 1.1.7 *Given a graph $G = (V, E)$, a vertex is called free if it is not incident to any other edge in a matching $\mathscr{M}$ of G. An alternating path relative to a matching $\mathscr{M}$ is a path $P = u_1u_2, u_2u_3, \ldots, u_{s-1}u_s$ where its edges are alternating between $E \setminus \mathscr{M}$ and $\mathscr{M}$. An augmenting path relative to a matching $\mathscr{M}$ is an alternating path of odd length and both of it vertex endpoints are free.*

Augmenting path

Consider Figure 1.1. To better distinguish between row and column vertices we use $\boxed{1},\boxed{2},\ldots,\boxed{6}$ for the rows and $\textcircled{1},\textcircled{2},\ldots,\textcircled{6}$ for the columns. A non-perfect but maximal matching is given by $M = \{(\boxed{4},\textcircled{5}),(\boxed{1},\textcircled{1}),(\boxed{6},\textcircled{2}),(\boxed{2},\textcircled{6}),(\boxed{5},\textcircled{4})\}$. We can easily see that an augmenting path alternating between rows and columns is given by $\boxed{3}\textcircled{5}$, $\textcircled{5}\boxed{4}$, $\boxed{4}\textcircled{1}$, $\textcircled{1}\boxed{1}$, $\boxed{1}\textcircled{2}$, $\textcircled{2}\boxed{6}$, $\boxed{6}\textcircled{6}$, $\textcircled{6}\boxed{2}$, $\boxed{2}\textcircled{4}$, $\textcircled{4}\boxed{5}$, $\boxed{5}\textcircled{3}$. Both endpoints $\boxed{3}$ and $\textcircled{3}$ of this augmenting path are free.

In a bipartite graph $G_b = (V_r, V_c, E)$ one vertex endpoint of any augmenting path must be in V_r whereas the other one must be in V_c. The symmetric difference, $A \oplus B$ of two edge sets A, B is defined to be $(A \setminus B) \cup (B \setminus A)$.

Using these definitions and notations, the following theorem (Berge [1957]) gives a constructive algorithm for finding perfect matchings in bipartite graphs.

Theorem 1.1.1 *If $\mathscr{M}$ is a nonmaximum matching of a bipartite graph $G_b = (V_r, V_c, E)$, then there exists an augmenting path P relative to $\mathscr{M}$ such that $P = \tilde{\mathscr{M}} \oplus \mathscr{M}$ and $\tilde{\mathscr{M}}$ is a matching with cardinality $|\mathscr{M}| + 1$.*

According to this theorem, a combinatorial method of finding a perfect matching in a bipartite graph is to seek augmenting paths.

The perfect matching as discussed so far only takes the nonzero structure of the matrix into account. For their use as static pivoting methods prior to the LU decomposition one requires, in addition, to maximize the absolute value of the product of the diagonal entries. This is referred to as maximum weighted matching. In this case a permutation π has to be found, which maximizes

$$\prod_{i=1}^{n} |a_{\pi(i)i}|. \tag{1.1}$$

The maximization of this product is transferred into a minimization of a sum as follows. We define a matrix $C = (c_{ij})$ via

$$c_{ij} = \begin{cases} \log a_i - \log |a_{ij}|, & a_{ij} \neq 0, \\ \infty, & \text{otherwise}, \end{cases}$$

where $a_i = \max_j |a_{ij}|$ is the maximum element in row i of matrix A. A permutation π which minimizes the sum

$$\sum_{i=1}^{n} c_{\pi(i)i}$$

also maximizes the product (1.1). The minimization problem is known as the linear-sum assignment problem or bipartite weighted matching problem in combinatorial optimization. The problem is solved by a sparse variant of the Hungarian method. The complexity is $\mathcal{O}(n\tau \log n)$ for sparse matrices with τ entries. For matrices, whose associated graphs fulfill special requirements, this bound can be reduced further to $\mathcal{O}(n^{\alpha}(\tau + n \log n))$ with $\alpha < 1$. All graphs arising from finite-difference or finite element discretizations meet the conditions (Gupta and Ying [1999]). As before, we finally get a perfect matching which in turn defines a nonsymmetric permutation.

When solving the assignment problem, two dual vectors $u = (u_i)$ and $v = (v_i)$ are computed which satisfy

$$u_i + v_j = c_{ij}, \qquad (i,j) \in \mathcal{M}, \tag{1.2}$$
$$u_i + v_j \leq c_{ij}, \qquad \text{otherwise.} \tag{1.3}$$

Using the exponential function these vectors can be used to scale the initial matrix. To do so define two diagonal matrices D_r and D_c through

$$D_r = \operatorname{diag}(d_1^r, d_2^r, \ldots, d_n^r), \qquad d_i^r = \exp(u_i), \tag{1.4}$$
$$D_c = \operatorname{diag}(d_1^c, d_2^c, \ldots, d_n^c), \qquad d_j^c = \exp(v_j)/a_j. \tag{1.5}$$

Using (1.2) and (1.3) and the definition of C, it immediately follows that $\tilde{A} = \Pi^T D_r A D_c$ satisfies

$$|\tilde{a}_{ii}| = 1, \tag{1.6}$$
$$|\tilde{a}_{ij}| \leq 1. \tag{1.7}$$

The permuted and scaled system $\tilde{A}$ has been observed to have significantly better numerical properties when being used for direct methods or for preconditioned iterative methods; cf., e.g., Benzi et al. [2000]; Duff and Koster [1999]. Olschowka and Neumaier [1996] introduced these scalings and permutation for reducing pivoting in Gaussian elimination of full matrices. The first implementation for sparse matrix problems was introduced by Duff and Koster [1999]. For symmetric matrices $|A|$, these nonsymmetric matchings can be converted to

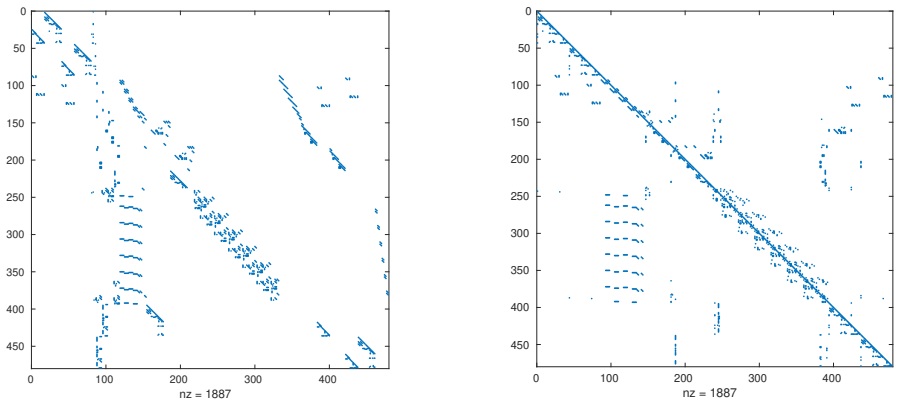


Figure 1.2. Maximum weight matching. Left side: original matrix A. Right side: permuted and rescaled matrix $\tilde{A} = \Pi^T D_r A D_c$.

a symmetric permutation P and a symmetric scaling $D_s = (D_r D_c)^{1/2}$ such that $P^T D_s A D_s P$ consists mostly of diagonal blocks of size 1×1 and 2×2 satisfying a similar condition as (1.6) and (1.7), where in practice it rarely happens that 1×1 blocks are identical to 0 (Duff and Pralet [2004]). Recently, successful parallel approaches to compute maximum weighted matchings have been proposed (Langguth et al. [2011, 2014]).

Maximum weight matching

To conclude this section we demonstrate the effectiveness of maximum weight matchings using a simple sample matrix "west0479" from the SuiteSparse Matrix Collection. The matrix can also directly be loaded into MATLAB using `load west0479`. In Figure 1.2 we display the matrix before and after applying maximum weighted matchings. To illustrate the improved diagonal dominance we further compute $r_i = |a_{ii}| / \sum_{j=1}^{n} |a_{ij}|$ for each row of A and $\tilde{A} = \Pi^T D_r A D_s$, $i = 1, \ldots, n$. r_i can be read as the relative diagonal dominance of row i and yields a number between 0 and 1. Moreover, whenever $r_i > \frac{1}{2}$, the row is strictly diagonal dominant, i.e., $|a_{ii}| > \sum_{j:j \neq i} |a_{ij}|$. In Figure 1.3 we display for both matrices r_i by sorting its values in increasing order and taking $\frac{1}{2}$ as the reference line. We can see the dramatic impact of maximum weighted matchings in improving the diagonal dominance of the given matrix and thus paving the way to a static pivoting approach in incomplete or complete LU decomposition methods.

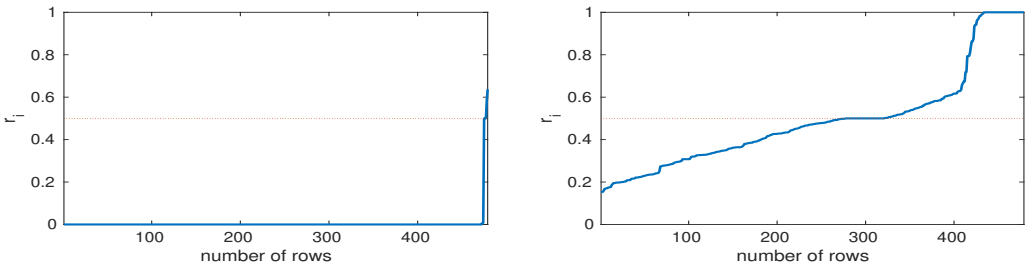


Figure 1.3. Diagonal dominance. Left side: r_i for A. Right side: r_i for $\tilde{A} = \Pi^T D_r A D_c$.

1.2 Symbolic reordering techniques based on multilevel nested dissection

When dealing with large sparse matrices a crucial factor that determines the computation time is the amount of fill that is produced during the factorization of the underlying matrix. To reduce the complexity there exist many mainly symmetric reordering techniques that attempt to reduce the fill-in heuristically. Here we will demonstrate only one of these methods, the so-called nested dissection method. The main reason for selecting this method is that it can be easily used for parallel computations.

Recursive multilevel nested dissection methods for direct decomposition methods were first introduced in the context of multiprocessing. If parallel direct methods are used to solve a sparse system of equations, then a graph partitioning algorithm can be used to compute a fill reducing ordering that leads to a high degree of concurrency in the factorization phase.

Definition 1.2.1 *For a matrix $A \in \mathbb{R}^{n,n}$ we define the associated (directed) graph $G_d(A) = (V, E)$, where $V = \{1, \ldots, n\}$ and the set of edges $E = \{(i,j) | \, a_{ij} \neq 0\}$. The (undirected) graph is given by $G_d(|A| + |A|^T)$ and is denoted simply by $G(A)$.*

In graph terminology for a sparse matrix A we simply have a directed edge (i, j) for any nonzero entry a_{ij} in $G_d(A)$ whereas the orientation of the edge is ignored in $G(A)$.

The research on graph-partitioning methods in the mid nineties has resulted in high-quality software packages, e.g., METIS (Karypis and Kumar [1998]). These methods often compute orderings that, on the one hand, lead to small fill-in for (incomplete) factorization methods while on the other hand they provide a high level of concurrency. We will briefly review the main idea of multilevel nested dissection in terms of graph partitioning.

Definition 1.2.2 *Let $A \in \mathbb{R}^{n,n}$ and consider its graph $G(A) = (V, E)$. A k-way graph partitioning consists of partitioning V into k disjoint subsets $V_1, V_2, \ldots, V_k$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $\cup_i V_i = V$. The subset $E_s = E \cap \bigcup_{i \neq j}(V_i \times V_j)$ is called the edge separator.*

Typically we want a k-way partitioning to be balanced, i.e., each V_i should satisfy $|V_i| \approx n/k$. The edge separator E_s refers to the edges that have to be taken away from the graph in order to have k separate subgraphs associated with $V_1, \ldots, V_k$ and the number of elements of E_s is usually referred to as the edge-cut.

Definition 1.2.3 *Given $A \in \mathbb{R}^{n,n}$, a vertex separator V_s of $G(A) = (V, E)$ is a set of vertices such that there exists a k-way partitioning $V_1, V_2, \ldots, V_k$ of $V \setminus V_s$ having no edge $e \in V_i \times V_j$ for $i \neq j$.*

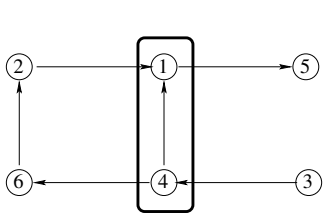
A useful vertex separator V_s should not only separate $G(A)$ into k independent subgraphs associated with $V_1, \ldots, V_k$, it is intended that the numbers of edges $\cup_{i=1}^{k} |\{e_{is} \in V_i, s \in V_s\}|$ is also small.

Nested dissection recursively splits a graph $G(A) = (V, E)$ into almost equal parts by constructing a vertex separator V_s until the desired number k of partitionings are obtained. If k is a power of 2, then a natural way of obtaining a vertex separator is to first obtain a 2-way partitioning of the graph, a so-called graph bisection with its associated edge separator E_s. After that a vertex separator V_s is computed from E_s, which gives a 2-way partitioning V_1, V_2 of $V \setminus V_s$. This process is then repeated separately for the subgraphs associated with V_1, V_2 until eventually a $k = 2^l$-way partitioning is obtained. For the reordering of the underlying matrix A, the vertices associated with V_1 are taken first followed by V_2 and V_s. This reordering is repeated similarly during repeated bisection of each V_i. In general, vertex separators of small size result in low fill-in.

Vertex separators

To illustrate vertex separators, we consider the reordered matrix $\Pi^T A$ from Figure 1.1 after a matching is applied. In Figure 1.4 we display its graph $G(\Pi^T A)$ ignoring the orientation of the edges. A 2-way partitioning is obtained with $V_1 = \{3, 5\}$, $V_2 = \{2, 6\}$, and a vertex separator $V_s = \{1, 4\}$. The associated reordering refers to taking the rows and the columns of $\Pi^T A$ in the order $3, 5, 2, 6, 1, 4$.

Since a naive approach to compute a recursive graph bisection is typically computationally expensive, combinatorial multilevel graph bisection has been



$$\left(\begin{array}{cc|cc|cc} 3 & 0 & 0 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 3 & 0 & 1 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 & 3 & 4 \end{array}\right)$$

Figure 1.4. A 2-way partition with vertex separator $V_s = \{1, 4\}$ and the associated reordered matrix placing the two rows and columns associated with V_s to the end.

used to accelerate the process. The basic structure is simple. The multilevel approach consists of three phases: at first there is a coarsening phase which compresses the given graph successively level by level by about half of its size. When the coarsest graph with about a few hundred vertices is reached, the second phase, namely, the so-called bisection is applied. This is a high quality partitioning algorithm. After that, during the uncoarsening phase, the given bisection is successively refined as it is prolongated towards the original graph.

Coarsening phase

The initial graph $G_0 = (V_0, E_0) = G(A)$ of $A \in \mathbb{R}^{n,n}$ is transformed during the coarsening phase into a sequence of graphs $G_1, G_2, \ldots, G_m$ of decreasing size such that $|V_0| \gg |V_1| \gg |V_2| \gg \cdots \gg |V_m|$. Given the graph $G_i = (V_i, E_i)$, the next coarser graph G_{i+1} is obtained from G_i by collapsing adjacent vertices. This can be done, e.g., by using a maximal matching $\mathcal{M}_i$ of G_i (cf. Definitions 1.1.3 and 1.1.4). Using $\mathcal{M}_i$, the next coarser graph G_{i+1} is constructed from G_i collapsing the vertices being matched into multinodes, i.e., the elements of $\mathcal{M}_i$ together with the unmatched vertices of G_i become the new vertices V_{i+1} of G_{i+1}. The new edges E_{i+1} are the remaining edges from E_i connected with the collapsed vertices. There are various differences in the construction of maximal matchings (Karypis and Kumar [1998]; Chevalier and Pellegrini [2008]). One of the most popular and efficient methods is heavy edge matching (Karypis and Kumar [1998]).

Partitioning phase

At the coarsest level m, a 2-way partitioning $V_{m,1} \dot{\cup} V_{m,2} = V_m$ of $G_m = (V_m, E_m)$ is computed, each of them containing about half of the vertices of G_m. This specific partitioning of G_m can be obtained by using various algorithms such as spec-

tral bisection (Fiedler [1975]) or combinatorial methods based on Kernighan–Lin variants (Kernighan and Lin [1970]; Fiduccia and Mattheyses [1997]). It is demonstrated in Karypis and Kumar [1998] that for the coarsest graph, combinatorial methods typically compute smaller edge-cut separators compared with spectral bisection methods. However, since the size of the coarsest graph G_m is small (typically $|V_m| < 100$), this step is negligible with respect to the total amount of computation time.

Uncoarsening phase

Suppose that at the coarsest level m, an edge separator $E_{m,s}$ of G_m associated with the 2-way partitioning has been computed that has led to a sufficient edge-cut of G_m with $V_{m,1}$, $V_{m,2}$ of almost equal size. Then $E_{m,s}$ is prolongated to G_{m-1} by reversing the process of collapsing matched vertices. This leads to an initial edge separator $E_{m-1,s}$ for G_{m-1}. But since G_{m-1} is finer, $E_{m-1,s}$ is suboptimal and one usually decreases the edge-cut of the partitioning by local refinement heuristics such as the Kernighan–Lin partitioning algorithm (Kernighan and Lin [1970]) or the Fiduccia–Mattheyses method (Fiduccia and Mattheyses [1997]). Repeating this refinement procedure level-by-level we obtain a sequence of edge separators $E_{m,s}, E_{m-1,s}, \ldots, E_{0,s}$ and, eventually, an edge separator $E_s = E_{0,s}$ of the initial graph $G(A)$ is obtained. If one is seeking for a vertex separator V_s of $G(A)$, then one usually computes V_s from E_s at the end.

There have been a number of methods that are used for graph partitioning, e.g. METIS (Karypis and Kumar [1998]), a parallel MPI version PARMETIS (Karypis et al. [1999]), or a recent multithreaded approach MT-METIS (LaSalle and Karypis [2013]). Another example for a parallel partitioning algorithm is SCOTCH (Chevalier and Pellegrini [2008]).

Multilevel nested dissection

We will continue Example 1.1 using the matrix $\tilde{A} = \Pi^T D_r A D_s$ that has been rescaled and permuted using maximum weight matching. We illustrate in Figure 1.5 how multilevel nested dissection changes the pattern $\hat{A} = P^T \tilde{A} P$, where P refers to the permutation matrix associated with the partitioning of $G(\tilde{A})$.

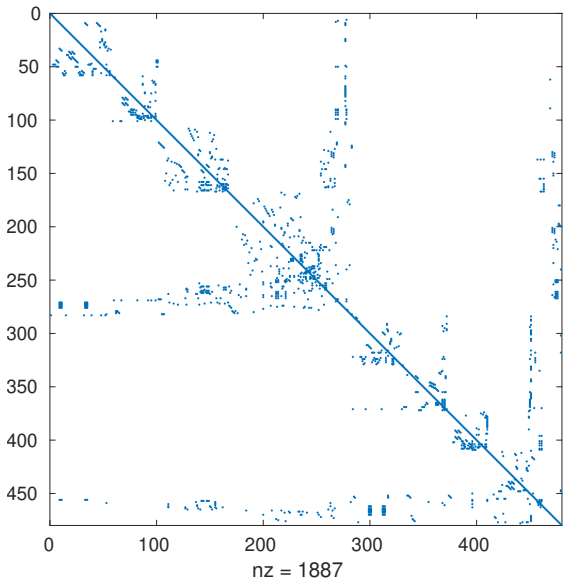


Figure 1.5. Application of multilevel nested dissection after the matrix is already rescaled and permuted using maximum weight matching.

1.3 Sparse LU factorization

In this section we will assume that the given matrix $A \in \mathbb{R}^{n,n}$ is nonsingular and that it can be factorized as $A = LU$, where L is a lower triangular matrix with unit diagonal and U is an upper triangular matrix. It is well known (George and Liu [1981]), if $A = LU$, where L and $U^\top$ are lower triangular matrices, then in the generic case we will have $G_d(L+U) \supset G_d(A)$, i.e., we will only get additional edges unless some entries cancel by "accident" during the elimination. In the following we will ignore cancellations. Throughout this section we will always assume that the diagonal entries of A are nonzero as well. We also assume that $G_d(A)$ is connected.

In the preceding sections we have argued that maximum weight matching often leads to a rescaled and reordered matrix such that static pivoting is likely to be enough, i.e., pivoting is restricted to some dense blocks inside the LU factorization. Furthermore, reordering strategies such as multilevel nested dissection have further symmetrically permuted the system such that the fill-in that occurs during Gaussian elimination is acceptable and even parallel approaches could be drawn from this reordering, thus assuming that A does not need further reordering and a factorization $A = LU$ exists is a realistic scenario in what follows.

$$\left(\begin{array}{cc|cc|cc} 3 & 0 & 0 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 3 & 0 & 1 & 0 \\ 0 & 0 & 1 & 2 & \times & 0 \\ \hline 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 & 3 & 4 \end{array}\right)$$

Figure 1.6. Fill-in with respect to $L+U$ is denoted by $\times$.

1.3.1 The elimination tree

The basis of determining the fill-in in the triangular factors L and U as a by-product of the Gaussian elimination can be characterized as follows (see Gilbert [1994] and the references therein).

Theorem 1.3.1 *Given $A = LU$ with the aforementioned assumptions, there exists an edge (i, j) in $G_d(L+U)$ if and only if there exists a path*

$$ix_1, x_2x_3, \ldots, x_kj$$

in $G_d(A)$ such that $x_1, \ldots, x_k < \min(i, j)$.

In other words, during Gaussian elimination we obtain a fill edge (i, j) for every path from i to j through vertices less than $\min(i, j)$.

Fill-in

We will use the matrix $\Pi^T A$ from Example 1.2 and sketch the fill-in obtained during Gaussian elimination in Figure 1.6.

The fastest known method for predicting the filled graph $G_d(L+U)$ is Gaussian elimination. The situation is simplified if the graph is undirected. In the following we ignore the orientation of the edges and simply consider the undirected graph $G(A)$ and $G(L+U)$, respectively.

Definition 1.3.1 *The undirected graph $G(L+U)$ that is derived from the undirected graph $G(A)$ by applying Theorem 1.3.1 is called the filled graph and it will be denoted by $G_f(A)$.*

Remark 1.3.1 *It follows immediately from the construction of elimination tree $T(A)$ and Theorem 1.3.1 that additional edges of $G_f(A)$ which are not covered by*

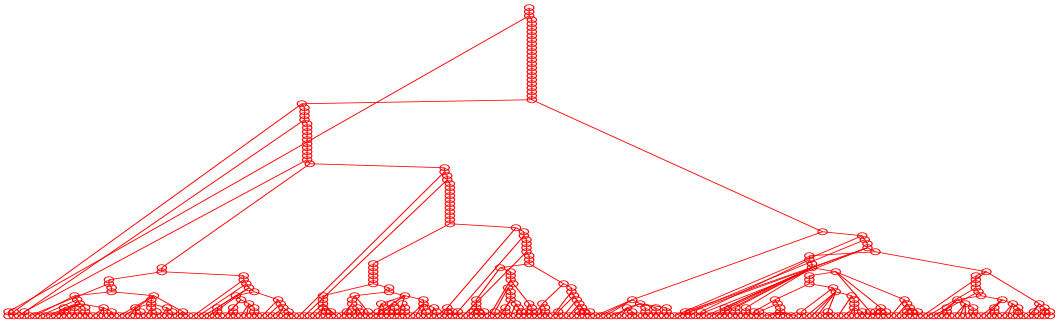


Figure 1.7. Elimination tree of "west0479" after maximum weight matching and nested dissection are applied.

the elimination tree can only show up between a vertex and some of its ancestors (referred to as "back-edges"). In contrast to that, "cross-edges" between unrelated vertices do not exist.

Remark 1.3.2 *One immediate consequence of Remark 1.3.1 is that triangular factors can be computed independently starting from the leaves until the vertices meet a common parent, i.e., column j of L and U^T only depend on those columns s of L and U^T such that s is a descendant of j in the elimination tree $T(A)$.*

Elimination tree

We use the matrix west0479 from Example 1.2, after maximum weight matching and multilevel nested dissection have been applied. We use the MATLAB `etreeplot` to display its elimination tree (see Figure 1.7). The elimination tree displays the high level of concurrency that is induced by nested dissection, since by Remark 1.3.2 the computations can be executed independently at each leaf node towards the root until a common parent vertex is reached.

Further conclusions can be easily derived from the elimination tree, in particular, Remark 1.3.2 in conjunction with Theorem 1.3.1.

Remark 1.3.3 *Consider some $k \in \{1, \ldots, n\}$. Then there exists a (fill) edge (j,k) with $j < k$ if and only if there exists a common descendant i of k, j in $T(A)$ such that $a_{ik} \neq 0$. This follows from the fact that once $a_{ik} \neq 0$, by Theorem 1.3.1 this induces (fill) edges (j,k) in the filled graph $G_f(A)$ for all nodes j between i and k in the elimination tree $T(A)$, i.e., for all ancestors of i that are also descendants of k. This way, i propagates fill-edges along the branch from i to k in $T(A)$ and the*

information $a_{ik} \neq 0$ can be used as path compression to advance from i towards k along the elimination tree.

1.3.2 The supernodal factorization approach

We have already seen that the elimination tree reveals information about concurrency. It is further useful to determine the fill-in L and U^T. This information can be computed from the elimination tree $T(A)$ together with $G(A)$. The basis for determining the fill-in in each column is again Remark 1.3.3. Suppose we are interested in the nonzero entries of column j of L and U^T. Then for all descendants of j, i.e., the nodes of the subtree $T(j)$ rooted at vertex j, a nonzero entry $a_{ik} \neq 0$ also implies $l_{kj} \neq 0$. Thus, starting at any leaf i, we obtain its fill by all $a_{ik} \neq 0$ such that $k > i$ and when we move forward from i to its parent j, vertex j will inherit the fill from node i for all $k > j$ plus the nonzero entries given by $a_{jk} \neq 0$ such that $k > j$. When we reach a common parent node k with multiple children, the same argument applies using the union of fill-in greater than k from its children together with the nonzero entries $a_{kl} \neq 0$ such that $l > k$. We summarize this result in a very simple algorithm

Computation of fill-in

Require: $A \in \mathbb{R}^{n,n}$ such that A has the same pattern as $|A| + |A|^T$.
Ensure: sparse strict lower triangular pattern $P \in \mathbb{R}^{n,n}$ with same pattern as L, U^T.

```
compute parent array p of the elimination tree T(A)
for j = 1, ..., n do
    supplement nonzeros of column j of P with all i > j such that a_ij ≠ 0
    k = p_j
    if k > 0 then
        supplement nonzeros of column k of P with nonzeros of column j of P greater than k
    end if
end for
```

Algorithm 1.3.2 only deals with the fill pattern. One additional aspect that allows one to raise efficiency and to speed up the numerical factorization significantly is to detect dense submatrices in the factorization. Block structures allow

one to collect parts of the matrix in dense blocks and to treat them commonly using dense matrix kernels such as level-3 BLAS and LAPACK (Anderson et al. [1999]).

Dense blocks can be read off from the elimination tree employing Algorithm 1.3.2.

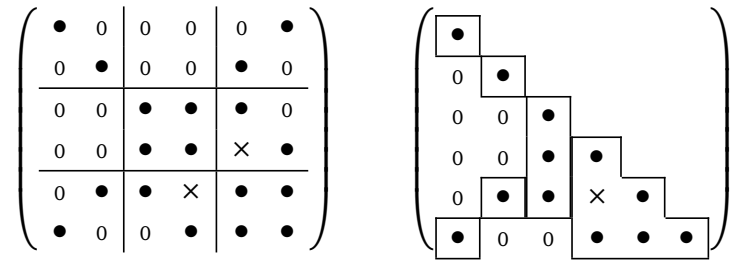
Definition 1.3.2 *Denote by $\mathscr{P}_j$ the nonzero indices of column j of P as computed by Algorithm 1.3.2. A sequence $k, k+1, \ldots, k+s-1$ is called a supernode of size s if the columns of $\mathscr{P}_j = \mathscr{P}_{j+1} \cup \{j+1\}$ for all $j = k, \ldots, k+s-2$.*

In simple words, Definition 1.3.2 states that for a supernode s subsequent columns can be grouped together in one dense block with a triangular diagonal block and a dense subdiagonal block since they perfectly match the associated trapezoidal shape. We can thus easily supplement Algorithm 1.3.2 with a supernode detection.

Computation of fill-in and supernodes

Require: $A \in \mathbb{R}^{n,n}$ such that A has the same pattern as $|A| + |A|^T$.
Ensure: sparse strict lower triangular pattern $P \in \mathbb{R}^{n,n}$ with the same pattern as L, U^T as well as column size $s \in \mathbb{R}^m$ of each supernode.

compute parent array p of the elimination tree $T(A)$
$m \leftarrow 0$
for $j = 1, \ldots, n$ **do**
  supplement nonzeros of column j of P with all $i > j$ such that $a_{ij} \neq 0$
  denote by r the number of entries in column j of P
  if $j > 1$ and $j = p_{j-1}$ and $s_m + r = l$ **then**
    $s_m \leftarrow s_m + 1$ ▷ continue current supernode
  else
    $m \leftarrow m+1$, $s_m \leftarrow 1$, $l \leftarrow r$ ▷ start new supernode
  end if
  $k = p_j$
  if $k > 0$ **then**
    supplement nonzeros of column k of P with nonzeros of column j of P greater than k
  end if
end for



(a) Undirected graph $G(A)$ represented as a matrix.

(b) Supernodes in the triangular factor.

Figure 1.8. Matrix representation of an undirected graph $G(A)$ and supernodes in its triangular factor. Entries of $G(A)$ are denoted by •, fill-in is denoted by ×.

Supernode computation

To illustrate the use of supernodes, we consider the matrix pattern from Figure 1.8a and illustrate the underlying dense block structure in Figure 1.8b. Supernodes are the columns 1, 2, 3 as scalar columns as well as columns 4–6 as one single supernode.

Supernodes form the basis of several improvements, e.g., a supernode can be stored as one or two dense matrices. Beside the storage scheme as dense matrices, the nonzero row indices for these blocks need only be stored once. Next the use of dense submatrices allows the usage of dense matrix kernels using level-3 BLAS (Anderson et al. [1999]).

Supernodes

We use the matrix west0479 from Example 1.2, after maximum weight matching and multilevel nested dissection have been applied. We use its undirected graph to compute the supernodal structure. Certainly, since the matrix is nonsymmetric, the block structure is only suboptimal. We display the supernodal structure for the associated Cholesky factor, i.e., for the Cholesky factor of a symmetric positive definite matrix with same undirected graph as our matrix (see top part of Figure 1.9). Furthermore, we display the supernodal structure for the factors L and U computed from the nonsymmetric matrix without pivoting (see bottom part of Figure 1.9).

While the construction of supernodes is fairly easy in the symmetric case, its generalization for the nonsymmetric case is significantly harder, since one has to deal with pivoting in each step of Gaussian elimination. In this case one uses the column elimination tree (George and Ng [1985]).

1.4 Supernodal data structures

High-performance sparse solver libraries have been a very important part of scientific and engineering computing for years, and their importance continues to grow as microprocessor architectures become more complex and software libraries become better designed to integrate easily within applications. Despite the fact that there are various science and engineering applications, the underlying algorithms typically have remarkable similarities, especially those algorithms that are most challenging to implement well in parallel. It is not too strong a statement to say that these software libraries are essential to the broad success of scalable high-performance computing in computational sciences. In this section we demonstrate the benefit of supernodal data structures within the sparse solver package PARDISO (Schenk and Gärtner [2004]; Bollhöfer et al. [2020]). We illustrate it by using the triangular solution process. The forward and backward substitution is performed columnwise with respect to the columns of L, starting with the first column, as depicted in Figure 1.10. The data dependencies here allow one to store vectors y, z, b, and x in only one vector r. When column j is reached, r_j contains the solution for y_j. All other elements of L in this column, i.e., L_{ij} with $i = j+1,\ldots,N$, are used to update the remaining entries in r by

$$r_i = r_i - r_j L_{ij}. \tag{1.8}$$

The backward substitution with L^T will take place rowwise, since we use L and perform the substitution columnwise with respect to L, as shown in the lower part of Figure 1.10. In contrast to the forward substitution the iteration over columns starts at the last column N and proceeds to the first one. If column j is reached, then r_j, which contains the j-component of the solution vector x_j, is computed by subtracting the dot product of the remaining elements in the column L_{ij} and the corresponding elements of r_i with $i = j+1,\ldots,N$ from it:

$$r_j = r_j - r_i L_{ij}. \tag{1.9}$$

After all columns have been processed r contains the required solution x. It is important to note that line 5 represents in both substitutions an indexed DAXPY and

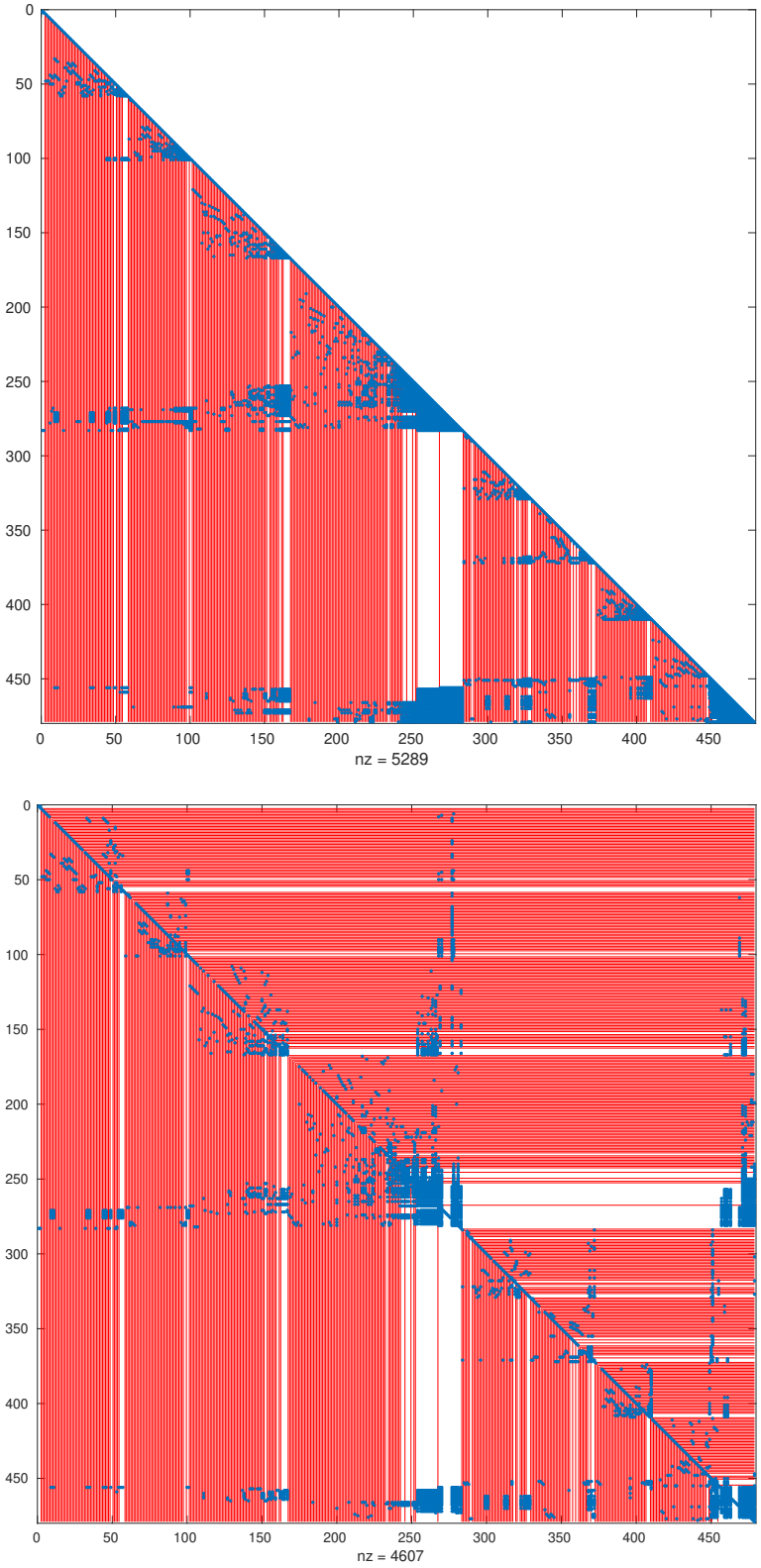
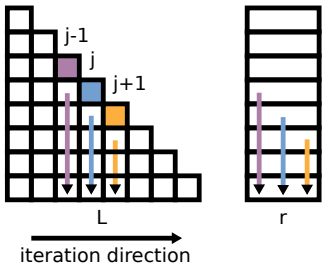
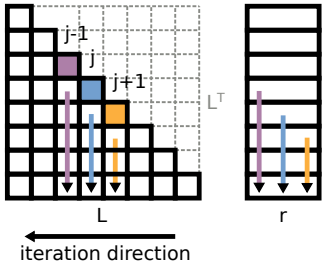


Figure 1.9. Supernodal structure. Top: vertical lines display the blocking of the supernodes with respect to the associated Cholesky factor. Bottom: vertical and horizontal lines display the blocking of the supernodes applied to L and U.




```
1: procedure SPARSE FORWARD SUBSTITUTION
2:     for j = 0; j < n; j++ do
3:         for i = xl[j]; i < xl[j+1]; i++ do
4:             row = id[i]
5:             r[row] -= r[j] * l[i]        ▷ indexed DAXPY
6:         end for
7:     end for
8: end procedure
```




```
1: procedure SPARSE BACKWARD SUBSTITUTION
2:     for j = n; j > 0; j - - do
3:         for i = xl[j]; i < xl[j+1]; i++ do
4:             row = id[i]
5:             r[j] -= r[row] * l[i]        ▷ indexed DDOT
6:         end for
7:     end for
8: end procedure
```

Figure 1.10. Sparse triangular substitution in CSC format based on indexed DAXPY/DDOT kernel operations.

indexed DDOT kernel operations that have to be computed during the streaming operations of the vector r and the column j of the numerical factor L. As we are dealing with sparse matrices it makes no sense to store the lower triangular matrix L as a dense matrix. Hence PARDISO uses its own data structure to store L, as shown in Figure 1.11.

Adjacent columns exhibiting the same row sparsity structure form a *panel*, also known as a *supernode*. A panel's column count is called the *panel size* n_p. The columns of a panel are stored consecutively in memory excluding the zero entries. Note that columns of panels are padded in the front with zeros so they get the same length as the first column inside their panel. The padding is of utmost performance for the PARDISO solver to use Level-3 BLAS and LAPACK functionalities (Schenk [2000]). Furthermore panels are stored consecutively in the `l` array. Row and column information is now stored in accompanying arrays. The `xsuper` array stores for each panel the index of its first column. Also note that here column indices are the running count of nonzero columns. Column indices are used as indices into the `xl` array to look up the start of the column in the `l` array which contains the numerical values of the factor L. To determine the row index of a column's element an additional array `id` is used, which holds for each

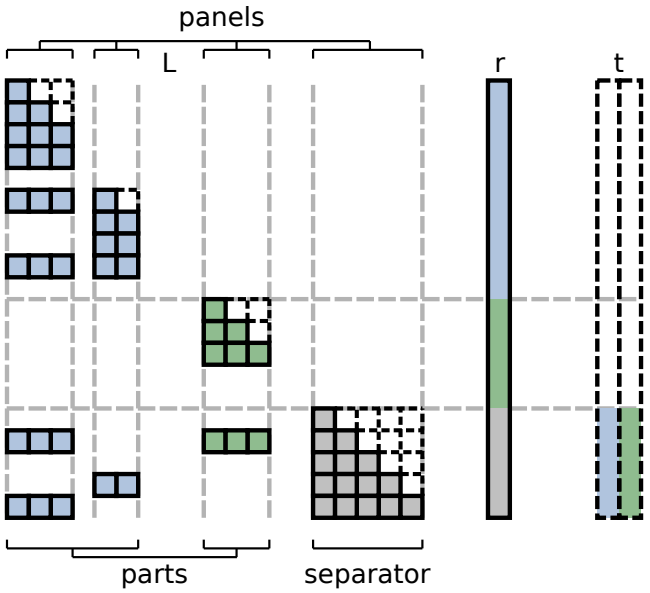


Figure 1.11. Sparse matrix data structures in PARDISO. Adjacent columns of L exhibiting the same structure form panels also known as supernodes. Groups of panels which touch independent elements of the right hand side r are parts. The last part in the lower triangular matrix L is called the separator.

panel the row indices. The start of a panel inside `id` is found via `xid` array. The first row index of panel p is `id[xid[p]]`. For serial execution this information is enough. However, during parallel forward/backward substitution concurrent updates to the same entry of `r` must be avoided. The *parts* structure contains the start (and end) indices of the panels which can be updated independently as they do not touch the same entries of r. Two parts, colored blue and green, are shown in Figure 1.11. The last part in the bottom right corner of L is special and is called the *separator* and is colored gray. Parts which would touch entries of `r` in the range of the separator perform their updates into separate temporary arrays `t`. Before the separator is then serially updated, the results of the temporary arrays are gathered back into `r`. The backward substitution works the same, just reversed and only updates to different temporary arrays are not required. The complete forward substitution and backward substitution is listed in Algorithms 1 and 2.

Algorithm 1 Forward substitution in PARDISO. Note that in case of serial execution separated updates to temporary arrays in line 10–13 are not necessary and can be handled via the loop in lines 6–9.

```
 1: procedure FORWARD
 2:   for part o in parts do                          ▷ parallel execution
 3:     for panel p in part p do
 4:       for column j in panel do                                ▷ unroll
 5:         i = xid[p] + offset
 6:         for k = xl[j] + offset; k < sep; ++k do
 7:           row = id[i++]
 8:           r[row] - = r[j] l[k]                    ▷ indexed DAXPY
 9:         end for
10:         for k = sep + 1; k < xl[j+1]; ++k do
11:           row = id[i++]
12:           t[row,p] -= r[j] l[k]                   ▷ indexed DAXPY
13:         end for
14:       end for
15:     end for
16:   end for
17:   r[i] = r[i] - sum(t[i,:])                  ▷ gather temporary arrays
18:   for panel p in separator do                        ▷ serial execution
19:     for column j in panel do                                ▷ unroll
20:       i = xid[p] + offset
21:       for k = xl[j] + offset; k < xl[j+1]; ++k do
22:         row = id[i++]
23:         r[row] -= r[j] l[k]                       ▷ indexed DAXPY
24:       end for
25:     end for
26:   end for
27: end procedure
```

Algorithm 2 Backward substitution in PARDISO. Separator (sep.), parts, and panels are iterated over in reversed (rev.) order.

```
 1: procedure BACKWARD
 2:   for panel p in sep. rev. do                          ▷ serial execution
 3:     for col. j in panel p rev. do                      ▷ unroll
 4:       i = xid[p] + offset
 5:       for k = xl[j] + offset; k < xl[j+1]; ++k do
 6:         row = id[i++]
 7:         r[j] -= r[row] l[k]                           ▷ indexed DDOT
 8:       end for
 9:       offset = offset - 1
10:     end for
11:   end for
12:   for part in parts do                                ▷ parallel execution
13:     for panel p in part rev. do
14:       for col. j in panel p rev. do                   ▷ unroll
15:         i = xid[p] + offset
16:         for k = xl[j] + offset; k < xl[j+1]; ++k do
17:           row = id[i++]
18:           r[j] -= r[row] l[k]                       ▷ indexed DDOT
19:         end for
20:         offset = offset - 1
21:       end for
22:     end for
23:   end for
24: end procedure
```

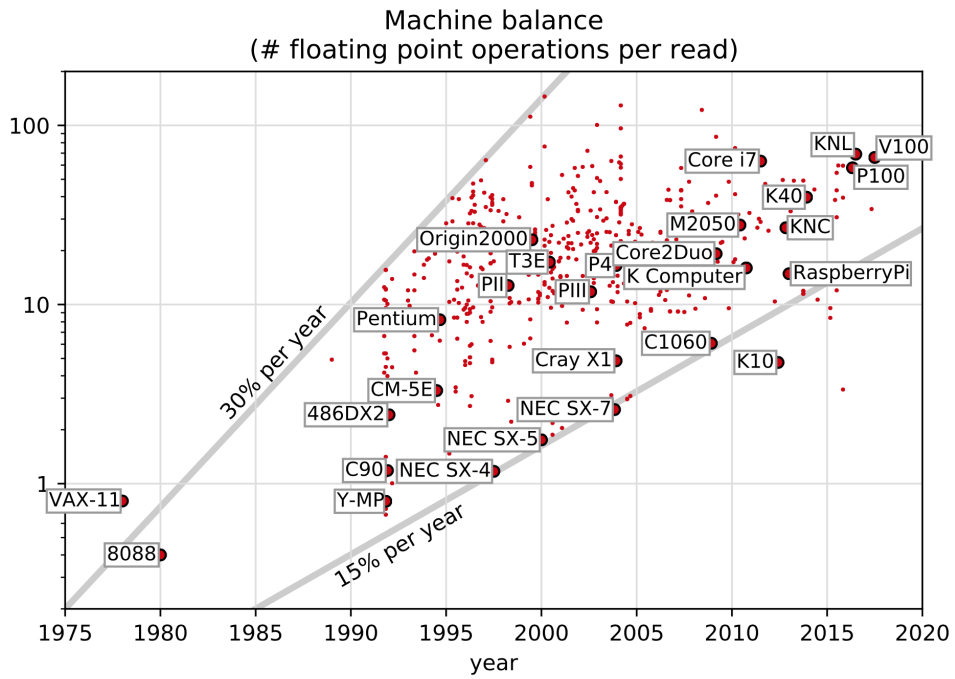


Figure 1.12. Evolution of machine balance of processors. From Abdelfattah et al. [2021].

1.5 High-performance computing mathematical libraries — sparse linear factorization solvers

High-performance computing mathematical libraries have been a very important part of scientific and engineering computing for years, and their importance continues to grow as microprocessor architectures become more complex and software libraries become better designed to integrate easily within applications. Despite the fact that there are various science and engineering applications, the underlying algorithms typically have remarkable similarities, especially those algorithms that are most challenging to implement well in parallel. It is not too strong a statement to say that these software libraries are essential to the broad success of scalable high-performance computing in computational sciences (Kothe and Kendall [2007]; Reed et al. [2005]; Bader [2007]).

The recent trends in hardware development have added additional questions to this scenario because today's codes are not guaranteed to exploit the performance of next-generation hardware to a satisfying degree. The so-called memory wall, i.e., the increasing performance gap between memory access and processor

speed, will force scientific computing software to deal with the efficient use of complex multiple memory hierarchies. This is illustrated in Figure 1.12, which shows how many floating point operations are needed per memory access on various processors to fully utilize their power. We can clearly see the increasing trend over the years. In addition, manycore architectures with hundreds of cores using a decreased processor clock rate will add additional algorithmic and software challenges to scientific computing. Autotuning tools, libraries, and software tools based on dwarfs (Asanovic et al. [2009]) will be designed that will help researchers and application developers to improve the performance of a given application.

In this work we will focus on systems of linear equations that arise at the heart of many scientific and engineering applications. Many of these linear systems are sparse, i.e., most of the elements in the coefficient matrix are zero. Direct methods based on matrix factorizations are sometimes needed to ensure accurate solutions. For example, an accurate solution of sparse linear systems is needed in shift-invert Lanczos to compute interior eigenvalues (Ericsson and Ruhe [1980]; Lin et al. [2021]). The performance and resource usage of sparse matrix factorizations are critical to time-to-solution and maximum problem size solvable on a given platform.

The innermost computational kernels of many large-scale scientific applications and industrial numerical simulations are often either a large sparse matrix problem or a nonlinear optimization method which again can be reduced to a large sparse matrix problem. Sparse direct linear solvers are a core part of many problems in computational science and typically consume a significant portion of the overall computational time required by the simulations. While on one hand modern computer architectures provide larger memory resources and faster multicore processors, on the other hand the need for solving large scale application problems often compensates for these developments. The request for fast and memory efficient — and robust — solvers has been important for many years and remains an open field for further developments.

As mentioned before, the use of graph-pivoting techniques as an alternative approach to traditional pivoting methods emerged two decades ago (Schenk and Gärtner [2004]). These graph-based methods typically build a bipartite graph of a symmetric indefinite matrix A and, by traversing vertices and edges in the graph, these approaches compute a maximum weighted matching that in turn defines a permutation of the rows and/or columns in A. The important advantage of all these methods is that they allow — in a parallel environment — the precomputation of the underlying elimination process, thus making the Gaussian elimination process much more scalable. Almost all modern parallel sparse

direct solver tools (see Table 1.1) are now using these kinds of graph-pivoting techniques, since they are the key for high sequential and parallel efficiency.

For sparse direct algorithms there exist a wide range of possible methods, such as multifrontal, left- or right-looking supernodal methods, or a combination of these methods (Davis [2006]). As a result, different high-level implementations are available, such as SuperLU (Li and Demmel [2003]), MUMPS (Amestoy et al. [2000b]), PARDISO (Schenk and Gärtner [2004, 2006]; Bollhöfer et al. [2020]) and WSMP (Gupta [2002]). The communication patterns of all these different packages and methods are — more or less — very similar. All of them are using supernodal blocking strategies and this data structure will be used for performance modeling in the next chapters.

Table 1.1 lists a few available software packages for the direct solution of sparse linear algebra problems. Comparison of the most popular direct solvers can be found in a study by Gould et al. [2007]. The interest is in software for high-performance computers for solving problems in numerical linear algebra, especially sparse direct systems. We are, in particular, interested in the forward and backward solution process of sparse direct solvers since they build the computational kernel, e.g., in FETI-DP methods. FETI-DP are known to be highly parallelizable, but all implementations are using sparse direct solvers as building blocks on each compute node in order to efficiently solve the coarse grid (Riha et al. [2019]; Meca et al. [2018]; Klawonn and Rheinbach [2010]). Another application domain relying on efficient direct sparse solvers stem from applying interior point methods to problems in the power grid sector (Kardoš et al. [2020]).

We will investigate and analyze the performance of the forward/backward solution process of the PARDISO library (Schenk and Gärtner [2004, 2006]; Bollhöfer et al. [2020]). Detailed performance analysis for a representative sparse solver kernel based on a modified Berkeley roofline model will be presented in the next chapters. In addition to PARDISO, the considerations in this thesis may also be relevant to the sparse triangular solve phases of other solvers like SuperLU (Li [2005]), UMFPACK (Davis and Duff [1997]), or MUMPS (Amestoy et al. [2000a, 2001, 2006]). The performance of sparse direct solvers has, of course, been considered earlier by many different authors, e. g., Heath and Raghavan [1999]; Li [2008]; Marrakchi and Jemni [2017]; Park et al. [2014]; Liu et al. [2016], using hardware relevant at the time of publication. Often, the performance is reported for the proposed implementation on a certain processor or GPU together with related metrics, e. g., data volume or cache misses. More advanced works such as Vuduc et al. [2002] provide upper and lower bounds for the performance. In this thesis, the goal is to apply analytical performance models which allow us

not only to evaluate performance, but also to anticipate and prevent bottlenecks. Traditionally, the Berkeley roofline model (Callahan et al. [1988]; Hockney and Curington [1989]; Schönauer [2000]; Williams et al. [2009]) has been used for this task to relate the performance of a code with the hardware's capabilities. The Berkeley roofline model is introduced in Chapter 2.4. Our modification of the model, which allows us to model a combination of sequential and parallel executions, is described in Chapter 3.3. Finally in Chapter 4 the model is evaluated on various matrices and CPU architectures.

SPARSE DIRECT SOLVERS	License	Type		Language	Mode			Sparse Direct			Last release date
		Real	Complex		Shared	Accel.	Dist	SPD	SI	Gen	
PARDISO	USI	X	X	F77/C/C++	X		M	X	X	X	2020-12-01
SuiteSparse	LGPL/GPL	X	X	C	X	C		X		X	2018-07-05
MUMPS	CeCILL-C	X	X	F77/F95	X		M	X	X	X	2021-04-16
SuperLU	BSD	X	X	F77/F95/C	X	C	M			X	2020-10-17
DSCPACK	PD	X		C	X		M	X			2015-05-23
KKTDirect	PD	X		C/C++	X			LDLT			2010-04-21
Myramath	GPL	X	X	C++	X			X	X		2020-05-20
PaStiX	LGPL	X	X	F95/C/C++	X	C	M	X	X	X	2021-04-08
PSPASES	Own	X		F77/F95/C			M	X			1999-05-09
qr_mumps	LGPL	X	X	F77/F95/C	X	C		X		X	2021-04-2021
Quern	PD	X		C/C++	X					X	2009-02-04
SPARSE	Own	X	X	C	X			X		X	1988-04-01
SPOOLES	PD	X	X	C	X		M			X	1999-04-08
SPRAL	New BSD	X	X	F77/F95/C	X	C		X	X		2016-09-23
TAUCS	Own	X	X	C	X			X		X	2003-09-04
Trilinos/Amesos	LGPL	X		C/C++	X		M	X		X	2017-09-07
Trilinos/Amesos2	BSD	X	X	C++	X		M	X		X	2017-09-07

Table 1.1. List of sparse direct solvers collected by Dongarra and Sukkari [2021].

Chapter 2

Performance modeling of sparse factorization solvers

Performance engineering plays an important role in development of scientific software. In order to achieve optimal performance, one needs to analyze the code and computer architecture to identify bottlenecks and possibilities for optimization (Köstler and Rüde [2013]; Minami [2019]). This helps us spend our effort only where there is a potential for improvement. While simple code can often be analyzed using intuition and guessing, this is not possible for nontrivial codes. Performance models are helping us to analyze a code by guiding our attention on important features and neglecting unnecessary details (Gropp et al. [1999]; Williams [2008]; Kreutzer et al. [2015]). This chapter reviews two performance models we will later use for modeling forward and backward substitution code used in sparse factorization solvers.

The chapter is organized as follows. In Section 2.2 computer architecture is briefly introduced. Next, Section 2.3 reviews a collection of command line tools called LIKWID. These tools are useful for various performance engineering tasks, e.g., determining the hardware and its specification, microbenchmarking, and profiling. After that, two established performance models are reviewed: the Berkeley roofline model in Section 2.4 and Erlangen execution-cache-memory model in Section 2.5. Finally in Section 2.6 the execution-cache-memory model is used to analyze performance of two simple computational kernels. Models of these two kernels will be used later as a base for modeling sparse triangular solves.

2.1 Exploiting the memory hierarchies, autotuning research, and sparse factorization solvers

Modern microprocessors are highly sensitive to the spatial and temporal locality of data. Regardless of the programming model, performance of future parallel applications will crucially depend on the quality of the generated code which is traditionally the responsibility of the compiler. The compiler selects which optimization to perform in terms of, e.g., loop-unrolling, out-of-order instruction capabilities, or register scheduling. Choosing parameters for these optimizations, and selecting among alternative implementations, is the key to efficient use of the underlying hardware. The resulting space of optimization alternatives is large (Balaprakash et al. [2018]).

Autotuner projects such as, e.g. ATLAS [1] (Whaley and Petitet [2005]; Whaley et al. [2001]), FLAME [2] (Bientinesi et al. [2005]), and OSKI [3] (Demmel et al. [2005]; Vuduc [2003]; Lee et al. [2004]), gained popularity as a very effective approach for producing high-quality portable scientific code. In these projects, the set of library kernels is automatically optimized by generating many variants of a given kernel and by running that kernel in a target platform. The search process may take hours to complete on the platform. However, it needs to be performed only once when the library is installed on the platform. The resulting codes might be several times faster than naive implementations.

As an example, reordering the vertices and elements in a mesh can have a significant impact on performance. Graph and hypergraph techniques have been used in this area to automatically generate highly efficient, platform-adapted implementations of *sparse matrix kernels*. These kernels are frequently computational bottlenecks in diverse applications in computational science and engineering applications. However, the task of extracting near-peak performance on modern cache-based superscalar machines has proven to be extremely difficult. Many sparse matrices from applications have a natural block structure that can be exploited by storing columns as a collection of blocks and thus accelerating the performance of sparse matrix kernels significantly. It is shown in recent projects such as Azad et al. [2016]; Vuduc [2003] that it is possible to build an automatic tuning system to generate implementations whose performances exceed that of the best hand-tuned code. The algorithmic methods behind these research projects are based on reordering methods on the discrete graph structure

[1] Automatically tuned linear algebra software.
[2] Formal linear algebra methods environment.
[3] Optimized sparse kernel interface.

in order to maximize the block structure and, thus, obtaining high performance by maximizing spatial and temporal locality.

2.2 Computer architecture

Modern processors utilize numerous techniques that aim to improve performance — the number of computations done per unit of time. In the past, the performance used to be increased only by increasing the CPU clock frequency. However, increasing the frequency results in higher power consumption and, consequently, the CPU needs a larger heat sink in order to prevent it from overheating. As a result, the clock frequency of recent CPUs is about 2.0–3.5 GHz and does not increase anymore. Since the trend of doing computations faster due to increasing frequency is over, the effort of CPU manufacturers focuses on performing computations at the same time in parallel. This leads to adding more computational units to modern processors. These units, called cores, perform instructions independently, thus the overall performance is effectively increased. This scheme works well when every core works on different data, but when an access to shared data is required the cores need to be synchronized in order to prevent data conflicts and inconsistent results. The synchronization, however, requires serialization of computation and prevention parallel computations, thus reducing the overall performance.

Another possibility for parallel computations is implementing instructions that perform more than one operation applied on vectors of data instead of single elements. Such instructions are called SIMD (single instruction multiple data), examples of which are SSE (streaming SIMD extensions) which operates on 128 b registers containing two floating point numbers in double precision or four numbers in single precision, or newer AVX (advanced vector extensions) with registers twice as big, computing four operations in double precision in a single instruction.

Another CPU optimization focuses on computational pattern, where there is multiplication followed by addition. An example of such an operation can be found in many compute kernels in various scientific applications. In order to make these computations more efficient, many modern CPU architectures compute operations similar to $a = a + b * c$ in one instruction. This type of instruction is called fused multiply-add (FMA). The new version of an AVX extension, called AVX2, adds support for vectorized FMA operations, effectively performing 8 operations in a single instruction using double precision (16 operations in single precision).

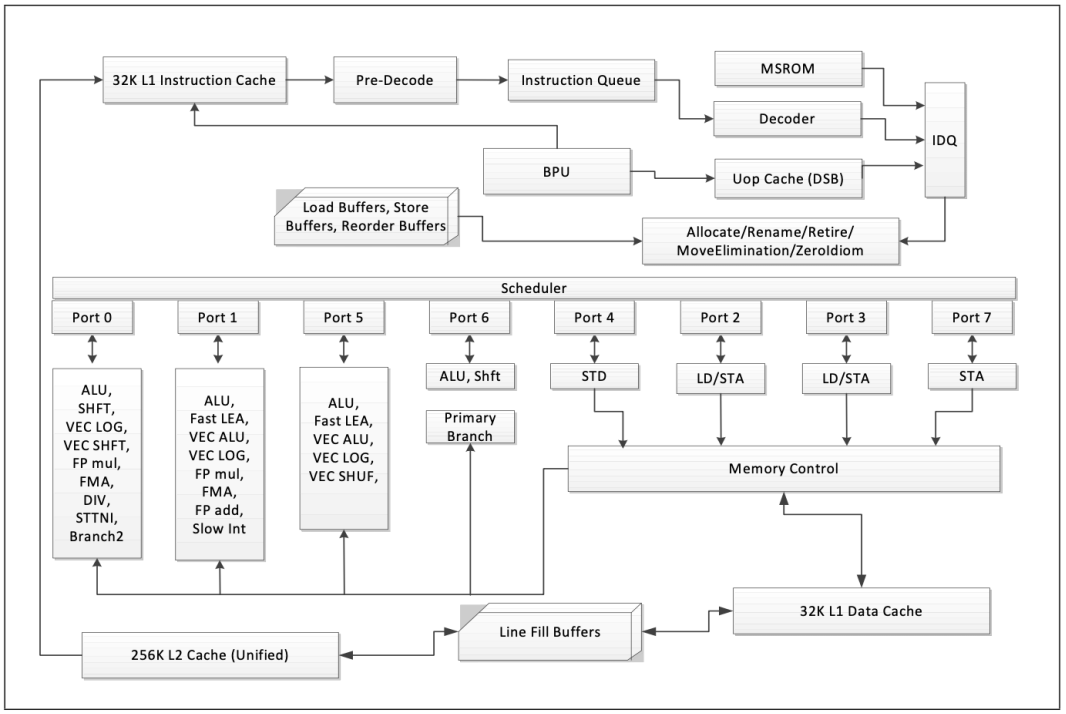


Figure 2.1. CPU core pipeline of Haswell microarchitecture. From Intel Corp. [2016].

The parallelism can be exploited not only on the level of multiple cores, but also on the instruction level within a single processor, an example of which is instruction pipelining. Pipelining attempts to divide incoming instructions into a series of sequential steps, allowing the processor to dispatch a new instruction every cycle, instead of performing a whole instruction which takes several cycles. Additionally, in order to maximize instruction throughput, the instructions are executed out of order, if there are available processing units. A CPU core of Intel Haswell microarchitecture is shown in Figure 2.1, consisting of eight dispatch ports in total, four of which have units for computational operations and four for memory operations. The scheduler can dispatch up to eight micro-ops every cycle, one on each port.

Another level of complexity is introduced when considering data movements. The CPU performs all operations on data stored in registers, the fast but small memory inside the CPU. Before the computations start, data have to first be loaded from memory to registers and when the computations are done, they have to be restored back to memory. The memory is very slow compared to the registers, which introduces a significant bottleneck hampering the performance

of a computation. To make the memory access more efficient there are multiple levels of memory between the CPU and main memory, each with a different size and access speed, referred to as cache. Usually the cache has three levels, called L1, L2, and L3. L1 is closest to the CPU and has the smallest size (few kB), while L3 is closest to the memory and has the largest size (few MB). Any data transferred between the memory and registers have to go through all cache levels. In some cases data are often reused in the computation, thus they can stay in the cache, avoiding communication with the slow memory.

In order to improve performance and the ability of the system to be expanded, e.g., in the case of servers often having more than one processor, every processor has its own memory and memory controller. This memory layout is called NUMA (non-uniform memory access). The benefit of such an architecture is that it increases memory bandwidth, the rate at which data can be read and stored into memory, which prevents a bottleneck introduced by all the cores accessing the memory and saturating the available bandwidth, resulting in idle cores waiting for data instead of doing useful computation. On the other hand, introducing multiple memory domains creates problems when one processor needs to access data in memory belonging to another processor. This situation is possible but not very efficient, as it increases the load at a single memory controller and, additionally, the data have to be transferred through a link between processors, which can become a bottleneck. The programmer works with a virtual memory spanning all memory domains, so he does not see this underlying complexity, but he should keep this in mind in order to write efficient code. For example, an operating system places memory pages to the physical memory of the processor that writes to the page for the first time, so called first touch. If a single thread initializes all the data that are then accessed in parallel, it might allocate the data in a suboptimal way. If the initialization is performed in parallel, significantly better memory utilization can be achieved.

Modern server processors contain a lot of cores (usually 14 or more) and with so many cores accessing the memory it is very easy to saturate the memory bandwidth. Intel solves this problem by splitting the memory and L3 cache between two NUMA domains and use second memory controllers. This reduces the number of cores using the same memory controller and doubles the memory bandwidth. Intel calls this architecture cluster-on-die (CoD).

Peak performance of a CPU is a theoretical maximum the processor can achieve. It requires utilization of all cores running at the base frequency and every core achieving the highest possible floating point throughput. Evaluation of the peak performance of modern CPUs is a very intricate process, requiring one to consider a multitude of factors. It depends on an instruction set and the num-

name		IVB	HSW-D	HSW-S	BDW	SKX	KNL	ZEN-D	ZEN-S
processor name		Intel Xeon E5-2660 v2	Intel Xeon E3-1240 v3	Intel Xeon E5-2695 v3	Intel Xeon E5-2630 v4	Intel Xeon Gold 6148	Intel Xeon Phi 7210	AMD Ryzen 7 1700X	AMD EPYC 745
micro arch.		Ivy Bridge	Haswell	Haswell	Broadwell	Skylake	Knigths Landing	Zen	Zen
freq	[GHz]	2.2	3.4	2.3	2.2	2.4	≈ 1.3	3.4	2.3
cores		10	4	2 × 7	10	20	64	8	24
ISA		AVX	AVX2	AVX2	AVX2	AVX-512	AVX-512	AVX2	AVX2
NUMA LDs		1	1	2	1	1	1	1	4
L1	[KiB]	32	32	32	32	32	32	32	32
L2	[KiB]	256	256	256	256	1024	1024	512	512
L3	[MiB]	25	8	2 × 17.5	25	28	-	2 × 8	8 × 8
scalar read bw.									
1 core	[GB/s]	9.5	16.6	12.1	11.5	14.5	8.5	19.3	19.3
NUMA LD	[GB/s]	44.4	22.7	31.2	56.3	108.0	75.2	33.7	37.6
scalar ADD+MUL/FMA									
1 core	[F/cy]	2	4	4	4	4	4	4	4
NUMA LD	[F/cy]	20	16	28	40	80	256	32	24
scalar machine balance B_m									
1 core	[B/F]	2.2	1.2	1.3	1.3	1.5	1.6	1.4	2.1
NUMA LD	[B/F]	1.0	0.4	0.5	0.6	0.6	0.2	0.3	0.7

Table 2.1. Details of evaluated hardware systems. KNL's bandwidth numbers are for DDR memory.

ber and type of units in a core. A detailed analysis of different Intel architectures can be found in Dolbeau [2018]. As an example we can analyze Intel Xeon E5-2695 v3. This is a 14 core CPU with Haswell architecture and base frequency of 2.3 GHz (detailed description is in Table 2.1). The Haswell architecture supports AVX2 instructions (8 FLOPs per instruction) and can execute two instructions per cycle (ports 1 and 5), resulting in a total of 16 FLOPs per cycle per core. Consequently, the peak performance is $P_{peak} = 2.3 * 14 * 8 * 2 = 515.2$ GFLOPs/s.

The theoretical memory bandwidth can be found in a datasheet, but the attainable bandwidth of a specific application depends on the access patters, number of threads used, whether NUMA is used, and other factors. Using the theoretical bandwidth is thus not precise for purposes of performance analysis. A more realistic bandwidth can be obtained using microbenchmarks, ideally with similar memory access patterns as the application. In the following section we will review a performance tool called LIKWID which supports software developers, benchmarkers, and application users to model and get the best performance

on a given system. We will use this tool later for performance modeling for a sparse factorization solver.

2.3 Performance modeling using the LIKWID tools

LIKWID ("like I knew what I'm doing") (Treibig et al. [2010]) is a set of command line tools to support optimization and performance engineering. It consists of tools that display thread and cache topology, discover CPU and memory performance using benchmarks, alter CPU frequency and other settings, and allow performance evaluation of user applications. Typical workflow using the LIKWID tools could be (i) use `likwid-topology` to find information about the hardware; (ii) gather various performance metrics using `likwid-bench` (memory bandwidth, performance using different instruction sets, etc.). Next, (iii) run an application specifying thread affinity using `likwid-pin`; and (iv) evaluate performance of a user application using `likwid-perfctr` by collecing performance metrics either for the whole runtime or only for a specified code region. A short description of the most useful tools follows.

likwid-topology

Performance engineering requires in-depth knowledge about node topology, like NUMA domains, cache hierarchy and sizes, CPU architecture, frequency, cores, and HW threads. This information can be gathered using various command line tools, which might be time consuming, especially since some of them present long output where it is difficult to find required information. `likwid-topology` provides a holistic picture of node topology, collecting information from different available sources in the operating system and presenting a comprehensive and easy to understand overview of the node topology either in intuitive text form or an ASCII art style.

likwid-pin

Thread affinity is crucial for performance of scientific applications. Knowing the topology (obtained, for example, by `likwid-topology`), one can pin threads to cores according to the application's requirements. Pinning threads make performance measurements more consistent between runs and usually also improve performance bacause the threads are not assigned to cores randomly. This is very important when multiple NUMA domains are used to minimize the data

traffic between NUMA domains. `likwid-pin` can be used with all applications based on POSIX threads, which include most of OpenMP implementations. The pinning is achieved by overloading the `pthread_create` call and pinning every thread upon creation. Some OpenMP implementations create shepherd threads. These threads do not execute any user code and therefore should not be pinned. This is achieved by providing a mask or using one of the predefined masks for known implementations.

likwid-perfctr

All modern CPUs provide hardware counters, registers that can be set to count various hardware events. The main purpose of these counters is to help manufacturers with development of the CPU, but they are also available to the user. The counters are accessed using model specific registers (MSRs) and can be configured to count various events like fetching data, storing data, cache hits/misses, or calculations. As the counters are implemented in hardware, they come with no overhead. A downside of using the counters is that they measure events on a given core and cannot distinguish between processes. However, on systems with only one user this is usually not a problem.

`likwid-perfctr` is a command line tool that configures and reads the counters and is used as a wrapper for a user application. It can be used in two modes: it can either measure the whole runtime of the application or only specific blocks of code called regions. If the whole runtime is measured, no modification of the application is needed. For measuring the regions, markers denoting the beginning and end of a region have to be inserted into the application code and have to be linked with the likwid library. The markers are library function calls that read values of the counters. Every marker has a name (user defined string) identifying the region. The string does not have to be unique, but all regions with the same name are summed up and are indistinguishable in the output. After the user application finishes, the data from counters are processed and a summary is presented. As reading the counters is a simple operation and all the processing is done after the user app finishes, there is very little overhead.

The measured events are specified as command line arguments to the wrapper application, so the measurement can be repeated several times measuring different events without recompiling the code. The raw counts usually do not provide a lot of insight. However, `likwid-perfctr` can combine the raw counts to derive useful performance metrics like FLOPs/s or memory bandwidth. Additionally, `likwid-perfctr` also has the functionality of `likwid-pin`. This is very important because without pinning the threads could move between cores, making the measurements inaccurate.

likwid-bench

Writing a benchmark is a very intricate process. The purpose of a benchmark is to measure some specific computation; however, the naive user code might not necessarily be the most efficient implementation. A compiler exploits a lot of optimizations, e.g., if results of some computations are not used, these computations are often dropped altogether. Alternatively, a compiler can replace the user code by a more efficient implementation. In both cases, the benchmark would end up measuring something different from what was intended. To be sure the benchmark measures the proper thing, the code most likely has to be written in assembly or at least the user should check the assembly generated by a compiler.

`likwid-bench` is a benchmark suite for prototyping low-level assembly kernels. It contains a set of various kernels, for example, dot product, daxpy, load, store, copy, and many others. It allows the user to define his own kernels. The kernels are defined as text files that are compiled during compilation of the suite. And the suite takes care of everything else: running the kernel on a problem of a given size, on a given number of threads, and presenting results to the user.

2.4 Berkeley roofline model - a performance model for multicore architectures

Programmers often do not need an understanding of every detail of CPU design. They should focus on general concepts rather than details of every available architecture. A tool that can provide a simplified model of the CPU hiding most of the architecture specific complexity is very valuable. Probably the most popular tool was popularized by Williams et al. [2009]. This tool is called the roofline model. It became very popular because it hides most of the CPU complexity and presents an intuitive and easy to use model that can guide performance engineering. This model has proven to be useful not only on the most common architecture, x86_64, where it was extended to address cache hierarchy (Marques et al. [2020]), but it was successfully used also on other architectures, e. g., ARM, an architecture developed for battery powered devices, where power efficiency is the biggest concern, but since then it found its way to desktop computers and even the most powerful supercomputers (Alappat et al. [2020]); GPU accelerators (Ding and Williams [2019]; Yang et al. [2020]); Intel Xeon Phi (Druinsky et al. [2016]) and even FPGA (Nguyen et al. [2020]). Moreover it was also used for modeling energy consumption (Choi et al. [2013]).

The model analyzes bottlenecks during execution on a given hardware. A

compute kernel reads data from memory, does some computations, and writes the results back to memory. Let's denote W as the size of data read or written to memory in Bytes and F the number of operations the kernel computes. Usually we are interested in floating point operations (FLOPs), additions and multiplications, but we could generalize this to any kind of operations. Let's assume the machine has peak performance P_{peak} measured in [FLOPs/s] and peak memory bandwidth b_s measured in [B/s]. It is reasonable to expect the processor needs F/P_{peak} s to finish all computations and it needs W/b_s s for the memory transfer (reading and writing data). If we assume the computation and memory transfer can perfectly overlap, the total runtime is equal to the one that take longer:

$$T = \max(F/P_{peak}, W/b_s). \tag{2.1}$$

Working with runtime is not very useful because it depends on the size of the problem. Instead we usually compare performance $P = F/T$ [FLOPs/s]. Let's also define arithmetic intensity as the number of operations per one byte of memory transfer, $I = F/W$. Then we can write the expected performance as

$$P = min(P_{peak}, I \cdot b_s). \tag{2.2}$$

Sometimes it is easier working with code balance instead of arithmetic intensity. Let's define code balance as the number of transferred bytes per one operation, $B_c = W/F = I^{-1}$. Then we can write (2.2) as

$$P = min(P_{peak}, b_s/B_c). \tag{2.3}$$

We can see a graphical representation of (2.2) in Figure 2.2 in a blue color. The shape of the graph looks like a roof, which gives the model its name. The performance bound increases with increasing arithmetic intensity until it saturates at the peak performance. This happens where the two lines corresponding to the two bottlenecks intersect. The arithmetic intensity of this intersection equals P_{peak}/b_s or machine balance $B_m = b_s/P_{peak}$. We call it machine balance and not code balance because it characterizes the machine.

In Figure 2.2 we can also see three vertical lines. These lines represent three generic kernels with arithmetic intensity 1, 4, and 16. We can expect the performance of these kernels somewhere along the respective lines. Note that the lines are below the roofline, since the roofline is the performance upper bound. The red kernel (arithmetic intensity 1) is in the bandwidth limited region. The performance is increasing with increasing arithmetic intensity. Looking at the intersection of the red and blue line, we can expect a performance bound of approximately 44 GFLOPs/s. The brown kernel (arithmetic intensity 16) is in the

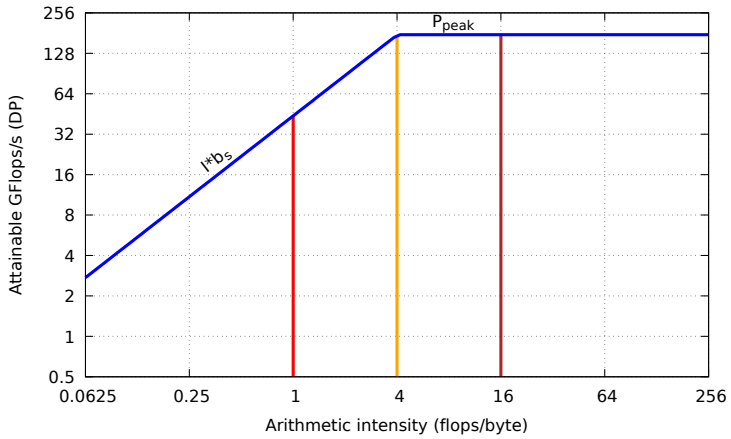


Figure 2.2. Roofline model of Intel Xeon E5-2660 v2 (blue) and three generic kernels (red, orange, and brown).

compute bound region. With increasing arithmetic intensity the memory traffic decreases, but the performance does not increase. And the orange kernel (arithmetic intensity $4 \approx B_m^{-1}$) is between these regions. Performance of this kernel depends on machine ability to overlap computation and memory communication (loads and stores).

2.4.1 Roofline ceilings

For realistic applications the measured performance of a kernel is often far below the roofline. The CPU and memory architecture utilize several paradigms that improve the performance, as described in section 2.2. To get close to the maximum performance the kernel needs to exploit these. Failing to do so results in a huge performance penalty. For both bounds the roofline model takes into account, in-core execution and memory bandwidth; we can show ceilings showing impact on performance when certain optimizations are not implemented.

In-core roofline ceilings

As discussed in section 2.2 processors achieve peak performance when all cores are utilized and all of them are using only the instructions that perform the most operations. These are usually the vector instructions AVX on Haswell architecture and newer AVX2 (FMA operation applied on a vector). If the compiler is unable to use these instructions, it comes with a huge penalty in performance. If scalar instructions are used, only 1/4 of peak performance can be achieved. If FMA is

not used, performance drops to 1/2. And if neither AVX nor FMA is used, then we can expect only 1/8 of peak performance.

For example, the following code snippet computes sum of elements in a vector:

```
1: for i = 0; i < n; i++ do
2:     sum += a[i]
3: end for
```

It may seem this code cannot be vectorized because it would cause write conflicts in the variable `sum`. But the compiler can transform this code introducing new variables.

```
1: for i = 0; i < n; i+=4 do
2:     sum0 += a[i ]
3:     sum1 += a[i + 1]
4:     sum2 += a[i + 2]
5:     sum3 += a[i + 3]
6: end for
```

In this code there are no dependencies anymore, so it can be vectorized. This means the four consecutive elements from the array fit into a 256 b AVX register and the same for the sum variables. Then the AVX instruction computes all four operations at the same time, achieving 4 FLOPs per instruction.

Another example is a prefix sum. In this case there are loop-carried dependencies preventing vectorization:

```
1: for i = 1; i < n-1; i++ do
2:     a[i] += a[i-1]
3: end for
```

So while in the first case the loop can be vectorized and achieve 4 FLOPs/instruction, in the second example vectorization is not possible, achieving only 1 FLOP/instruction with scalar instructions. Note that both algorithms use only additions and no multiplications, so FMA is not used. As a result the best performance one could expect is $P_{peak}/2$ for the first code and $P_{peak}/8$ for the second one. Visualization of some in-core ceilings is given in Figure 2.3.

Bandwidth roofline ceilings

In Figure 3.5a we can see memory bandwidth achieved by two Haswell processors (desktop and server) using a different number of cores. The desktop processor nearly saturates the bandwidth with one core, but the server processor achieves only about 40 % of bandwidth with a single core and needs at least 3

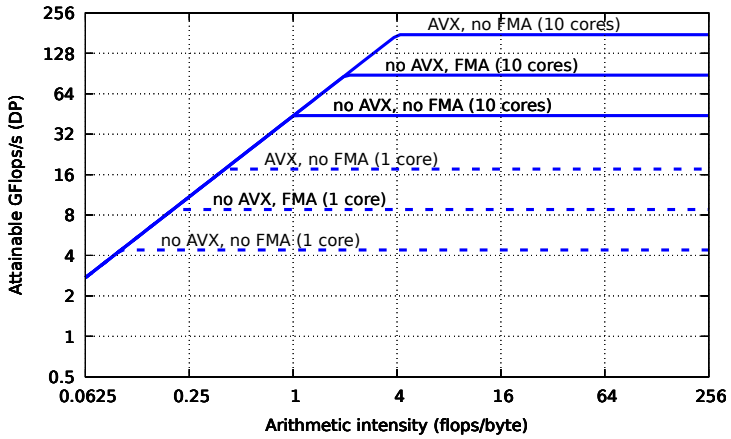


Figure 2.3. Berkeley roofline model of Intel Xeon E5-2660 v2 with various in-core ceilings for 1 and 10 cores.

or 4 cores to fully saturate the bandwidth.

When data are transferred between memory and the L3 cache or different cache levels, they are always transferred in chunks called cache line. On Intel processors the size of the cache line is usually 64 B. Even if only 1 B is needed, the whole cache line is transferred. Or when data are accessed with nonunit stride, some data are transferred and not used. This can lead to saturating the memory bandwidth while getting little useful data.

When NUMA is used, there is a memory controller at every NUMA domain. If all data are allocated on the same controller (this can happen, for example, by wrong first touch (initializing the data sequentially)), then the other controllers are not used, lowering the bandwidth. Also when processes from one domain access data from another domain, the communication goes through a NUMA link between domains, which can become a bottleneck.

In Figure 2.4 we can see the effect of lower bandwidth when single core is used on the roofline model. As the bandwidth is lower, the corresponding line moves down. Note also the intersection of the horizontal and skewed line. It moved from arithmetic intensity 4 F/B to about 16 F/B, so a kernel that is compute bound on 10 cores could become memory bound on 1 core.

As it is not possible reading data from memory using one thread and using all available threads for computations or vice versa, we can combine Figures 2.3 and 2.4. The roofline model with both in-core and bandwidth ceilings is shown in Figure 2.5.

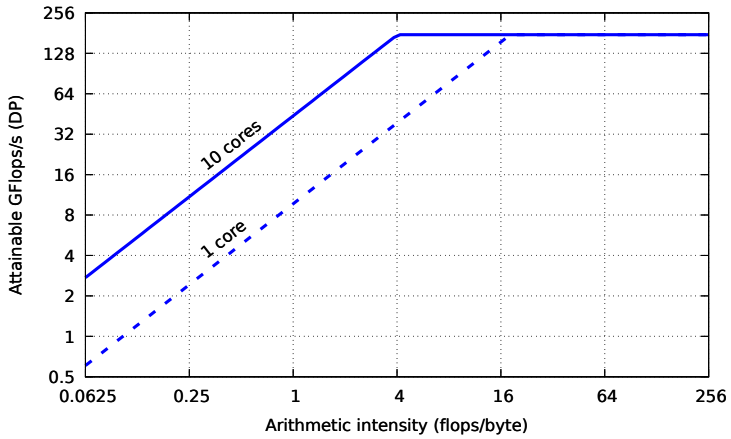


Figure 2.4. Roofline model of Intel Xeon E5-2660 v2 with bandwidth ceilings for 1 and 10 cores.

2.5 Erlangen Execution-cache-memory model

The Roofline model is a simple model for performance prediction in the saturated case. Hereby it is assumed code is limited by floating point performance or by the memory bandwidth. The Execution-cache-memory (ECM) performance model (Treibig and Hager [2010]; Hager et al. [2016]) is a refinement of the roofline model.[4] In contrast it allows a performance prediction on the single core level as well as a scaling prediction over the socket. The model takes into account the duration of the code execution inside the core separated by arithmetic and data movement. Furthermore data transfers in the memory/cache hierarchy are considered as well as the achievable memory bandwidth. Finally both parts build the single core model, which is used to determine the scaling behavior over the cores until a bottleneck is reached. For memory bound codes this is typically the achievable memory bandwidth, as all other infrastructures like Intel's L3 cache on Ivy Bridge, Haswell, and Broadwell scale perfectly.

The ECM model has some restrictions on the code to be analyzed. It is important that streaming accesses are performed. This means prefetching works perfectly and can hide latency effects. The ECM model predicts the number of CPU cycles (cy) required to execute a certain number of iterations of a given loop on a single core. Since the smallest amount of data transferred between cache levels is one cache line (CL), it is a reasonable unit of work for the predictions. The size of the cache line on Intel processors is 64 B, which is for streaming

[4]For further details regarding the ECM model refer to Stengel et al. [2015].

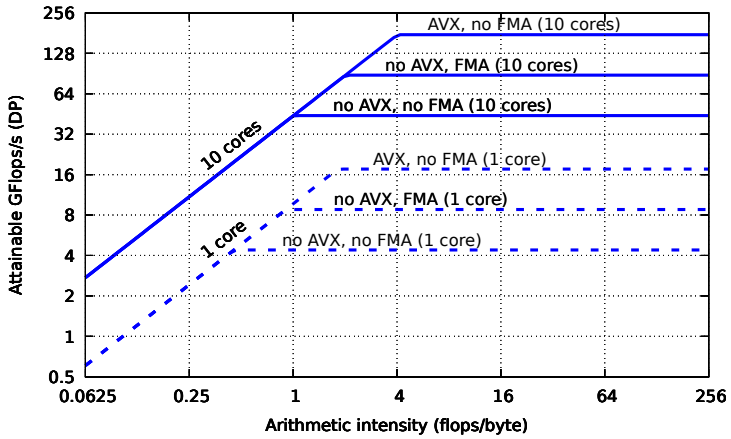


Figure 2.5. Roofline model of Intel Xeon E5-2660 v2 with both in-core and bandwidth ceilings.

kernels 8 iterations with floating point numbers in double precision.

To construct the model, we consider two parts separately: the in-core execution, assuming all data were already fetched to the L1 cache and there are no cache misses, and the time to fetch the data from its location to the L1 cache.

In-core execution

To determine the in-core execution time, a simple model for instruction throughput on the given architecture is required. Figure 2.1 shows a port model for the Intel Haswell architecture. The port scheduler schedules instructions to ports independently out of program order, making sure all data dependencies are met.

Instructions inside ports are pipelined, but only one instruction per port can be issued per cycle. Haswell can perform two loads (LD, ports 2*D* and 3*D*) and one store (ST, port 4) of sizes up to 32 B at the same time (Intel Corp. [2016]), each. Each load and store requires an address to be generated by an address generation unit (AGU, ports 2, 3, and 7). However on port 7 only a simple AGU is located, which is limited to simple addressing modes[5] (Intel Corp. [2016]; Hofmann et al. [2016]). For floating point operations, the cores host two FMA units (ports 0 and 1), two multiplication (MUL) units (ports 0 and 1), and one add (ADD) unit (port 1).

For the ECM model the duration of the execution inside the core is split into

[5]AGUs on ports 2 and 3 support addressing "base plus index plus offset," AGU on port 7 supports only simple addressing mode "base plus offset."

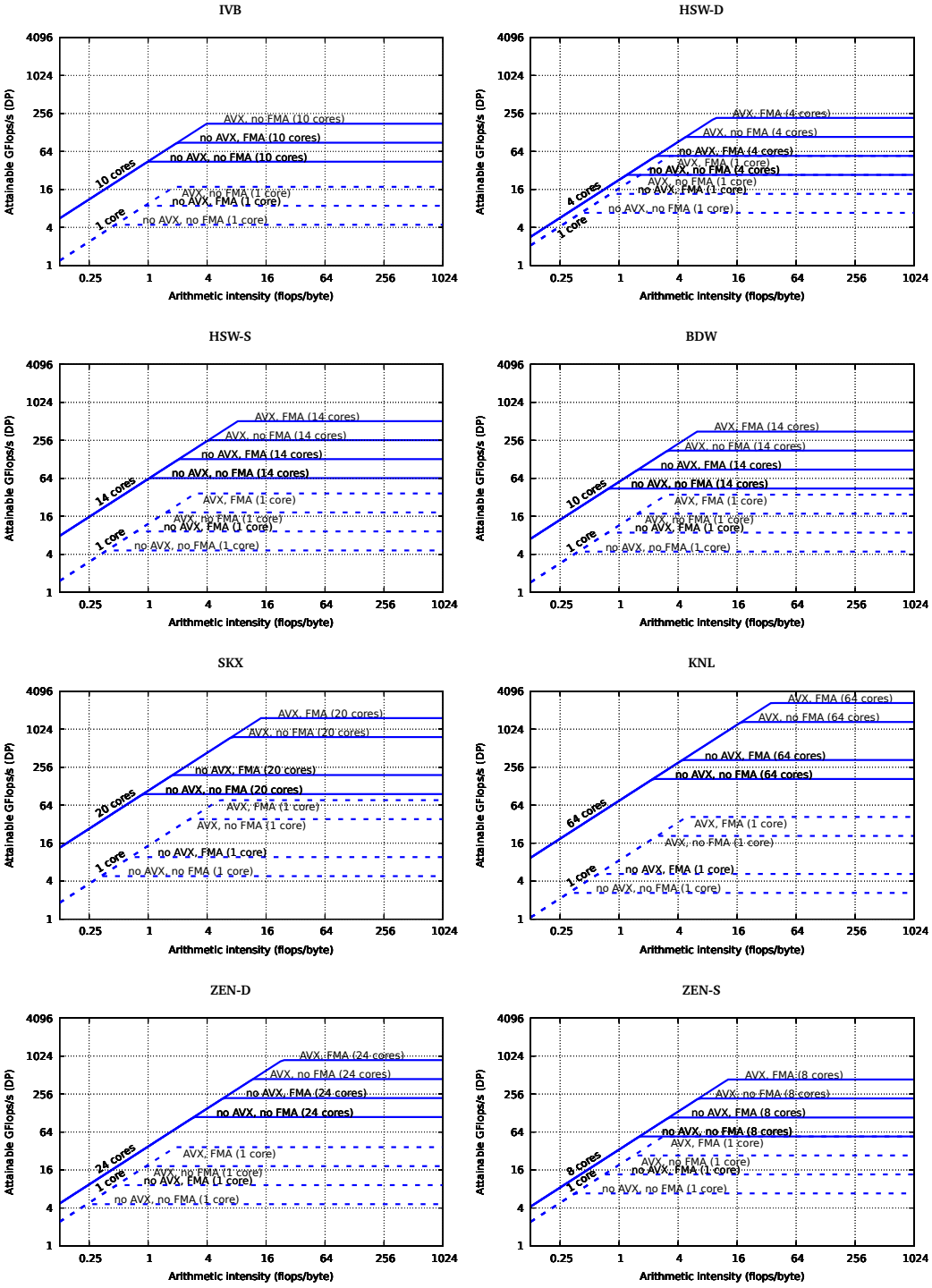


Figure 2.6. Roofline models of processors listed in Table 2.1.

two categories. The first one comprises only data movement between registers and the L1 cache, i.e., loads and stores occurring on ports $2D$, $3D$, and 4. The second category consists of the rest, which can possibly overlap with the loads and stores, like arithmetic and logic. The execution in-core time T_{core} is set to the time of the port with the longest execution time.

Data transfers through the memory hierarchy

Data that are not present in the L1 cache must be fetched from lower levels and modified data evicted to make room for new cache lines. On Intel architectures, transfer of one cache line between adjacent cache levels takes 2 cycles. Transfer time of one 64 B cache line can be computed knowing the memory bandwidth b_s and clock frequency f as $64 * f / b_s$ cycles.

As b_s, one could take the nominal memory bandwidth. However, this bandwidth is practically never reached and depends strongly on the access pattern used. This is caused by the organization of the memory subsystem, where e. g., banking conflicts and DRAM page misses impair performance. With detailed knowledge about the internals of the memory controller (scheduling strategies, thresholds for strategy switching, ...) and DRAM modules this could also be modeled, which is far from being trivial (Jacob et al. [2007]). As this is beyond the scope of the ECM model, typically, a microbenchmark resembling the used access pattern by the code under investigation is used to measure the attainable bandwidth and use it as input for the model.

2.6 ECM model application

2.6.1 DAXPY vector addition

In the following text we give a brief introduction to the ECM model by analyzing a simple daxpy-like kernel on the HSW-S system, whose processor is based on the Intel Haswell microarchitecture. The daxpy kernel to be analyzed is

```
for (int i = 0; i < N; ++i)
  r[i] +=  s * l[i];
```

The vectors `r` and `l` are double-precision floating point vectors with `N` elements each. Furthermore `s` is a scalar double-precision floating point variable. The code is vectorized via AVX and FMA3 instructions by the compiler and additionally 4-way unrolled to reach full performance. The unrolled code becomes

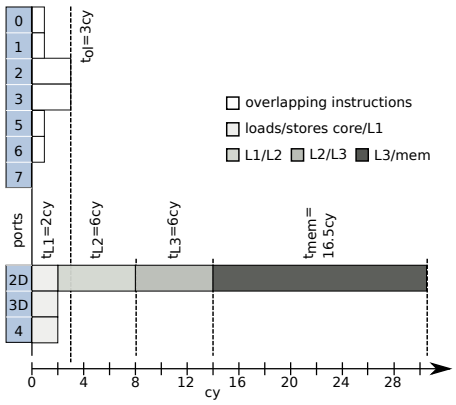


Figure 2.7. Runtime contributions of different execution ports and cache/memory hierarchy levels for eight iterations of the daxpy kernel on HSW-S (Haswell microarchitecture). It seems that the simple AGU on port 7 is not used and all the addresses are generated on ports 2 and 3.

```
for (int i = 0; i < N; i += 4)
{
  r[i  ] +=  s * l[i  ];
  r[i+1] +=  s * l[i+1];
  r[i+2] +=  s * l[i+2];
  r[i+3] +=  s * l[i+3];
}
```

For easier modeling, we take as many iterations into account as are needed to process a whole cache line[6]. Hence, for the daxpy kernel with double-precision floating point numbers the work package we model is eight iterations (or two iterations of the unrolled loop). To process these eight iterations with AVX and FMA3, four 32 B AVX loads (`r[:]` and `l[:]`), two 32 B AVX stores (`r[:]`), and two FMAs are required. Hereby we define the "work" performed by eight iterations is to transfer $W = 192\,\mathrm{B}$. In order to determine the duration of the execution of the code inside the core we assume all operands reside in the L1 cache.

The duration of the in-core execution depends on the core's architecture. For HSW-S it's the Haswell microarchitecture. The superscalar design has several ports with different execution units for different types of instructions, shown as part of Figure 2.1. Instructions scheduled to different ports run independently. The instruction scheduler takes care that no data dependencies are violated.

Haswell can perform two loads (ports 2*D* and 3*D*) and one store (port 4)

[6]On Intel architectures, the length of the cache line is 64 B.

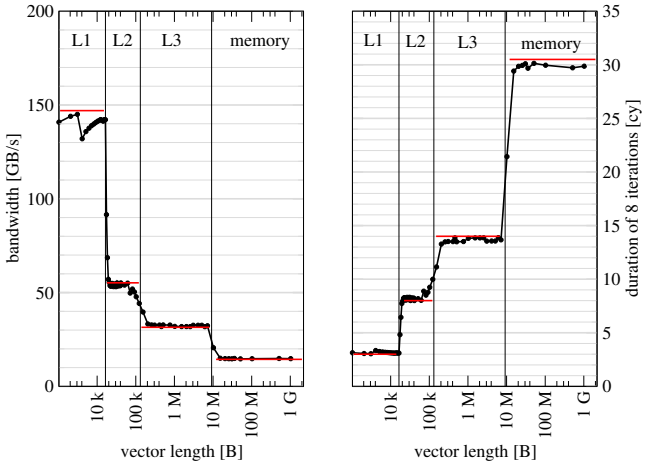


Figure 2.8. Memory bandwidth (left) and duration in CPU cycles of 8 iterations, i.e., two iterations of the compiler generated loop (right), for the daxpy kernel on HSW-S: measurement (black) and ECM prediction (red). As the vectors are getting larger and cannot fit in the higher cache levels, the number of cycles for performing 8 iterations increases, which means the performance decreases. Note that the total working set required for vectors `r` and `l` is twice the AVX vector length.

of sizes up to 32 B at the same time (Fog [2016]). This constellation requires the store address to be simple addressing as only in this case can the address generation unit (AGU) on port 7 be used. For relevant floating point operations, it hosts two FMA units (ports 0 and 1) two MUL units (ports 0 and 1), and one ADD unit (port 1).

Under these considerations two iterations of the compiled loop would require 1 cy for the FMA (ports 0 and 1), 2 cy for the four AVX loads (ports 2 and 3), and 2 cy for the four AVX stores (port 4). 6 addresses need to be generated. This takes 2 cy, if the simple AGU on port 7 is used. Instructions regarding index counter increments are for this case negligible, hence we ignore them. The distribution of the instructions over the ports is found in Figure 2.7.

For the ECM model the duration of the execution inside the core is split into two categories. The first one comprises only data movement between registers and the L1 cache, i.e. loads, stores occurring, and address generation. The second category consists of the rest, which can possibly overlap with the loads and stores, like arithmetic, logic. The maximum duration t_{L1} of the first class (ports 2, 3, 4, and 7) is

$$t_{\mathrm{L1}} = 2\,\mathrm{cy}. \tag{2.4}$$

The maximum duration of the latter class (ports 0, 1, 5, and 6), t_{ol}, is

$$t_{\mathrm{ol}} = 1\,\mathrm{cy} \tag{2.5}$$

(normalized to eight iterations) as the maximum duration on the ports overlapping with data movements. The performance P_{L1}, when all data are fetched from L1, is then computed as

$$P_{\mathrm{L1}} = \frac{W}{\max(t_{\mathrm{ol}}, t_{\mathrm{L1}})} f, \tag{2.6}$$

where W denotes the loop specific work performed and f clock frequency of the core. For this loop with $W = 192\,\mathrm{B}$ and $f = 2.3\,\mathrm{GHz}$ we have $P_{\mathrm{L1}} = 220.8\,\mathrm{GB/s}$. Figure 2.8 shows ECM prediction and measured performance for the DAXPY kernel of various vector lengths. The graph on the left shows performance and the graph on the right the number of CPU cycles needed to perform 8 iterations of the loop. We can see that larger vectors that do not fit into the L1 cache and need to be stored in the L2 or L3 cache or even the main memory require more cycles, which results in worse performance. The measurements for the L1 cache reveal that only around 145 GB/s are reached. If the simple AGU on port 7 is not used, the store addresses are generated by the two remaining AGUs. This causes t_{ol} to be increased by 1 cy and become the new bottleneck. This results in a corrected

$$t_{\mathrm{ol}} = 3\,\mathrm{cy} \tag{2.7}$$

with $P_{\mathrm{L1}} = 147.2\,\mathrm{GB/s}$, which is in line with the measurements as shown in Figure 2.8.

Modeling the performance when data reside in different cache levels from L1 requires analyzing data transfers between these levels. For daxpy this is straightforward as vectors `r` and `l` are streamed from/to memory and no cache reuse takes place. Throughout the cache/memory hierarchy we transfer between each cache level three cache lines (cl) for each iteration: load 1 cl of `l`, load 1 cl of `r`, and store 1 cl of `r`.

Transferring a cache line between L1/L2 and L2/L3 takes 2 cy each. Hence it takes

$$t_{\mathrm{L2}} = 6\,\mathrm{cy} \qquad \text{and} \qquad t_{\mathrm{L3}} = 6\,\mathrm{cy} \tag{2.8}$$

to transfer our three cache lines, respectively. For computing the performance P_{L2} and P_{L3}, when data reside in the L2 or L3 cache, respectively, we have to add t_{L2} and t_{L3} to the duration of the data path, as on the considered Intel architectures data transfers seem be be serialized when streaming accesses occur.[7] The

[7] This need not be the actual implementation inside the architecture; it only resembles the observation and can be different on other architectures.

performance is then

$$P_{\mathrm{L2}} = \frac{W}{\max(t_{\mathrm{ol}}, t_{\mathrm{L1}} + t_{\mathrm{L2}})} f, \tag{2.9}$$

$$P_{\mathrm{L3}} = \frac{W}{\max(t_{\mathrm{ol}}, t_{\mathrm{L1}} + t_{\mathrm{L2}} + t_{\mathrm{L3}})} f. \tag{2.10}$$

In our case $P_{\mathrm{L2}} = 55.2\,\mathrm{GB/s}$ and $P_{\mathrm{L3}} = 31.5\,\mathrm{GB/s}$.

To determine the memory bandwidth we use McCalpin's STREAM copy benchmark (McCalpin [1995]) which achieves (without nontemporal stores and including the write allocate) a bandwidth of $\approx 26.9\,\mathrm{GB/s}$ when all cores of a cluster are utilized.[8] With the core's clock frequency of 2.3 GHz it takes 5.5 cy to transfer one cache line between L3 cache and memory. Transferring our three cache lines between these two levels takes then

$$t_{\mathrm{mem}} = 16.5\,\mathrm{cy}. \tag{2.11}$$

The performance P_{mem}, when every vector is streamed from memory, is then

$$P_{\mathrm{mem}} = \frac{W}{\max(t_{\mathrm{ol}}, t_{\mathrm{L1}} + t_{\mathrm{L2}} + t_{\mathrm{L3}} + t_{\mathrm{mem}})} f. \tag{2.12}$$

This leads to $P_{\mathrm{mem}} = 14.5\,\mathrm{GB/s}$.

2.6.2 Indirect DAXPY

Indirect DAXPY is similar to the code analyzed in section 2.6.1, with the difference of indirect access `idx` to the vector `r`:

```
for (int i = 0; i < N; ++i)
  r[idx[i]] +=  s * l[i];
```

The vectors `r` and `l` are double-precision floating point vectors with `N` elements each. The `idx` vector contains `4 B` integers. Furthermore `s` is a scalar double precision floating point variable. The code is 4-way unrolled and by the compiler, when optimizations are turned on and target ISA is AVX2 and FMA3. One iteration over this newly formed loop now performs 4 scalar iterations. The code snipped from above effectively becomes

[8] When cluster-on-die (CoD) is enabled, the cores are split into two NUMA domains. The HSW-S processor has 14 cores, that are split into two NUMA domains, each with 7 cores.

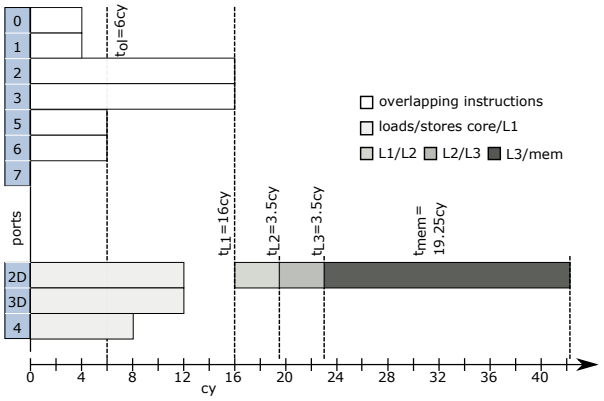


Figure 2.9. Runtime contributions of different execution ports and cache/memory hierarchy levels for eight iterations of the indirect daxpy kernel on HSW-S (Haswell microarchitecture).

```
for (int i = 0; i < N; i += 4)
{
  r[idx[i  ]] += s * l[i  ];
  r[idx[i+1]] += s * l[i+1];
  r[idx[i+2]] += s * l[i+2];
  r[idx[i+3]] += s * l[i+3];
}
```

For the ECM model we determine the duration of the execution of the code inside the core under the assumption all operands reside in the L1 cache.

Eight iterations of the loop (or two iterations of the unrolled loop) require sixteen 8 B loads (`r[:]` and `l[:]`), eight 4 B loads (`idx[:]`), eight 8 B stores (`r[:]`), and 32 address generations. Furthermore eight scalar fused-multiply-adds are used. We define the work performed by eight iterations of the loop (or two iterations of compiler unrolled loop) to be $W = 224\,\mathrm{B}$.

How long the execution takes depends on the underlying architecture. In order to determine this, we take a look at the Intel Haswell microarchitecture in Figure 2.1. The superscalar design has several ports with different execution units for different types of instructions. Instructions scheduled to different ports run independently. The instruction scheduler takes care that no data dependencies are violated. Instructions inside ports are pipelined, but only one instruction per port can be issued per cycle. Haswell can perform two loads (ports 2*D* and 3*D*) and one store (port 4) of sizes up to 32 B at the same time (Fog [2016]). This constellation requires the store address to be simple addressing as only in this case the address generation unit (AGU) on port 7 can be used. For relevant

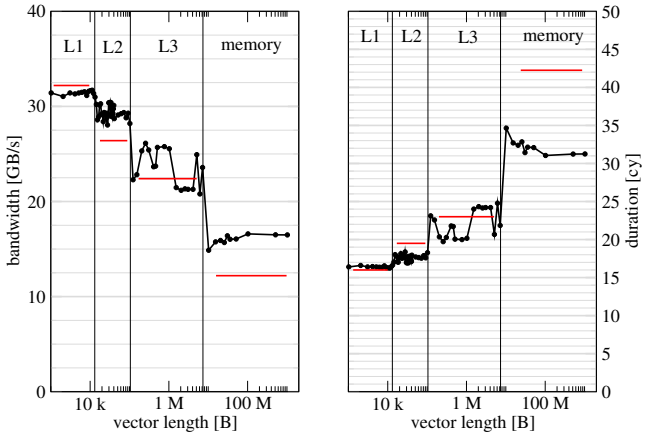


Figure 2.10. Memory bandwidth (left) and duration in CPU cycles of 8 iterations, i.e., two iteration of the compiler generated loop (right), for the indirect daxpy kernel on HSW-S: measurement (black) and ECM prediction (red). As the vectors are getting larger and cannot fit in the higher cache levels, the number of cycles for performing 8 iterations increases, which means the performance decreases.

floating point operations, it hosts two FMA units (ports 0 and 1), two MUL units (ports 0 and 1), and one ADD unit (port 1).

Under these considerations, two iterations of the compiled loop would require 4 cy for the FMA (ports 0 and 1), 12 cy for the twenty-four loads (ports 2 and 3), and 8 cy for the eight stores (port 4). 32 addresses need to be generated. We assume the simple AGU on port 7 is not used, as we observed for DAXPY (section 2.6.1). Generating all 32 addresses on AGUs on ports 2 and 3 requires 16 cy. Instructions regarding index counter increments are for this case negligible, hence we ignore them. We assume that the out-of-order engine can fill bubbles in the pipelines of the different ports as the loop iterations are independent of each other. We classify the ports into arithmetic/logic (ports 0, 1, 5, and 6) and data movement (ports 2, 3, 4, and 7). The maximum duration t_{ol} of the first class of ports is

$$t_{\mathrm{ol}} = \max(4\,\mathrm{cy}, 4\,\mathrm{cy}, 6\,\mathrm{cy}, 6\,\mathrm{cy}) = 6\,\mathrm{cy} \tag{2.13}$$

and the maximum duration of the data movement ports t_{L1} which load/store data from/to L1 is

$$t_{\mathrm{L1}} = \max(16\,\mathrm{cy}, 16\,\mathrm{cy}, 8\,\mathrm{cy}, 0\,\mathrm{cy}) = 16\,\mathrm{cy}. \tag{2.14}$$

The performance P_{L1}, when all data are fetched from L1, is then computed as

$$P_{\mathrm{L1}} = \frac{W}{\max(t_{\mathrm{ol}}, t_{\mathrm{L1}})} f, \tag{2.15}$$

where W denotes the loop specific work performed and f the clock frequency of the core. For this loop with $W = 224\,\mathrm{B}$ and $f = 2.3\,\mathrm{GHz}$ we have $P_{\mathrm{L1}} = 32.2\,\mathrm{GB/s}$, which is in line with the measurements in Figure 2.10.

Modeling the performance when data reside in different cache levels than L1 requires analyzing the data transfers between these levels. In this case this is straightforward as vectors `r`, `l`, and `idx` are streamed from/to memory and no cache reuse takes place. Throughout the cache/memory hierarchy we transfer between each cache level $224\,\mathrm{B} = 3.5\,\mathrm{cl}$ (cache lines) for each iteration: load 64 B of `l`, load 64 B of `r`, load 32 B of `idx`, and store 64 B of `r`.

Transferring a cache line between L1/L2 and L2/L3 takes 1 cy (Intel Corp. [2016]). Hence it takes

$$t_{\mathrm{L2}} = 3.5\,\mathrm{cy} \text{ and } t_{\mathrm{L3}} = 3.5\,\mathrm{cy} \tag{2.16}$$

to transfer our 224 B. For computing the performance P_{L2} and P_{L3}, when data reside in the L2 or L3 cache, respectively, we have to add t_{L2} and t_{L3} to the duration of the data path, as on the considered Intel architectures data transfers seem be be serialized when streaming accesses occur[9]. The performance is then

$$P_{\mathrm{L2}} = \frac{W}{\max(t_{\mathrm{ol}}, t_{\mathrm{L1}} + t_{\mathrm{L2}})} f, \tag{2.17}$$

$$P_{\mathrm{L3}} = \frac{W}{\max(t_{\mathrm{ol}}, t_{\mathrm{L1}} + t_{\mathrm{L2}} + t_{\mathrm{L3}})} f. \tag{2.18}$$

In our case $P_{\mathrm{L2}} = 26.4\,\mathrm{GB/s}$ and $P_{\mathrm{L3}} = 22.4\,\mathrm{GB/s}$.

How many cache lines per cycle can be transferred between L3 and memory could theoretically be obtained by taking the nominal memory bandwidth into account. However, this bandwidth is practically never reached and depends strongly on the access pattern used. This is caused by the organization of the memory subsystem, where, e.g., banking conflicts and DRAM page misses reduce performance. With detailed knowledge about the internals of the memory controller (scheduling strategies, thresholds for strategy switching, ...) and DRAM modules this could also be modeled, which is far from being trivial (Jacob et al.

[9] This need not to be the actual implementation inside the architecture, it only resembles the observation and can be different on other architectures.

[2007]). As this is beyond the scope of the ECM model, typically a microbenchmark resembling the used access pattern by the code under investigation is used to measure the attainable bandwidth and use it as input for the model. For this model we use McCalpin's STREAM copy benchmark (McCalpin [1995]) which achieves (without nontemporal stores and including the write allocate) a bandwidth of $\approx 26.9\,\mathrm{GB/s}$ when all cores of a cluster are utilized. With the core's clock frequency of 2.3 GHz it takes 5.5 cy to transfer one cache line between L3 cache and memory. Transferring our 224 B between these to levels takes

$$t_{\mathrm{mem}} = 19.25\,\mathrm{cy}. \tag{2.19}$$

The performance P_{mem}, when every vector is streamed from memory, is then

$$P_{\mathrm{mem}} = \frac{W}{\max(t_{\mathrm{ol}}, t_{\mathrm{L1}} + t_{\mathrm{L2}} + t_{\mathrm{L3}} + t_{\mathrm{mem}})} f. \tag{2.20}$$

This leads to $P_{\mathrm{mem}} = 12.2\,\mathrm{GB/s}$. Which is about 23 % less than the achieved performance on HSW-S architecture. While the ECM prediction of the DAXPY kernel is in line with measurements, in the case of an indirect DAXPY kernel the ECM model underestimates the measured performance. The ECM model was developed and studied mostly for vectorized codes. As the indirect DAXPY kernel is not vectorized, it would require further investigation.

Chapter 3

Node-level performance modeling of sparse triangular solve

Even in scientific computing, simulation code development often lacks a basic understanding of performance bottlenecks and relevant optimization opportunities. In this chapter we will use a structured model-based performance engineering approach on the compute node level for a fundamental kernel operation in sparse factorization solvers. We aim at a deep understanding of how code performance comes about and which hardware bottlenecks might apply. The pivotal ingredient of this process is a performance model which links software requirements with hardware capabilities. Such models are often simplified such as the well-known Berkeley roofline model or the Erlangen ECM model, but it leads to deeper insights and strikingly more accurate runtime predictions.

The package PARDISO[1] is a thread-safe, high-performance, and memory-efficient, serial and parallel solver for the direct solution of unsymmetric and symmetric sparse linear systems on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques (Anderson et al. [1999]). In order to improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining parallelism is exploited with a combination of left- and right-looking supernode techniques.

PARDISO calculates the solution of a set of sparse linear equations with multiple right-hand sides, using a parallel LU, LDL^T, or LL^T factorization. PARDISO supports a wide range of sparse matrix types and computes the solution of real or complex, symmetric, structurally symmetric or unsymmetric, positive definite,

[1] A discussion of the algorithms used in PARDISO, the user manual, and more information on the solver can be found at `http://www.pardiso-project.org`

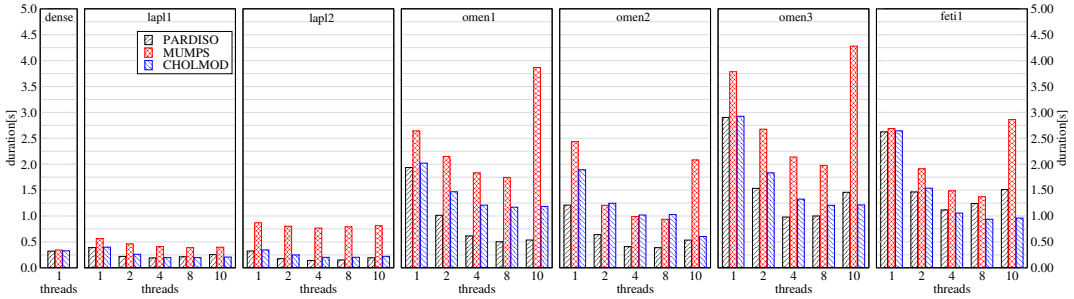


Figure 3.1. Duration of forward/backward substitution for the dense, lapl1, and lapl2 matrices for different sparse direct solvers on the IVB system.

indefinite or Hermitian sparse linear system of equations on shared-memory multiprocessing architectures.

The parallel pivoting methods for unsymmetric matrices allow complete supernode pivoting in order to ensure numerical stability and scalability during the factorization process. For sufficiently large problem sizes numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture and a speedup of up to seven using eight processors has been observed. The package is implemented using multithreading using OpenMP directives. PARDISO performs the analysis steps depending on the structure of the input matrix A. See Bollhöfer et al. [2020] for further details.

For comparison and to motivate the usage of the PARDISO solver, we run initial benchmarks on various matrices. Here we used CHOLMOD (Chen et al. [2008]; Davis [2006]), MUMPS (Amestoy et al. [2000a, 2001, 2006]), and PARDISO (Schenk and Gärtner [2004]; Kuzmin et al. [2013]) solvers. For CHOLMOD and MUMPS we set METIS reordering, so all solvers have about the same factors. We measured duration of the forward/backward substitution for our benchmark matrices. The results for the Ivy Bridge system are shown in Figure 3.1.

In this chapter, we will mainly analyze the data transfer between the different cache levels and the FLOPs performed during the sparse triangular solve phase. The results of this analysis are used as input for our performance model. Therefore, we mostly inspect the different instantiations of the loops resulting from Algorithms 1 and 2 on page 22 and 23. We only consider the innermost loops because all the arithmetics are performed here. Our assumption is that the instructions regarding control variables of the loops are negligible, hence we do not take them into account.

3.1 Algorithm and data structures of sparse triangular solve

Let A be an $N \times N$ matrix and x and b be vectors of size N. The linear system $Ax = b$ is solved via LDL^T decomposition by factorizing A into a lower diagonal matrix L and a diagonal matrix D such that $A = LDL^T$. The system is then solved in three steps. First, $Ly = b$ is solved via forward substitution. This is followed by a diagonal solve $Dz = y$ and, afterwards, the resulting z vector is used to solve for the solution vector $L^T x = z$ via backward substitution. In PARDISO, the forward substitution is performed columnwise, starting with the first column. The data dependencies here allow us to store vectors y, z, b, and x in only one vector r.

The sparse matrix is stored in a PARDISO specific format shown in Figure 1.11. Adjacent columns exhibiting the same row sparsity structure form a *panel*, also known as a *supernode*. A panel's column count is called the *panel size s*. The columns of a panel are stored consecutively in memory excluding the zero entries. Note that columns of panels are padded in the front with zeros so they get the same length as the first column inside their panel. The padding is of utmost importance for the PARDISO solver to use Level-2/3 BLAS and LAPACK functionalities. Please see Bollhöfer et al. [2020] for more details. Furthermore, panels are stored consecutively in the array `l`. Row and column information is now stored in accompanying arrays. Column indices are used as indices into the array `xl` to look up the start of the column in the array `l` which contains the numerical values of the factor L. To determine the row index of a column's element array `id` is used, which holds the row indices for each panel.

For parallel execution concurrent updates to the same entry of `r` must be avoided. The *parts* structure contains the start (and end) indices of the panels which can be updated independently as they do not touch the same entries of r. Two parts, colored blue and green, are shown in Figure 1.11. The last part in the bottom right corner of L is special and is called the *separator* and is colored gray. Parts which would touch entries of `r` in the range of the separator perform their updates into separate temporary arrays `t`. Before the separator is then serially updated, the results of the temporary arrays are gathered back into `r`. The backward substitution works the same, just reversed, and updates to different temporary arrays are not required. Figure 3.2 shows a possible execution of forward solve. The nested dissection reordering assigns to every partition about the same number of columns. However, panels in different partitions can have different size (number of columns). Then different threads execute differently unrolled code, as can be seen on the left side of Figure 3.2. For easier modeling

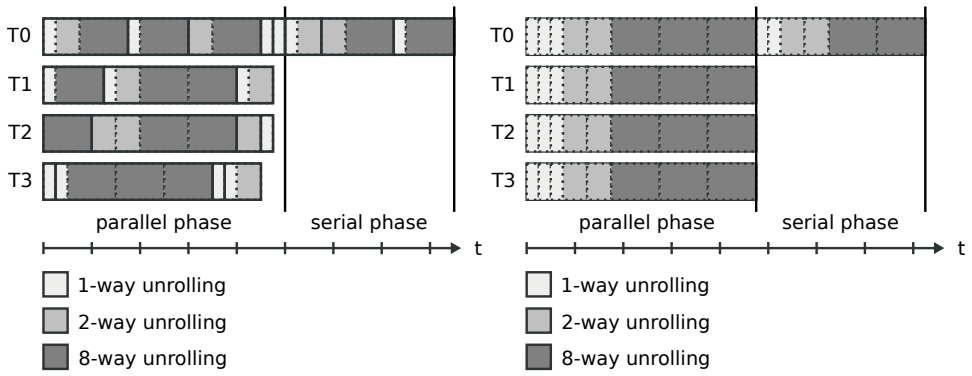


Figure 3.2. Possible real execution of forward solve. In the parallel phase cores can execute differently unrolled loops at the same time. For easier modeling the execution of the unrollings is sorted and evenly distributed over all cores.

we assume the execution of the unrollings is sorted and evenly distributed over cores, as shows the right side of Figure 3.2.

The complete forward substitution is listed in Algorithm 1. If no parallel execution is required then panels are updated successively in serial, and during forward substitution, updates to temporary arrays are not necessary. Please note that through the dense storage of panels, indirect accesses to `r` are required, resulting in an "indexed DAXPY"-like operation, which prohibits a straightforward vectorization. For performance reasons (discussed in Section 3.2.2) the loops over the columns (blue text) in Algorithm 1 are 1-, 2-, and 8-way unrolled. The algorithm for the backward substitution in Algorithm 2 looks nearly the same, except that the serial part is executed first, which is then followed by the parallel section.

Parallel handling of the separator during the forward and backward substitution is in principle possible. Hereby the loops over the rows in line 21 of Algorithm 1 and line 5 of Algorithm 2 would be parallelized. However, typically the number of rows for sparse problems is too small to benefit from this optimization and at worst it could introduce significant overhead.

3.2 Performance analysis of sparse triangular solve

All entries of the matrix L are stored as double-precision floating point numbers in the vector `l`, consuming 8 B (byte) each. Elements of the vector `xl` (column start indices in the vector `l`) and vector `xid` (start indices of row indices for each panel) are stored as 8 B integers, whereas for the entries of the vector `id` (row indices for each panel) 4 B integers are used.

3.2.1 Data transfers and FLOPs without unrolling

In the most simple case no unrolling is applied and the innermost loop of forward substitution from Algorithm 1 looks like

```
21: for k = xl[j] + offset; k < xl[j+1]; ++k do
22:     row = id[i++]
23:     r[row] -= r[j] l[k]
24: end for
```

As the loops from lines 6–9 and 10–13 are in principle identical to the loops in lines 21–24, we only discuss the latter. During each iteration one nonzero is processed, two FLOPs are performed, namely, a multiplication and an addition, and the following elements get loaded and stored: loaded: `id` (4 B), `r` (8 B), `l` (8 B); stored: `r` (8 B).

How much data are transferred inside the cache hierarchy depends on the size of the caches, their replacement strategies, the size of `r`, the average panel size, as well as the structure of the panels, i.e., which part of `r` is accessed. Here we assume `r` is small enough to be kept at least in last level cache (LLC) and temporal locality ensures it is not evicted. Row indices in `id` for a panel are loaded from memory for the panel's first column and then are reused during each iteration over the panel's remaining columns from the LLC in the worst case. With panel size $s = 1$, for each element of `l` one row index is transferred and no reuse is possible. In general reuse is only possible, starting with a panel's second column for panel sizes $s \geq 2$. Coefficients of `l` are always streamed in from memory, as they are used only once and the array `l` is typically too large to be kept in LLC. Figure 3.3 visualizes the transfers assuming three cache levels and `r` is cached in LLC. While iterating over panels and columns, the number of column elements decreases as L is a lower triangular matrix. Thereby the number of used elements from `r` also decreases, which we call the active part of `r`. At some point the active part can be completely kept in the L2 or even the L1 cache. This also holds true for `id`, except when a new panel starts, then the panel's row indices must first be loaded from memory. With Intel L2 cache 256 kB and L1 cache 32 kB, a maximum of about 13000 and 1600 elements of `r`, `id`, and `l` could fit into the L2 and L1 caches, respectively. However, due to the associativity and replacement strategy of the cache and the need to store other data, it is reasonable to expect only half of the size can be used. We can expect if the active part of `r` is less than about 6000 elements, it can be kept in the L2 cache, and if it is less than about 800 elements it fits into the L1 cache.

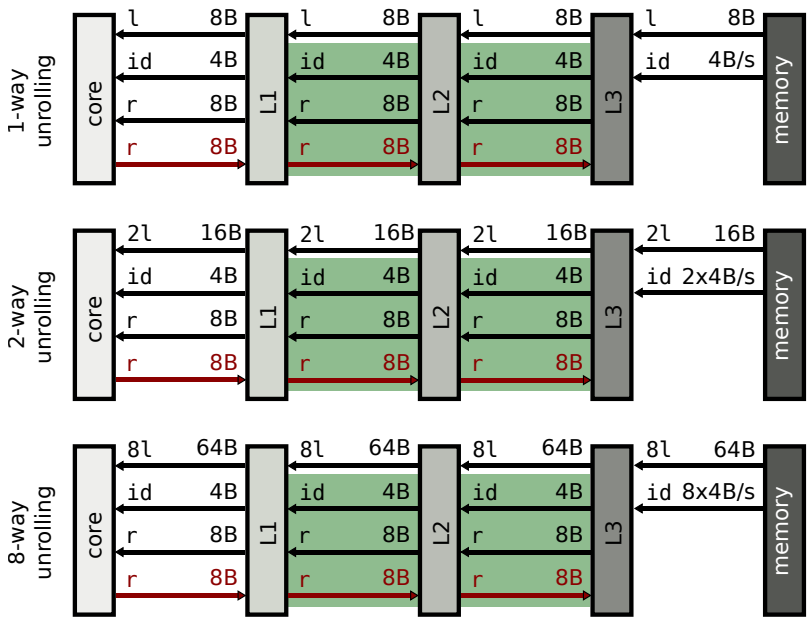


Figure 3.3. Data transfers for one iteration of the forward substitution when 1-, 2-, or 8-way column unrolling is applied for a panel with panel size s. Thereby 1, 2, or 8 nonzero elements of `l` are processed, respectively.

The innermost loop of the backward substitution from Algorithm 2 is

```
5: for k = xl[j] + offset; k < xl[j+1]; ++k do
6:     row = id[i++]
7:     r[j] = r[j] - r[row] l[k]
8: end for
```

As this loop from lines 5–8 is the same as the one from lines 16–19, all following statements hold true for both. As with forward substitution one nonzero is processed, two FLOPs are performed, but only loads occur: `id` (4 B), `r` (8 B), and `l` (8 B). Note that j is unchanged in the innermost loop, hence $r[j]$ always refers to the same element and is not considered for the data transfer analysis. Figure 3.4 displays the data transfers occurring for one nonzero update if `r` is cached in L3 cache.

3.2.2 Data transfers and FLOPs with unrolling

As noted in Section 3.1 it is beneficial to handle several columns at once. For this purpose PARDISO has additionally 2- and 8-way manually unrolled loops over columns. These are used when a panel contains more than one column. With an

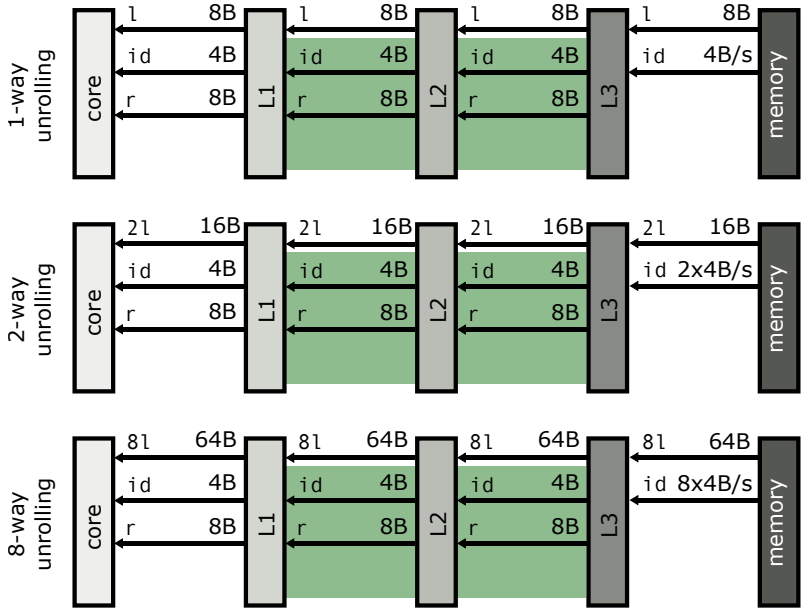


Figure 3.4. Data transfers for one iteration of the backward substitution when 1-, 2-, or 8-way column unrolling is applied for a panel with panel size s. Thereby 1, 2, or 8 nonzero elements of `l` are processed, respectively.

unrolling factor of two the innermost loop for forward substitution becomes

```
1: nj = nonzero column length
2: for k = xl[j] + offset; k < xl[j+1]; k += 2 do
3:     row = id[i++]
4:     r[row] = r[row] - r[j] l[k] - r[j+1] l[k+nj]
5: end for
```

Instead of processing only one nonzero the 2-way unrolling handles two nonzeros per iteration. Hence, four FLOPs are performed and two entries of `l` are loaded. Hereby the corresponding element of `r` only needs to be loaded once instead of twice, when processing two elements of `l`. All other transfers stay unchanged. In general with a u-way unrolling during each iteration, $2 \times u$ FLOPs are executed and $u \times 8\,\mathrm{B} + 20\,\mathrm{B}$ are transferred.

Unrolling of the backward substitution loop results in the following code for a 2-way unrolling:

```
1: nj = nonzero column length
2: for k = xl[j] + offset; k < xl[j+1]; k += 2 do
3:     row = id[i++]
4:     r[j] = r[j] - l[k] r[row]
```

```
5:    r[j+1] = r[j+1] - l[k+nj] r[row]
6: end for
```

Also here four FLOPs per iteration are performed, two entries of `l` are loaded and the corresponding element of `r` is loaded only once. For a u-way unrolling, $2 \times u$ FLOPs and $u \times 12\,\mathrm{B} + 8\,\mathrm{B}$ are loaded.

As already noted, we assume `r` and a panel's current row indices from `id` are at least cached in LLC or higher cache levels. Unrolling, hereby, only saves transfers inside the cache hierarchy. The bytes transferred between memory and LLC are left unaffected and depend only on the panel size. With larger panel sizes in total, fewer row indices `id` are needed for the whole matrix.

3.3 Application of modified Berkeley roofline model on sparse triangular solve

For our performance predictions of the sparse triangular solve, we apply the roofline model (Williams et al. [2009]). As mentioned before, the model takes into account the attainable memory bandwidth as well as the peak floating point performance of the processor and relates these hardware capabilities to the requirements of the code. It can be written as

$$P = \min(P_{\max}, B/B_c), \tag{3.1}$$

where $P_{\max}$ denotes the attainable floating point performance, B the attainable memory bandwidth, and B_c the code balance.

Here, $P_{\max}$ depends already on the floating point characteristics of the code and the processor and does not represent the peak floating point performance as it can be obtained from a processor's data sheet. If a processor supports vectorized FMA instructions, but only vectorized add or multiply instructions are used, then $P_{\max}$ is halved. Using scalar instructions instead of the AVX vectorized counterparts reduces $P_{\max}$ further by a factor of four. And, finally, if the floating point instruction mix does not equally utilize a processors floating point units, $P_{\max}$ is again reduced. For example, the Ivy Bridge (IVB) system (Section 3.5, Table 2.1) hosts an add and multiply unit, but if only one type of floating point instruction is used, only half of the theoretical floating point performance can be attained. In our case the compiler uses scalar FMA instructions for the core loops of the sparse triangular solve for architectures with AVX2 support and scalar add/multiply instructions for architectures without FMA support like IVB.

Substitution		Forward			Backward		
u		1	2	8	1	2	8
B_c	[B/F]	4−6	4−5	4−4.25	4−6	4−5	4−4.25

Table 3.1. Code balance B_c of forward/backward substitution for unrolling factors u when the factor L must be fetched from memory. Values depend on actual panel size s and possible cache reuse.

The attainable memory bandwidth B is measured with a microbenchmark, ideally resembling the application's memory access pattern. For the sparse triangular solve, we use a read-only benchmark from the LIKWID tool suite (Treibig et al. [2010]). As the sparse triangular solve in PARDISO only uses scalar load instructions we also use them for the microbenchmark.

The *code balance* B_c is the ratio of bytes transferred to the number of FLOPs performed in the code. In the best case, when the panel size s is large and the indices are cached, during the forward (1.8) and backward substitutions (1.9), each nonzero of L must be loaded once and the loading of indices can be neglected, respectively. Furthermore, the computation involves two FLOPs per nonzero. As nonzeros are stored in double precision consuming 8 B, this results in a best case code balance of $B_c = 8/2\text{B/F} = 4\text{B/F}$, where F denotes FLOP. Table 3.1 lists the code balance for different unrolling factors when the factor L must be fetched from memory and uses the data transfer and FLOP counts from Section 3.2. In contrast, the *machine balance* B_m defines the ratio for the whole system and uses the ratio of the attainable memory bandwidth B to the maximum floating point performance $P_{\max}$. This bandwidth is found in Table 2.1 for all systems. If $B_c > B_m$ then the roofline model indicates that the code's performance is limited by the memory bandwidth, i.e., that the code is memory bound. This is the case for the sparse triangular solve phase of all unrolling factors, when the factor L is too large to be kept in cache and completely located in memory.

To determine performance limits, we only consider the case when data reside in memory. Therefore the data transfers between the L3 cache and memory are relevant as shown in Figures 3.3 and 3.4. Only nonzero entries of L with the corresponding panel indices are loaded. Their amount depends only on the structure of L and is independent of the used loop unrollings. The roofline model for the memory bound case as a function of the number of threads t can be formulated as

$$P^A(t) = \frac{\text{nnz}(L) \times 2\frac{\text{FLOP}}{\text{nz}}}{\frac{D_A(t)}{B(t)}} \quad \frac{\text{FLOP}}{s}, \tag{3.2}$$

where nnz(L) denotes the number of nonzeros of the factor L resulting from a factorization of A for t threads, $B(t)$ is the attainable memory bandwidth of the system utilizing t threads, and $D_A(t)$ the data volume of nonzeros and indices making up L. The only adjustment to the original model here is its dependency on the number of threads. Choosing t equal to the number of total cores yields the original roofline model.

To distinguish between the parallel phase, where the parts are handled, and the serial part, where the separator is treated, we modify (3.2). We use the following formula for the *modified roofline model*:

$$P^A_{\text{mod}}(t) = \frac{\text{nnz}(L) \times 2\frac{\text{FLOP}}{\text{nz}}}{\frac{D^p_A(t)}{B(t)} + \frac{D^s_A(t)}{B(1)}} \quad \frac{\text{FLOP}}{s}, \tag{3.3}$$

where $D^p_A(t)$ represents the data volume of nonzeros and indices built up by the parallel parts, and $D^s_A(t)$ the data volume built up by the nonzeros and indices of the separator. Please note that both data volumes $D^p_A(t)$ and $D^s_A(t)$ depend on A and the number of threads t. The values for $D^p_A(t)$ and $D^s_A(t)$ are extracted from the factorized matrices a priori to solve.

Please note that the accuracy of the roofline model predictions, especially for single cores, as included in the modified model, can be inaccurate. If the in-core execution time (excluding floating point operations) or the in-cache traffic dominates the execution time, predictions become unreliable as this is not covered by the roofline model. In Chapter 4, we use dedicated single core measurements to validate that, in the case of the sparse triangular solve, this approach is valid. However, this effect can be modeled in detail with the ECM model (Treibig and Hager [2010]; Hager et al. [2016]; Stengel et al. [2015]) and was already studied for stencil kernels.

3.4 Application of Erlangen ECM model on sparse triangular solve

As shown in Chapter 3.2, there are many factors influencing the performance: for example, unrolling, active part of `r`, or presence of `id` in the cache. Detailed performance evaluation would require a model for every combination of these parameters and combining these models based on detailed analysis of a matrix A and its factor L. For real world matrices, this is not possible. To make the modeling feasible, we focus only on two extreme cases: first, we consider the code without unrolling that reads `id` from the memory and second, 8-way unrolled

code that already has `id` in cache. In both cases we are assuming the active part of `r` is large and the data reside in the L3 cache. The first scenario achieves the worst possible performance, while the second one can perform close to the theoretical maximum. These two cases give us a range of expected performance, such that for any matrix the sparse triangular solve algorithm should be within these bounds.

For the ECM model of the worst case we consider eight iterations of the inner most loop (8 rows) in order to fill one cache line. Eight iterations of both forward and backward substitution without unrolling require sixteen 8 B loads (`r[:]` and `l[:]`) and eight 4 B loads (`idx[:]`). On top of that, the forward substitution also requires eight 8 B stores (`r[:]`). These loads and stores need 32 addresses (forward substitution) or 24 addresses (backward substitution) to be generated, which takes 16 or 12 cycles, respectively. We assume that all the addresses are generated on ports 2 and 3. The simple AGU on port 7 is not used as we showed in Sections 2.6.1 and 2.6.2. The floating point arithmetics for eight iterations consists of 8 multiplications and 8 additions. This can be done in 8 cycles using the units on ports 0 and 1. Unlike in the DAXPY case, only 96 B (or 1.5 cl) are read from memory. This leads to expected performance of 1.18 GF/s (forward substitution) or 1.46 GF/s (backward substitution).

For the ECM model of the best case with 8-way unrolling we take 8 columns and in every column 8 rows (8 iterations of the innermost loop). Eight iterations of forward substitution with 8-way unrolling require 672 B (10.5 cl) to be loaded from cache and 512 B (8 cl) read from memory. Backward substitution require 608 B (9.5 cl) to be loaded from cache and 512 B (8 cl) read from memory. These loads and stores need 88 addresses (forward substitution) or 80 addresses (backward substitution) to be generated, which takes 44 or 40 cycles, respectively. The floating point arithmetics consist of 64 multiplications and 64 additions. This can be done in 64 cycles. In the best case we assume `id` is kept in the cache, thus only 512 B (8 cl) are read from memory. This leads to expected performance 2.7 GF/s (forward substitution) or 2.86 GF/s (backward substitution).

3.5 Experimental testbed for the performance evaluation

The specifications of the systems used for the performance analysis are described in Table 2.1. The machines with Intel processors are based on the Ivy Bridge (IVB), Haswell (HSW-D and HSW-S), Broadwell (BDW), Skylake (SKX), and

Knights Landing (KNL) microarchitectures. The first four microarchitectures are successors to each other and can be seen as traditional superscalar, multicore, SIMD capable processors. HSW-D and HSW-S are desktop and server systems, respectively. The HSW-S systems has cluster-on-die (CoD) enabled. Here, the processor's local L3 cache is divided into two parts and the memory forms two NUMA locality domains. SKX is the server variant of the Skylake microarchitecture including support for AVX-512 and hosts an additional FMA unit. The Knights Landing processor is a representative of Intels Xeon Phi line, a manycore architecture with SIMD lanes wider than in the formerly named processors. It is the successor of the Knights Corner manycore processor.

The exact AVX-512 ISA (instruction set architecture) for Knights Lading differs from the Skylake incarnation, but for our purpose is not relevant. Knights Lading includes a 16 GB large high bandwidth memory (HBM) with bandwidths up to 450 GB/s; see also the discussion in Chapter 4. We operate KNL in the flat memory model, where the DDR memory and HBM represent a NUMA domain, each. AMD-based systems include a desktop (ZEN-D) and server (ZEN-S) system based on the Zen microarchitecture. ZEN-S' processor with 24 cores consists of four NUMA LDs.

The CPU frequencies on all machines were fixed to the base frequencies specified in Table 2.1. On the ZEN systems we set the frequency to the nominal base frequency, but could not disable AMD's turbo mode, which allows cores to run above this frequency. For Knights Landing, altering frequencies are not supported and are handled by the processor itself. Furthermore, each thread's affinity was explicitly set. For all arrays, large 2 MiB pages were used. First-touch policy was in place, and we verified via the NUMA-API that the data always reside in the cores associated NUMA domain. On all systems supporting simultaneous multithreading (SMT) only physical cores were used. As compiler, Intel C/Fortran Compiler version 17.0.1 was used.

3.5.1 Read-only memory bandwidth and machine balance

For the performance model an effective memory bandwidth is required. For the evaluation of the effective bandwidth we measure read-only bandwidth. The measurements are reported in Table 2.1. As discussed in Section 3.3 only scalar loads are used. If enough cores are used then both scalar and vectorized read-only benchmarks saturate the memory bandwidth with the only difference being that the saturation of the latter is already achieved with fewer cores. This is exemplified on HSW-D and HSW-S systems shown in Figure 3.5a. HSW-D and HSW-S show the typical saturation behavior for (current Intel) desktop and

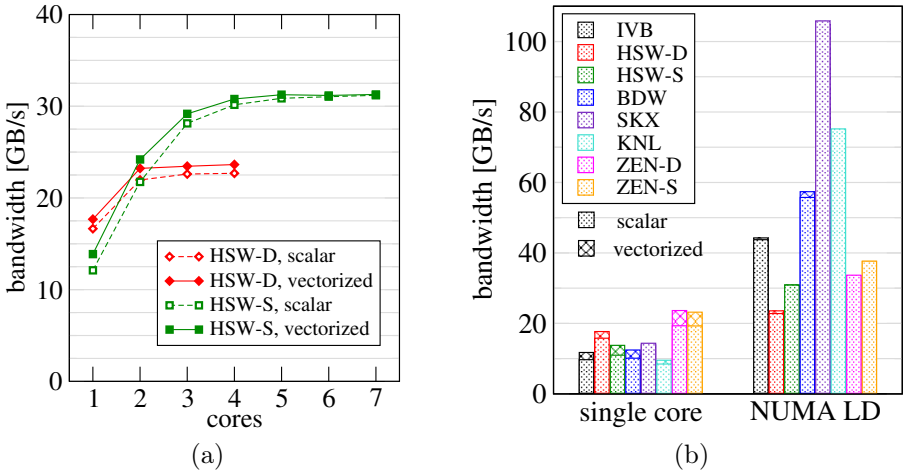


Figure 3.5. (a) Bandwidth over the number of cores of the read only benchmark in a scalar and vectorized version exemplified on HSW-D and HSW-S. (b) Single core bandwidth and saturated bandwidth with all available cores of the processor/cluster.

server systems. Typically desktop systems nearly saturate the memory bandwidth with one core. Figure 3.5b shows the difference between the scalar and vectorized read-only benchmark for the single core and the usage of all cores inside a NUMA LD over all systems in the test bed. IVB, HSW-S, and the ZEN-based systems reach, with one core and vector loads between 15 % and 25 %, a higher bandwidth than with scalar load instructions. However, utilizing the full NUMA LD, nearly no difference is visible.

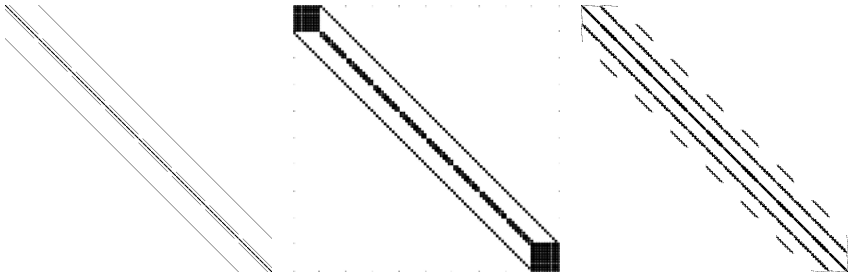
The machine balance B_m from Table 2.1 considers the scalar read-only memory bandwidths and the scalar double precision floating point capabilities of the processors. This is either a scalar FMA or, if unavailable, a scalar addition and multiplication for processing a nonzero.

3.5.2 Matrices for performance modeling

Table 3.2 lists matrix dimension (n), number of nonzeros in the matrix (nnz(A)), and the factor (nnz(L)) of the matrices used for benchmarking in the following sections. The reported numbers of nonzeros are reported for factorizations using single threaded execution and a panel size $s = 80$. All matrices are sparse except for the first matrix *dense*, where both the matrix and the factor L are dense. We use this dense matrix as a best case example for our single core performance investigations. The matrices *lapl1* and *lapl2* are test matrices arising from a finite

Matrix	n	nnz(A)	nnz(L)
dense	20×10^3	200×10^6	200×10^6
lapl1	256×10^3	3×10^6	219×10^6
lapl2	343×10^3	1×10^6	166×10^6
omen1	$1\,751 \times 10^3$	32×10^6	$1\,076 \times 10^6$
omen2	760×10^3	20×10^6	690×10^6
omen3	$1\,271 \times 10^3$	42×10^6	$1\,651 \times 10^6$
bddc	750×10^3	31×10^6	$1\,590 \times 10^6$

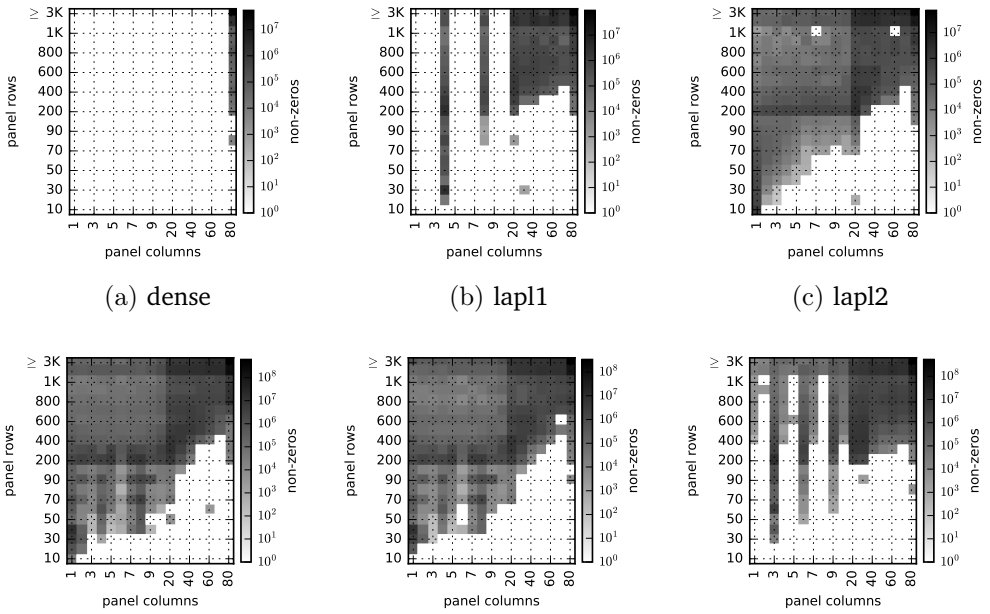
Table 3.2. Dimension (n) and number of nonzeros (nnz) for A and L for all benchmark matrices.



(a) lapl (b) omen (c) bddc

Figure 3.6. Structure of A for matrix classes lapl (a), omen (b), and bddc (c).

difference discretization of the Laplace operator in three dimensions with Dirichlet boundary conditions. In addition, the matrix *lapl2* contains a block structure of size 4. The *omen* matrices correspond to a set of representative matrices from an atomistic nanoelectronic device engineering simulation code (Luisier et al. [2011]). The matrix *bddc* arises from a finite element discretization of a typical solid mechanics problem. Here, as a material model, a J2-elasto-plasticity model was chosen and three-dimensional and piecewise quadratic tetrahedral finite elements were used for the discretization. The matrix *bddc* represents a typical subdomain problem arising in the BDDC (balancing domain decomposition by constraints) implicit finite element solver. Figure 3.6 shows the structure of A for different matrix classes, whereas more interesting, is the nonzero distribution over the panel sizes of the factor L found in Figure 3.7. Please note that the current factorization limits the number of parts to powers of two. To avoid load imbalance during the solve step we report results only for thread counts which are powers of two.



(a) dense (b) lapl1 (c) lapl2

(d) omen1 (e) omen2 (f) feti1

Figure 3.7. Multiparameter histograms showing how many nonzeros of L are in panels with a certain number of columns and rows. Panels with 3000 and more rows are accumulated. This plot gives an impression of how the work, i.e., nonzeros, in L is distributed over panel dimensions.

Chapter 4

Performance evaluation and analysis

4.1 Single core performance modeling, evaluation, and analysis

For the code generation of the sparse triangular solve discussed in Chapter 3, we instruct the compiler via directives to unroll the innermost loops of Algorithms 1 and 2, i. e., to perform an unrolling over the rows. Typically an unrolling factor of eight or sixteen showed the best performance for all architectures. We prohibit vectorization of these loops, which causes the compiler to use scalar floating point addition/multiplication or FMA instructions. We observe that with vectorization enabled for these loops the compiler performs manual gather and scatter, which results in the same or even poorer performance compared to the scalar unrolled versions. This is required as AVX2 only includes a vector gather instruction. Only for the 1-way column unrolling of the sparse triangular solve with KNL is the compiler generated loop utilizing vector scatter/gather instructions superior to the scalar version, which is why we use this version for the combination of KNL and 1-way column unrolling. However, SKX also supports vector scatter/gather instructions, but did not show any performance improvement.

For the single core measurements we use the matrix *dense*. We create three variants of it, where the maximum panel size s is limited to 1, 2, and 80 columns. This enables isolated measurement of the 1-, 2-, and 8-way unrolled loops. Furthermore, the matrix *dense* exhibits the most homogeneous access pattern as memory is only accessed in long contiguous streams and relates best to our read-only benchmark (Section 3.5.1) used as input for the modified roofline model (Section 3.3, (3.3)). Figure 4.1 shows the measured performance for all systems in our testbed for sparse triangular solve. The blue bars indicate the perfor-

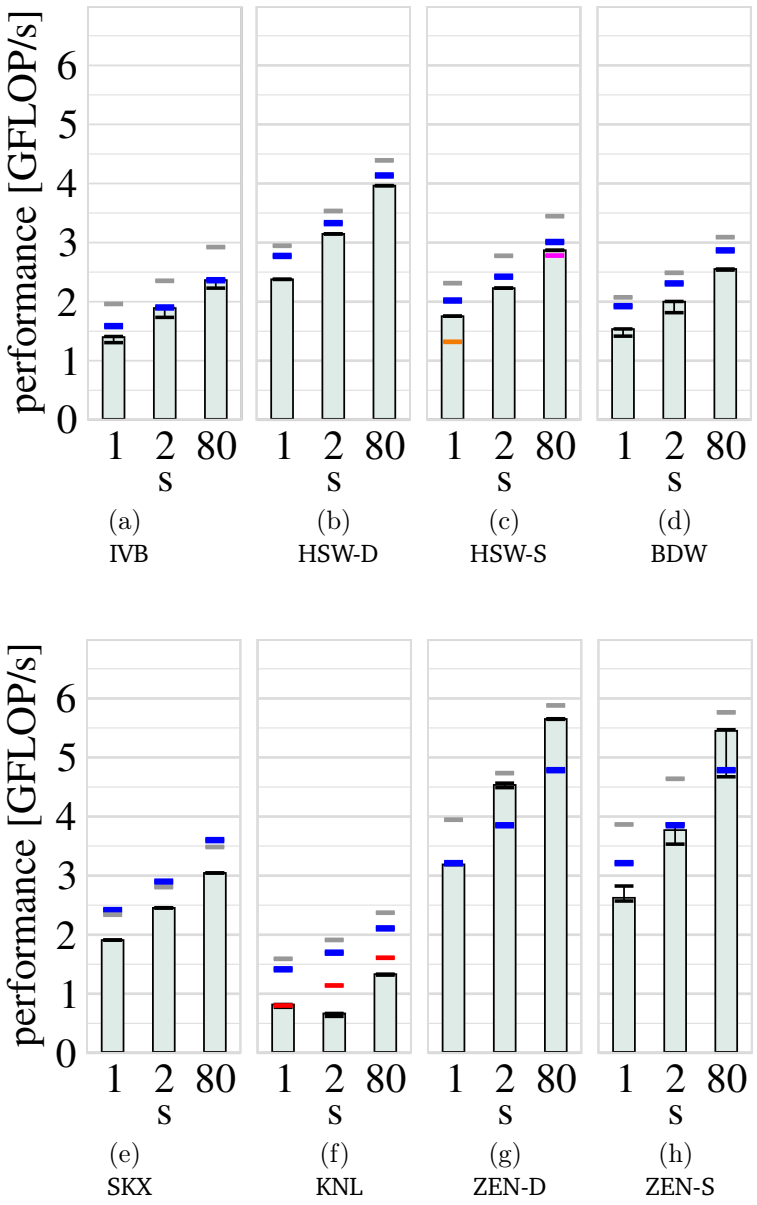


Figure 4.1. Performance of PARDISO's sparse triangular solve for the matrix dense. Blue and gray horizontal bars show predictions of the modified roofline model with scalar and vectorized read bandwidth, respectively. Orange and magenta horizontal bars show the worst and the best performance predictions of the ECM model for the HSW-S architecture.

mance limit from the modified roofline model. In general larger panels deliver a higher performance. This is expected as the code balance decreases with increasing panel sizes. For panel sizes $s = 1$ and $s = 2$ with matrix *dense*, the worst case code balance of $B_c = 6\,\mathrm{B/F}$ and $B_c = 5\,\mathrm{B/F}$ is reached, respectively. With a panel size $s = 80$ the code balance for the 8-way unrolled loop becomes nearly $B_c = 4\,\mathrm{B/F}$. The orange and magenta bars indicate the worst and the best performance limits from the ECM model. With a panel size $s = 1$ we expect the worst performance (close to the orange bar) as the unrolling is not possible. With a panel size $s = 80$ we expect the best performance (close to the magenta bar) because the most efficient 8-way unrolled code is used and the array `id` is many times reused in cache.

The modified roofline model predictions for the Intel based systems (except for KNL) deviate up to 25 % from the measurements, but achieve a higher performance with larger panels. For KNL the model error is in the range of 55 % to 160 % over all panel sizes which suggests some error in the model assumptions. On KNL a STREAM-like (scalar) copy and read benchmark achieves an L2 bandwidth of around 7.8 GB/s and 8.5 GB/s with one core, respectively. From Figures 3.3 and 3.4, we see that between the L1 and L2 caches (depending on the matrix) we might have a higher code balance compared to the one we assume between memory and L2. As the measured L2 bandwidth is nearly equal to the memory read-only bandwidth the former data path becomes the new bottleneck. However, determining a priori the code balance between the L1 and L2 caches for a matrix is analytically nontrivial which is why we derive the metric from the measured data traffic between the caches. The adjusted model based on these new inputs is shown in Figure 4.1f as red bars and reduces the model error significantly. The poor performance with panel size $s = 2$, however, is unclear and deserves further investigation. The roofline model for the AMD-based ZEN systems underestimates the performance by up to 20 %. It seems that sparse solve in the cases of ZEN-D and $s = 2, 80$ and ZEN-S and $s = 80$ achieves the bandwidth obtained with the vectorized read benchmark as the bars in Figures 4.1g and 4.1h match the gray lines, which represent the modified roofline model based on the vectorized read-only bandwidth. Measuring the actual clock frequency shows that ZEN-D and ZEN-S run with 3.8 GHz and 3.2 GHz during sparse solve, respectively. However, they also achieve this frequency during read-only bandwidth measurements.

4.2 Multiple core performance modeling, evaluation, and analysis

Performance and modeling results on multicore architectures for all matrices are shown in Figures 4.2 and 4.3. For HSW-D (second column) the main memory was not large enough for benchmarks with the *bddc*, *omen2*, and *omen3* matrices, which is indicated as "out of memory" in the figures. In this work we ensure correct NUMA placement for one locality domain; this is why we limit the scope of the analysis to one NUMA LD only. Hence, for HSW-S and ZEN-S fewer cores than the processor houses are used. Please keep in mind that for HSW-S, seven of 14 cores and for ZEN-S only six of 24 cores are used and with correct NUMA placement and usage of more cores a higher performance could be achieved.

As already pointed out, the number of independent parallel parts produced by the factorization is always a power of two. For nonpower of two thread counts this results in load imbalance and decreases performance, which is why we omit them in the graphs. Furthermore, the current implementation of the factorization increases the size of the separator in the factor L with increasing number of threads as already observed in scaling studies for sparse solve in Klawonn et al. [2015]. Figure 4.4 shows the fraction of nonzeros in L which are part of the separator, i.e., the part which must be executed serially. Except for *omen1* this prohibits the efficient usage of larger core counts and the test matrices reach their highest performance already with four or eight cores. This is also the reason why we cannot efficiently utilize KNL's high bandwidth memory. Despite that its bandwidth scales (nearly) linearly with the number of cores, its single core bandwidth is not significantly different from the one delivered from main memory. Only with higher core counts can the full HBM bandwidth be obtained. However, with 16, 32, and 64 threads the serial fraction of the resulting factor from the matrices is already too high and performance nearly drops to the single core performance level. Hence, we excluded HBM measurements from the plots as it gives no further insight. With SKX hosting 20 cores in one NUMA LD the impact of the separator becomes visible (fifth column of Figures 4.2 and 4.3). The separator of *lapl1*, *lapl2*, and *bddc* strongly increases over the number of cores and limits the scaling of the performance early. The maximum performance here is already reached with four cores. The increase of the serial fraction for *omen2* and *omen3* is not that pronounced. Here, the performance peak is reached with eight cores. Only with *omen1*, where the separator stays small, does performance increase up to 16 cores. All other architectures follow this pattern up to the number of cores used.

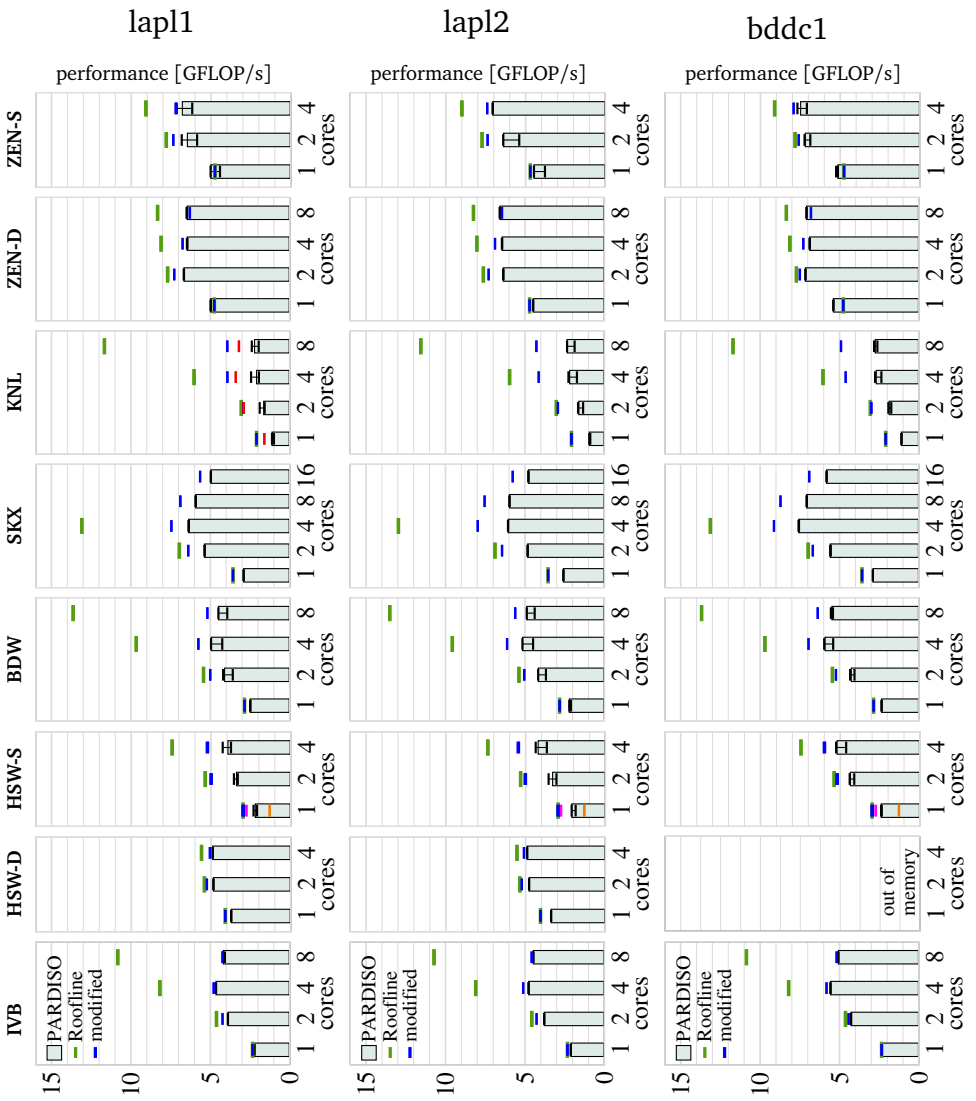


Figure 4.2. Performance of sparse triangular solve with first half of benchmark matrices of panel size $s = 80$ on one NUMA LD for each hardware system. Green and blue bars show predictions of the original and modified roofline model, respectively. HSW-S and ZEN-S have 14 and 24 cores in total and using all cores could achieve, with correct NUMA placement, a higher performance, respectively. Orange and magenta bars show the worst and the best sequential performance predictions of the ECM model for the HSW-S architecture.

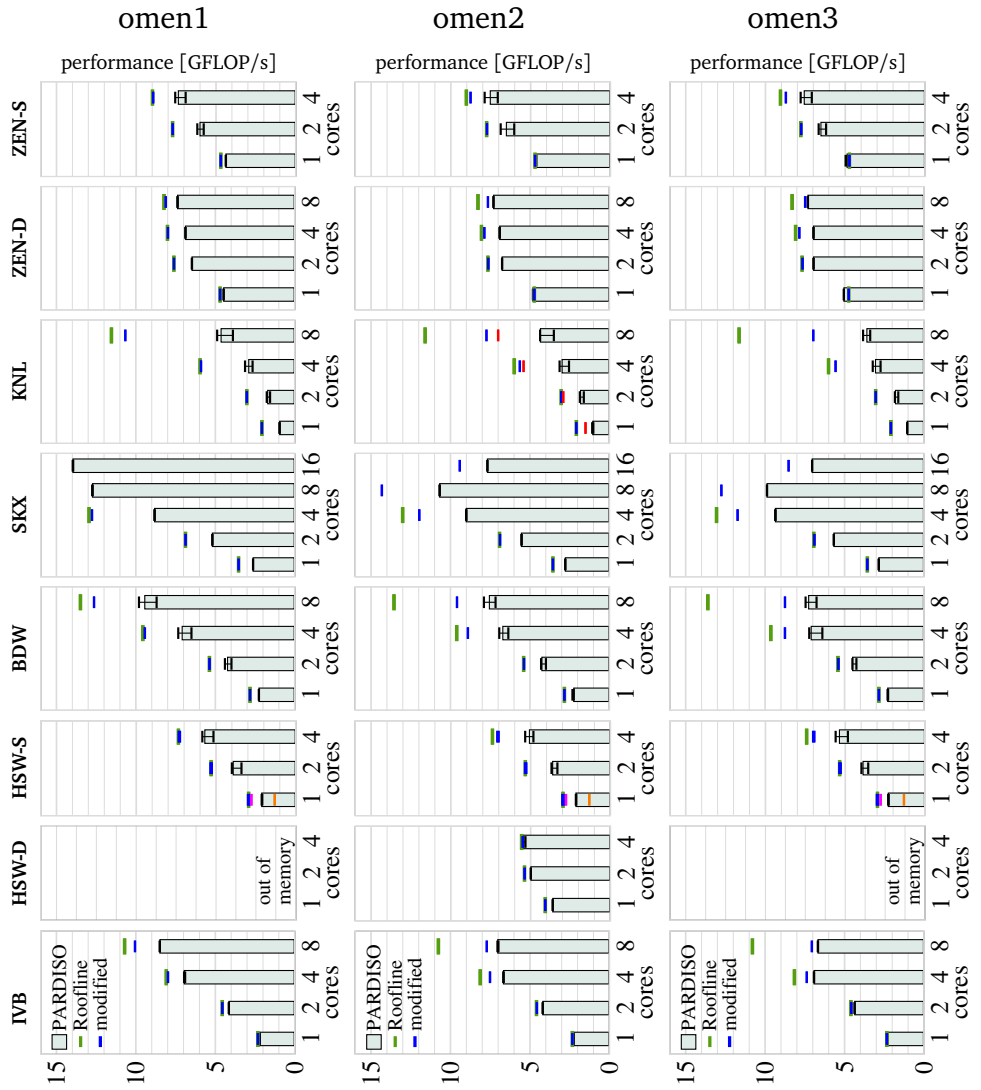


Figure 4.3. Performance of sparse triangular solve with second half of benchmark matrices of panel size $s = 80$ on one NUMA LD for each hardware system. Green and blue bars show predictions of the original and modified roofline model, respectively. HSW-S and ZEN-S have 14 and 24 cores in total and using all cores could achieve, with correct NUMA placement, a higher performance, respectively. Orange and magenta bars show the worst and the best sequential performance predictions of the ECM model for the HSW-S architecture.

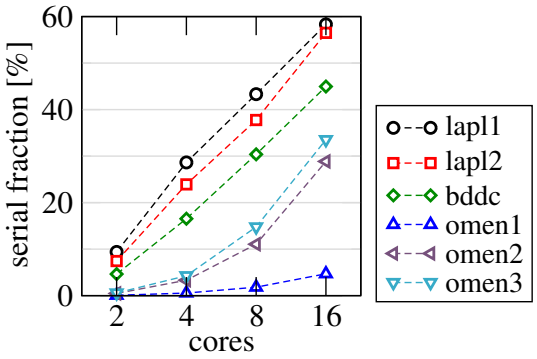


Figure 4.4. Fraction of nonzeros in the separator, which must be processed serially, depending on the number of cores for test matrices.

Figures 4.2 and 4.3 also show the performance limits for the original (3.2) and the modified roofline model (3.3) as green and blue bars, respectively. As the traditional model misses the serial fraction of the execution it scales with the achievable bandwidth of the number of cores used. This is increasingly pronounced, the larger the serial fraction becomes. Systems saturating with one core nearly the total memory bandwidth exhibit only a marginal gap, like HSW-D (second column in Figures 4.2 and 4.3) or slightly more pronounced ZEN-D (seventh column in Figures 4.2 and 4.3).

In general the modified roofline model captures the behavior of the measured performance over all systems and matrices as Figures 4.2 and 4.3 show. IVB and HSW-D, ZEN-D, and ZEN-S exhibit a model error of around 20 % shown in the bottom row of Figures 4.2 and 4.3. For HSW-S, BDW, and SKX the model deviates up to 60 % from the measurements. As measured data traffic is in the expected bounds a deeper analysis of the architectures and algorithm is required. One option for future investigations is the ECM performance model (Hager et al. [2016]), which not only accounts for all transfers inside the memory hierarchy but also performs an in-depth analysis of the execution in the core. For KNL the bottleneck with multiple cores becomes more complex than with a single core. The L2 bandwidth scales linearly with each core, whereas the memory bandwidth scales only with pairs of cores, i. e., per tile, and saturates at a certain point. Depending on the matrix and the size of its separator, L2 or memory bandwidth can now be the bottleneck. In order to keep the model simple, for KNL, we show as red bars in Figures 4.2 and 4.3, in addition to the modified roofline model, an adjusted version; cf. the single core measurements. Here, we consider *lapl1* and *omen2*. Note that all measurements might suffer from the poor performance of 2-way column unrolling. HSW-D and ZEN-D can nearly saturate the memory

bandwidth with one core and gain nearly no benefit from using more than two cores.

Figures 4.2 and 4.3 also show the worst and the best performance predictions of the ECM model as orange and magenta bars, respectively. In Section 3.4 the performance predictions for forward and backward substitution are computed. The sparse triangular solve consists of the forward substitution followed by the backward substitution. For this reason the avarage of the two is shown in the figures. The ECM model is created only for one core of the HSW-S architecture.

Chapter 5

Conclusion

PARDISO and other state-of-the-art sparse direct solvers utilize various techniques to improve performance. First, a reordering of the matrix is found. A graph partitioning algorithm, called nested dissection, produces a reordering that minimizes fill-in and provides a high level of concurrency in factorization. Then, the matrix is factorized using LU, LDL^T, or LL^T algorithm. Last, the solution is obtained by forward and backward substitution. In this thesis we focus on the last step, the forward and backward substitution. Certain applications require solving many times the same linear system A with different right-hand sides, and thus the forward and backward substitution is essential for performance of these applications. Such applications are, for example, FETI domain decomposition methods or certain optimization problems.

The Berkeley roofline model is widely used to visualize the performance of executed code together with the upper performance bounds given by the memory bandwidth and the processor peak performance. The model can hereby provide an insightful visualization of bottlenecks. In this thesis a modification of the roofline model that allows us to cover combination of serial and parallel execution is introduced. This modified form of the roofline model is then applied to the triangular solve step of PARDISO. The performance of the forward and backward substitution process has been analyzed and benchmarked for a representative set of sparse matrices on various modern x86-type multicore architectures and the Knights Landing manycore architecture. It has been shown how to also accurately measure the necessary quantities, such as the achievable memory bandwidth, for threaded code. The measurement approach, its validation, as well as limitations were discussed. Besides the Berkeley roofline model we also used the Erlangen ECM model. This model additionally accounts for in-cache traffic and the complete in-core execution. This advanced model delivers excel-

lent predictions for vectorized code. However, modeling scalar code execution, as it is used in sparse triangular solve, is still under development. For the scalar code of sparse triangular solve we got a reasonable upper and lower bound of performance, but we did not achieve such good predictions as for the vectorized codes.

In the current factorization algorithm, the serial fraction of the factor L increases with the number of cores. As a result, the highest performance for the sparse triangular solve phase is, typically, reached with four or eight cores. PARDISO's high performance sparse triangular solve favors hardware with a high memory bandwidth that can ideally be saturated with one or two cores.

The work presented in this thesis could be extended in several ways. One possible direction is to focus on the ECM model to improve performance predictions of sequential triangular solve and then apply it on the parallel code. One could also focus on the performance modeling of other kernels in linear solvers. The best candidate here is the sparse factorization algorithm, as this is the kernel with the longest execution time. Another direction is to focus on the extended roofline model presented in this thesis and use it for other applications that combine sequential and parallel execution.

Bibliography

Abdelfattah, A., Anzt, H., Boman, E., Carson, E., Cojean, T., Dongarra, J., Fox, A., Gates, M., Higham, N., Li, X., Loe, J., Luszczek, P., Pranesh, S., Rajamanickam, S., Ribizel, T., Smith, B., Swirydowicz, K., Thomas, S., Tomov, S. and Yang, U. [2021]. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic, The International Journal of High Performance Computing Applications p. 109434202110033. doi:10.1177/10943420211003313.

Aho, A., Hopcroft, J. and Ullman, J. [1983]. Data structures and algorithms, Addison-Wesley.

Alappat, C., Laukemann, J., Gruber, T., Hager, G., Wellein, G., Meyer, N. and Wettig, T. [2020]. Performance modeling of streaming kernels and sparse matrix-vector multiplication on a64fx, 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 1–7.

Amestoy, P., Duff, I. and L'Excellent, J.-Y. [2000a]. Multifrontal parallel distributed symmetric and unsymmetric solvers, Comput. Methods in Appl. Mech. Eng. **184**(2–4): 501 – 520. doi:10.1016/S0045-7825(99)00242-X.

Amestoy, P. R., Duff, I. S. and L'Excellent, J.-Y. [2000b]. Multifrontal parallel distributed symmetric and unsymmetric solvers, Comput. Methods Appl. Mech. Engrg. **184**: 501–520.

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y. and Koster, J. [2001]. A fully asynchronous multifrontal solver using distributed dynamic scheduling, SIAM J. Matrix Anal. Appl. **23**(1): 15–41. doi:10.1137/S0895479899358194.

Amestoy, P. R., Guermouche, A., L'Excellent, J.-Y. and Pralet, S. [2006]. Hybrid scheduling for the parallel solution of linear systems, Parallel Computing **32**(2): 136 – 156. doi:10.1016/j.parco.2005.07.004.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A. and Sorensen, D. [1999]. LAPACK Users' Guide, Third Edition, SIAM Publications.

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D. and Yelick, K. [2009]. A view of the parallel computing landscape, Commun. ACM **52**(10): 56–67.
URL: *https://doi.org/10.1145/1562764.1562783*

Azad, A., Ballard, G., Buluç, A., Demmel, J., Grigori, L., Schwartz, O., Toledo, S. and Williams, S. [2016]. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication, SIAM Journal on Scientific Computing **38**(6): C624–C651.
URL: *http://dx.doi.org/10.1137/15M104253X*

Bader, D. A. [2007]. Petascale Computing: Algorithms and Applications, Chapman & Hall/CRC Computational Science Series.

Balaprakash, P., Dongarra, J., Gamblin, T., Hall, M., Hollingsworth, J. K., Norris, B. and Vuduc, R. [2018]. Autotuning in high-performance computing applications, Proceedings of the IEEE **106**(11): 2068–2083.

Benzi, M., Haws, J. and Tuma, M. [2000]. Preconditioning highly indefinite and nonsymmetric matrices, SIAM J. Sci. Comput. **22**(4): 1333–1353.

Berge, C. [1957]. Two theorems in graph theory, Proceedings of National Academy of Science, USA, pp. 842–844.

Bientinesi, P., Quintana-Orti, E. S. and van de Geijn, R. [2005]. Representing linear algebra algorithms in code: The FLAME APIs., ACM Trans. Math. Softw. **31**(1): 1–26.

Bollhöfer, M., Schenk, O., Janalik, R., Hamm, S. and Gullapalli, K. [2020]. State-of-the-Art Sparse Direct Solvers, Birkhäuser, Cham, pp. 1–32.
URL: *https://doi.org/10.1007/978-3-030-43736-7_1*

Callahan, D., Cocke, J. and Kennedy, K. [1988]. Estimating interlock and improving balance for pipelined architectures, J. Parallel. Distr. Com. **5**(4): 334–358. doi:10.1016/0743-7315(88)90002-0.

Chen, Y., Davis, T. A., Hager, W. W. and Rajamanickam, S. [2008]. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate, ACM Trans. Math. Softw. **35**(3).
URL: *https://doi.org/10.1145/1391989.1391995*

Chevalier, C. and Pellegrini, F. [2008]. PT-SCOTCH: a tool for efficient parallel graph ordering, Parallel Comput. **34**(6–8): 318–331.

Choi, J. W., Bedard, D., Fowler, R. and Vuduc, R. [2013]. A roofline model of energy, 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 661–672.

Davis, T. [2006]. Direct Methods for Sparse Linear Systems.
URL: *http://epubs.siam.org/doi/abs/10.1137/1.9780898718881*

Davis, T. A. and Duff, I. S. [1997]. An unsymmetric-pattern multifrontal method for sparse LU factorization, SIAM J. Matrix Anal. Appl. **18**(1): 140–158.

Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R. C. and Yelick, K. [2005]. Self adapting linear algebra algorithms and software, Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation **93**(2).

Ding, N. and Williams, S. [2019]. An instruction roofline model for gpus, 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 7–18.

Dolbeau, R. [2018]. Theoretical peak flops per instruction set: a tutorial, The Journal of Supercomputing **74**(3). doi:10.1007/s11227-017-2177-5.

Dongarra, J. and Sukkari, D. [2021]. Freely available software for linear algebra (april 2021), https://docs.google.com/spreadsheets/d/11ESR3uucNvVKEoIcalP9gR7ApaOElLwmE5sAS-VRMOM/edit#gid=90156307.

Druinsky, A., Ghysels, P., Li, X. S., Marques, O., Williams, S., Barker, A., Kalchev, D. and Vassilevski, P. [2016]. Comparative performance analysis of coarse solvers for algebraic multigrid on multicore and manycore architectures, in R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, J. Kitowski and K. Wiatr (eds), Parallel Processing and Applied Mathematics, Springer International Publishing, Cham, pp. 116–127.

Duff, I. S. and Koster, J. [1999]. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices, SIAM J. Matrix Anal. Appl. **20**: 889–901.

Duff, I. S. and Pralet, S. [2004]. Strategies for scaling and pivoting for sparse symmetric indefinite problems, Technical Report TR/PA/04/59, CERFACS, Toulouse, France.

Ericsson, T. and Ruhe, A. [1980]. he spectral transformation lánczos method for the numerical solution of large sparse generalized symmetric eigenvalue problems, Mathematics of Computation **35**(152): 1251–1268.

Fiduccia, C. M. and Mattheyses, R. M. [1997]. A linear-time heuristic for improving network partitions, Proceedings of the 19th Design Automation Conference, IEEE, pp. 175–181.

Fiedler, M. [1975]. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory, Czecheslovak Mathematical Journal **25**(100): 619–633.

Fog, A. [2016]. Optimization manuals - 3. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers, `http://www.agner.org/optimize/microarchitecture.pdf`.

George, J. A. and Liu, J. W. [1981]. Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, Englewood Cliffs, NJ, USA.

George, J. A. and Ng, E. [1985]. An implementation of Gaussian elimination with partial pivoting for sparse systems, SIAM J. Sci. Statist. Comput. **6**(2): 390–409.

Gilbert, J. R. [1994]. Predicting structure in sparse matrix computations, SIAM J. Matrix Anal. Appl. **15**(1): 162–79.

Gould, N. I. M., Scott, J. A. and Hu, Y. [2007]. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations, ACM Trans. Math. Softw. **33**(2): 10–es.
URL: *https://doi.org/10.1145/1236463.1236465*

Gropp, W. D., Kaushik, D. K., Keyes, D. E. and Smith, B. F. [1999]. Towards realistic performance bounds for implicit CFD codes, Proc. Parallel CFD'99, pp. 233–240.

Gupta, A. [2002]. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices, SIAM J. Matrix Anal. Appl. **24**: 529 – 552.

Gupta, A. and Ying, L. [1999]. On algorithms for finding maximum matchings in bipartite graphs, Technical Report RC 21576 (97320), IBM T. J. Watson Research Center, Yorktown Heights, NY.

Hager, G., Treibig, J., Habich, J. and Wellein, G. [2016]. Exploring performance and power properties of modern multicore chips via simple machine models, Concurr. Comput. . doi:10.1002/cpe.3180.

Heath, M. T. and Raghavan, P. [1999]. Performance of Parallel Sparse Triangular Solution, pp. 289–305. doi:10.1007/978-1-4612-1516-5_13.

Hockney, R. W. and Curington, I. J. [1989]. $f_{1/2}$: A parameter to characterize memory and communication bottlenecks, Parallel Computing **10**(3): 277–286. doi:10.1016/0167-8191(89)90100-2.

Hofmann, J., Fey, D., Eitzinger, J., Hager, G. and Wellein, G. [2016]. Analysis of Intel's Haswell Microarchitecture Using the ECM Model and Microbenchmarks, pp. 210–222. doi:10.1007/978-3-319-30695-7_16.

Intel Corp. [2016]. Intel64 and IA-32 Architectures Optimization Reference Manual. Version: June 2016.

Jacob, B., Ng, S. and Wang, D. [2007]. Memory Systems: Cache, DRAM, Disk.

Kardoš, J., Kourounis, D. and Schenk, O. [2020]. Two-level parallel augmented schur complement interior-point algorithms for the solution of security constrained optimal power flow problems, IEEE Transactions on Power Systems **35**(2): 1340–1350. doi:10.1109/TPWRS.2019.2942964.

Karypis, G. and Kumar, V. [1998]. A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing **20**(1): 359–392.

Karypis, G., Schloegel, K. and Kumar, V. [1999]. ParMeTis: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 2.0, University of Minnesota, Dept. of Computer Science.
URL: *http://www-users.cs.umn.edu/~karypis/metis/parmetis/main.html*

Kernighan, B. and Lin, S. [1970]. An efficient heuristic procedure for partitioning graphs, The Bell System Technical Journal **29**(2): 291–307.

Klawonn, A., Lanser, M., Rheinbach, O., Stengel, H. and Wellein, G. [2015]. Hybrid MPI/OpenMP Parallelization in FETI-DP Methods. LNSCE, vol. 105, doi:10.1007/978-3-319-22997-3_4.

Klawonn, A. and Rheinbach, O. [2010]. Highly scalable parallel domain decomposition methods with an application to biomechanics, ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik **90**: 5 – 32.

Köstler, H. and Rüde, U. [2013]. The cse software challenge - covering the complete stack, **55**(3): 91–96.
URL: *https://doi.org/10.1524/itit.2013.0010*

Kothe, D. and Kendall, R. [2007]. Computational science requirements for leadership computing, Technical Report ORNL/TM-2007/44, Oak Ridge National Laboratory.

Kreutzer, M., Hager, G., Wellein, G., Pieper, A., Alvermann, A. and Fehske, H. [2015]. Performance engineering of the Kernel Polynomial Method on large-scale CPU-GPU systems, Proc. IPDPS 2015, pp. 417–426. doi:10.1109/IPDPS.2015.76.

Kuzmin, A., Luisier, M. and Schenk, O. [2013]. Fast Methods for Computing Selected Elements of the Green's Function in Massively Parallel Nanoelectronic Device Simulations, Vol. 8097 of Lecture Notes in Computer Science, pp. 533–544. doi:10.1007/978-3-642-40047-6_54.

Langguth, J., Azad, A. and Manne, F. [2014]. On parallel push-relabel based algorithms for bipartite maximum matching, Parallel Comput. **40**(7): 289–308.

Langguth, J., Patwary, M. M. A. and Manne, F. [2011]. Parallel algorithms for bipartite matching problems on distributed memory computers, Parallel Comput. **37**(12): 820–845.

LaSalle, D. and Karypis, G. [2013]. Multi–threaded graph partitioning, Technical report, Department of Computer Science & Engineering, University of Minnesota, Minneapolis.

Lee, B. C., Vuduc, R., Demmel, J. and Yelick, K. [2004]. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply, Proceedings of the International Conference on Parallel Processing, Montreal, Quebec, Canada.

Li, X. S. [2005]. An overview of SuperLU: Algorithms, implementation, and user interface, ACM Trans. Math. Softw. **31**(3): 302–325.

Li, X. S. [2008]. Evaluation of sparse LU factorization and triangular solution on multicore platforms, VECPAR 2008, pp. 287–300.

Li, X. S. and Demmel, J. W. [2003]. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Trans. Math. Software **29**(2): 110–140.

Lin, C.-P., Xie, H., Grimes, R. and Bai, Z. [2021]. On the shift-invert lanczos method for the buckling eigenvalue problem, International Journal for Numerical Methods in Engineering **122**(11): 2751–2769.
URL: *https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.6640*

Liu, W., Li, A., Hogg, J., Duff, I. S. and Vinter, B. [2016]. A synchronization-free algorithm for parallel sparse triangular solves, Euro-Par 2016: Parallel Processing, pp. 617–630.

Luisier, M., Boykin, T. B., Klimeck, G. and Fichtner, W. [2011]. Atomistic nanoelectronic device engineering with sustained performances up to 1.44 pflop/s, Proc. SC11, ACM.

Marques, D., Ilic, A., Matveev, Z. A. and Sousa, L. [2020]. Application-driven cache-aware roofline model, Future Generation Computer Systems **107**: 257–273.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X19309586*

Marrakchi, S. and Jemni, M. [2017]. Fine-grained parallel solution for solving sparse triangular systems on multicore platform using openmp interface, Proc. HPCS17, pp. 659–666.

McCalpin, J. D. [1995]. Memory bandwidth and machine balance in current high performance computers, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25.

Meca, O., Riha, L., Markopoulos, A., Brzobohaty, T. and Kozubek, T. [2018]. Using espreso as linear solver library for third party fem tools for solving large scale problems, in T. K. Jaros (ed.), HIGH PERFORMANCE COMPUTING IN SCIENCE AND ENGINEERING, HPCSE 2017, Vol. 11087 of Lecture Notes in Computer Science, VSB Tech Univ Ostrava, IT4Innovat Natl Supercomputing Ctr, SPRINGER INTERNATIONAL PUBLISHING AG, GEWERBESTRASSE 11,

CHAM, CH-6330, SWITZERLAND, pp. 130–143. 3rd International Conference on High Performance Computing in Science and Engineering (HPCSE), Karolinka, CZECH REPUBLIC, MAY 22-25, 2017.

Minami, K. [2019]. Performance Optimization of Applications, Springer Singapore, Singapore, pp. 11–39.
URL: *https://doi.org/10.1007/978-981-13-9802-5_2*

Nguyen, T., Williams, S., Siracusa, M., MacLean, C., Doerfler, D. and Wright, N. J. [2020]. The performance and energy efficiency potential of fpgas in scientific computing, 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 8–19.

Olschowka, M. and Neumaier, A. [1996]. A new pivoting strategy for Gaussian elimination, Linear Algebra and its Applications **240**: 131–151.

Park, J., Smelyanskiy, M., Sundaram, N. and Dubey, P. [2014]. Sparsifying synchronization for high-performance shared-memory sparse triangular solver, Proc. ISC 2014, pp. 124–140.

Reed, D., Bajcsy, R., Fernandez, M., Griffiths, J.-M., Mott, R., Dongarra, J., Johnson, C., Inouye, A., Miner, W., Matzke, M. and Ponick, T. [2005]. Computational science: Ensuring america's competitiveness, p. 117.

Riha, L., Merta, M., Vavrik, R., Brzobohaty, T., Markopoulos, A., Meca, O., Vysocky, O., Kozubek, T. and Vondrak, V. [2019]. A massively parallel and memory-efficient fem toolbox with a hybrid total feti solver with accelerator support, INTERNATIONAL JOURNAL OF HIGH PERFORMANCE COMPUTING APPLICATIONS **33**(4): 660–677.

Schenk, O. [2000]. Scalable parallel sparse LU factorization methods on shared memory multiprocessors, PhD thesis, ETH Zurich.

Schenk, O. and Gärtner, K. [2004]. Solving unsymmetric sparse systems of linear equations with PARDISO, J. Future. Gener. Comp. Sy. **20**(3): 475–487. doi:`10.1016/j.future.2003.07.011`.

Schenk, O. and Gärtner, K. [2006]. On fast factorization pivoting methods for symmetric indefinite systems, Electronic Transactions on Numerical Analysis **23**(1): 158–179.

Schönauer, W. [2000]. Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers.
URL: *http://www.rz.uni-karlsruhe.de/ rx03/book*

Stengel, H., Treibig, J., Hager, G. and Wellein, G. [2015]. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model, Proc. ICS'15, pp. 207–216. doi:10.1145/2751205.2751240.

Treibig, J. and Hager, G. [2010]. Introducing a Performance Model for Bandwidth-Limited Loop Kernels. doi:10.1007/978-3-642-14390-8_64.

Treibig, J., Hager, G. and Wellein, G. [2010]. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments, CoRR **abs/1004.4431**.
URL: *http://arxiv.org/abs/1004.4431*

Vuduc, R., Kamil, S., Hsu, J., Nishtala, R., Demmel, J. W. and Yelick, K. A. [2002]. Automatic performance tuning and analysis of sparse triangular solve, ICS.

Vuduc, R. W. [2003]. Automatic performance tuning of sparse matrix kernels, PhD thesis, University of California, Berkeley.

Whaley, R. C. and Petitet, A. [2005]. Minimizing development and maintenance costs in supporting persistently optimized BLAS, Software: Practice and Experience **35**(2): 101–121. http://www.cs.utsa.edu/~whaley/papers/spercw04.ps.

Whaley, R. C., Petitet, A. and Dongarra, J. J. [2001]. Automated empirical optimization of software and the ATLAS project, Parallel Computing **27**(1–2): 3–35. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (www.netlib.org/lapack/lawns/lawn147.ps).

Williams, S. W. [2008]. Auto-tuning Performance on Multicore Computers, PhD thesis, University of California, Berkeley.

Williams, S., Waterman, A. and Patterson, D. [2009]. Roofline: an insightful visual performance model for multicore architectures, Commun. ACM **52**(4): 65–76. doi:10.1145/1498765.1498785.

Yang, C., Kurth, T. and Williams, S. [2020]. Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system, Concurrency and Computation: Practice and Experience **32**(20): e5547.

e5547 cpe.5547.
URL: *https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5547*