DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Computational Science and Engineering

# Thesis title

**Ravil Dorozhinskii**

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Computational Science and Engineering

# Thesis title

# Titel der Abschlussarbeit

| | |
|---|---|
| Author: | Ravil Dorozhinskii |
| Supervisor: | Supervisor |
| Advisor: | Advisor |
| Submission Date: | Submission date |

I confirm that this master's thesis in informatics: computational science and engineering is my own work and I have documented all sources and material used.

Munich, Submission date　　　　　　　　Ravil Dorozhinskii

Acknowledgments

# Abstract

Parallel linear equation solvers are one of the most important components determining the scalability and efficiency of many supercomputing applications.

In many HPC applications, the optimized single-node performance is one of the most crucial building blocks for the optimization of probabilistic analysis or hybrid MPIOpenMP implementation.

reformulate the statement

# Contents

# 1. Solver selection and configuration

## 1.1. Problem statement

from PDEs discretization to ODE

Numerical integration of time dependent partial differential equations can be achieved using different numerical schemes, namely: Runge-Kutta methods. In numerical analysis, the Runge–Kutta methods are a family of implicit and explicit iterative methods [**wiki:runge-kutta**]. Implicit and explicit methods have their advantages and disadvantages which can be found in the corresponding literature.

In general, implicit methods are robust in case of stiff equations, however, on another hand they are more expensive in terms of computational cost since they introduce a non-linear part that has to be solve using the Newton's method as an example. There is also a class of integration methods called semi-implicit that were designed to reduce the cost of implicit ones and be able to cope with stiff equations. But in all cases numerical integration boils down to solution of a couple systems of linear equations of a type:

$$Ax = b \tag{1.1}$$

where $A \in \mathbb{R}^{\mathbb{N}\times\mathbb{N}}$ is an invertible square matrix, $b \in \mathbb{R}^{\mathbb{N}}$ represents the right-hand side, and $x \in \mathbb{R}^{\mathbb{N}}$ is the solution vector. At this point we refer to a system of linear equations as just a system and we will use these two terms interchangeably.

In some cases, especially in case of implicit or semi-implicit methods, we have to compute several systems to perform one step of numerical integration. Thus, solution of a system 1.1 is the computational core of any time integration scheme. In fact, it is the most expensive part and it is primary source of code optimization.

There are three major ways to solve a system of linear equations, namely: direct dense methods, sparse direct methods and iterative methods.

In the following sections 1.3, 1.4, 1.5 we are briefly going to review all three types of numerical solvers as well as their advantages and disadvantages and some aspects of their parallel implementations with a strong focus on their strong scaling behavior.

Strong scaling is important for this research since we want to find a solver and its configuration to solve a certain system with a fixed size as fast as possible. Additionally to that, due to a large number of numerical integration steps, we want to find a robust solver to avoid any application crash during simulations.

At this point we can set requirements for a solver that we are looking for:

- robustness
- numerical stability
- parallel efficiency
- open source licenses

## 1.2. Matrix sets, hardware, compilers and MPI library

NAME: experimental setup

During the stiudy, we we using two matrix sets: GRS and SuiteSparse. In our case, the SiteSparse matrix set was, in fact, few matrices downloaded from SuiteSparse Matrix Collection [**sparse-matrix-collection:1**], [**sparse-matrix-collection:2**]. We tried to choose different matrices from the collection with respect to both the number of equations $n$ in a system and ratio $R$ between the number of non-zero elements $nnz$ and the number of equations.

describe what ATHLET is

To generate GRS matrix set, we ran the most common GRS simulations in ATHLET and stopped the simulations somewhere in the middle saving corresponding shifted Jacobian matrices in the PETSc binary format.

describe what a shifted Jacobian matrix is

Tables 1.1 and 1.2 shows matrix properties of both matrix sets.

a mistake for pwr and cube-5

From now, we are going to introduce and use a definition of **skinny sparse matrices**. It is matrices with relatively low ration $R$ i.e. less than 15.

add sparsity plots to appendix

add sparsity plots to appendix

The objective of this study was to find and configure a solver which could fulfill all requirements listed above for the GRS matrix set. From time to time, we used the SuiteSparse set for comparison reasons. It seems to us that GRS matrix set was different because of some specifics of 1D pipeline discretization. For example, GRS matrices are skinny and blocked, where each block is a small, approximately 3-by-3, dense matrix. Additionally, some rows can contain only one element i.e. on the diagonal. It is a

| Name | n | nnz | nnz / n |
|---|---|---|---|
| pwr-3d | 6009 | 32537 | 5.4147 |
| cube-5 | 9325 | 117897 | 12.6431 |
| cube-64 | 100657 | 1388993 | 13.7993 |
| cube-645 | 1000045 | 13906057 | 13.9054 |
| k3-2 | 130101 | 787997 | 6.0568 |
| k3-18 | 1155955 | 7204723 | 6.2327 |

Table 1.1.: GRS matrix set

| Name | n | nnz | nnz / n | Field |
|---|---|---|---|---|
| cant | 62451 | 4007383 | 64.1684 | 2D/3D Problem |
| consph | 83334 | 6010480 | 72.1251 | 2D/3D Problem |
| CurlCurl_3 | 1219574 | 13544618 | 11.1060 | Model Reduction Problem |
| Geo_1438 | 1437960 | 63156690 | 43.9210 | 2D/3D Problem |
| memchip | 2707524 | 13343948 | 4.9285 | Circuit Simulation Problem |
| PFlow_742 | 742793 | 37138461 | 49.9984 | 2D/3D Problem |
| pkustk10 | 80676 | 4308984 | 53.4110 | Structural Problem |
| torso3 | 259156 | 4429042 | 7.0903 | 2D/3D Problem |
| x104 | 108384 | 8713602 | 80.3956 | Structural Problem |

Table 1.2.: SuiteSparse matrix set

result of dynamic pipeline switching of a reactor cooling system. As we will see in section 1.5, sparse direct solvers can be quite sensitive to the sparse structure of a matrix.

We used different hardware to measure performance of different solvers. The first machine was the GRS cluster (HW1) which was our main target. We used a LRZ CoolMUC-2 Linux cluster (HW2) every time when we got some ambiguous results in order to check whether a problem was hardware or software specific. Table 1.3 shows a single node specification of both machines.

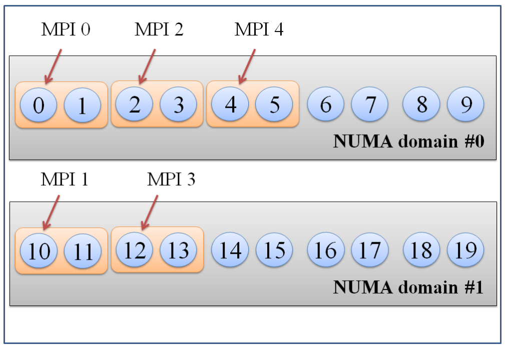
We decided to stick to the OpenMPI library which is an implementation of the MPI standard. OpenMPI is an open-source project and it is very well documented. The library has many options for processes pinning which was a crucial part in our study because we had to deal with multi-socket machines which, in turn, had multiple NUMA domains.

| | HW1 (GRS) | HW2 (LRZ Linux) |
|---|---|---|
| Architecture | x86_64 | x86_64 |
| CPU(s) | 20 | 28 |
| On-line CPU(s) list | 0-19 | 0-27 |
| Thread(s) per core | 1 | 1 |
| Core(s) per socket | 10 | 14 |
| Socket(s) | 2 | 2 |
| NUMA node(s) | 2 | 4 |
| Model | 62 | 63 |
| Model name | E5-2680 v2 | E5-2697 v3 |
| Stepping | 4 | 2 |
| CPU MHz | 1200.0 | 2036.707 |
| Virtualization | VT-x | VT-x |
| L1d cache | 32K | 32K |
| L1i cache | 32K | 32K |
| L2 cache | 256K | 256K |
| L3 cache | 25600K | 17920K |
| NUMA node0 CPU(s) | 0-9 | 0-6 |
| NUMA node1 CPU(s) | 10-19 | 7-13 |
| NUMA node2 CPU(s) | - | 14-20 |
| NUMA node3 CPU(s) | - | 21-27 |

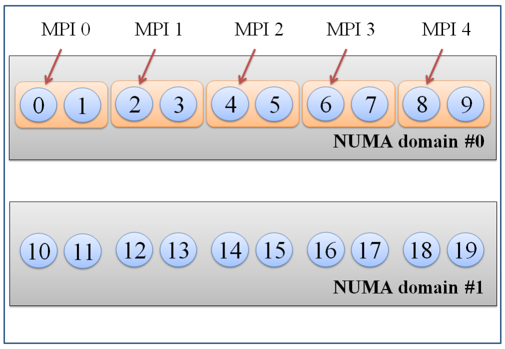
Table 1.3.: Hardware specification

To make process pinning deterministic, we developed a python script which automatically generated rank-files based on the number of MPI processes, OpenMP threads per MPI process, the maximum number of processing elements and the number of NUMA domains. The scrip always leaves appropriate gaps between MPI processes to allow each process to fork the corresponding number of threads within a parallel region.

A rank-file specifies explicit mapping between MPI processes (ranks) and actual processing elements (cores) within a machine. The script has two modes, namely: *spread* and *close*. Given a certain number of ranks, the spread mode tries to distribute them as spread as possible across available NUMA domains in round-robin fashion. In contrast to the spread strategy, the close mode groups ranks as close as possible to keep the maximum number of ranks within a single NUMA domain. Figure 1.1 shows an example of how these two modes work in case of 5 MPI ranks, 2 OpenMP threads per rank, on a compute node equipped with 20 cores and 2 NUMA domains (HW1).



(a) *Spread* mode



(b) *Close* mode

Figure 1.1.: An example of process pinning of 5 MPI processes with 2 OpenMP threads per rank in case of HW1

place for figure 1.1

We chose **Intel 18.2** compiler for this study as the newest and the most efficient Intel compiler at the time of writing.

## 1.3. Direct dense methods

MUST BE ONLY 2D Scalapack EXAMPLE HERE

Direct methods have very good numerical stability properties. They do not require to modify a system of equations as it is necessary for iterative methods. Sometimes we only need to permute some rows and columns in order to avoid small absolute values along the matrix diagonal, which can have their negative effect on numerical accuracy, in case of direct methods. However, this operation can be considered relatively cheap.

On another hand, the computational complexity of $O(n^3)$ and storage requirements of $O(n^2)$ make direct methods not suitable for computation of large systems with more than $\sim 10^3$ number of equations.

Another good property of direct dense solvers is they have direct memory access and they are based on dense linear algebra subroutines. As a result, implementation of these methods can exploit such programming techniques like cache blocking, tiling, data prefetching, utilization of hardware vector units and so on. All of these together make it possible to achieve more than 75% hardware performance of modern CPUs [**articles:blas-performance**] due to a high ratio of floating point operations per memory access.

A general way to solve a system of a type 1.1 is to perform $LU$ factorization where a matrix $A$ can be viewed as a product of a lower and upper triangular matrices.

$$Ax = LUx = b \tag{1.2}$$

Having computed matrices $L$ and $U$, two additional steps, forward and backward substitutions, are required to compute the solution vector $x$.

$$Ly = b \tag{1.3}$$
$$Ux = y \tag{1.4}$$

The main idea of parallel $LU$ decomposition is based on block structure of a matrix $A$. Any given matrix $A$ can be represented as a combination of sub-matrices $A_{ij}$.

$$A = \left(\begin{array}{c|c|c} A_{11} & A_{12} & A_{13} \\ \hline A_{21} & A_{22} & A_{23} \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right)$$

This allows to develop algorithms that can compute $LU$ decomposition of the full matrix $A$ in parallel using sub-matrices with three main routines i.e. matrix-matrix multiply, triangular solve with multiple right hand sides and the unblock $LU$ factorization for operation within a block column [**netlib:lapack-1**].

There exist three well known parallel approaches based on left, right and Crout decomposition forms. All of these three approaches have similar overall performance, with a slight advantage to the right-looking and Crout variants [**netlib:lapack-1**].

Let's consider right-looking algorithm as an example. The algorithm can be written as following:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \cdot \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \tag{1.5}$$

The algorithm starts with an assumption that both matrices $L_{11}$ and $U_{11}$ have already been computed.

Given equation 1.5, we can write expressions for $A_{12}$ and $A_{21}$ matrices:

$$A_{12} = L_{11} \cdot U_{12} \tag{1.6}$$
$$A_{21} = L_{21} \cdot U_{11} \tag{1.7}$$

To compute 1.6 we only have to perform two triangular solves since matrices $L_{11}$ and $U_{11}$ are known. These two tasks are concurrent and they can be computed in parallel. Additionally we can notice the triangular solve routine can be computed in parallel too. The next step is to perform $\hat{A_{22}}$ matrix update and block structure re-odering:

$$\hat{A_{22}} = A_{22} - L_{21} \cdot U_{12} \tag{1.8}$$

place for figure 1.2

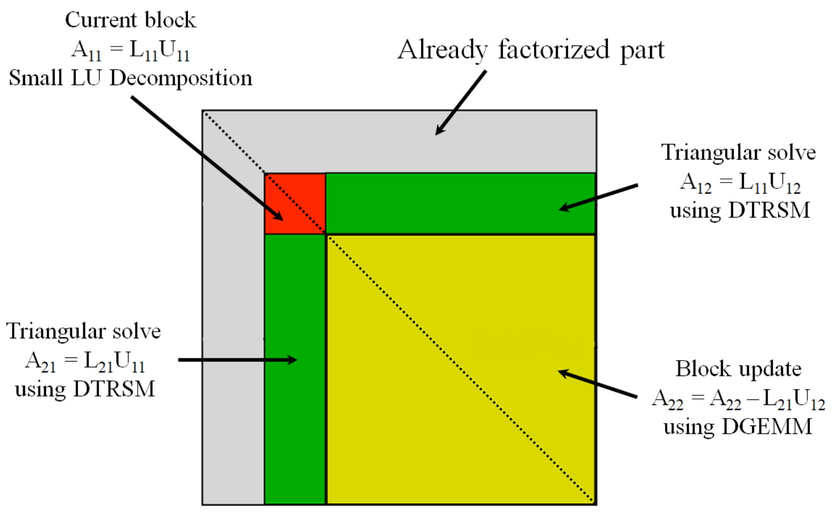


Figure 1.2.: Right-looking parallel LU decomposition

It is clear the algorithm is purely sequential at the first steps when we compute small *LU* decomposition. Therefore, it can have significant effect on algorithm strong scaling behavior. It should be mentioned that all three parallel implementations have the same problem i.e. they have an inherently sequential part at the beginning of a step.

Figure 1.3 shows results of strong scaling of dense *LU* factorization performed for a couple of matrices filled with random numbers with different sizes: namely: $5000 \times 5000$, $10000 \times 10000$ and $15000 \times 15000$. LAPACK and OpenBLAS libraries were used for the test. One can easily notice that performance of dense *LU* decomposition quickly deteriorates with reduction of the problem size. Additional factor that affects performance is strong scaling behavior of the triangular solve that we will discuss in section 1.4.

place for figure 1.3

add strong scaling for ScaLAPACK

Both left, right and Crout parallel matrix factorizations have been efficiently implemented in LAPACK (for shared-memory machines) and ScaLAPACK (distributed-memory machines) libraries. Both libraries belong to the Netlib project which is a

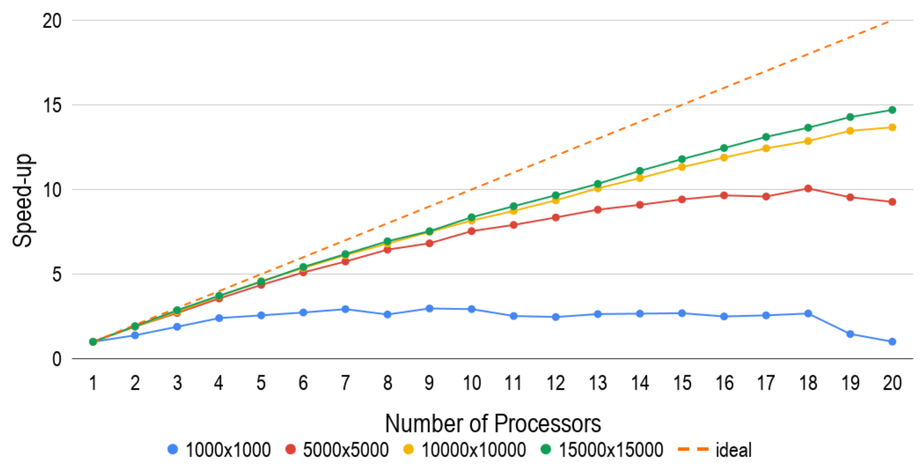


Figure 1.3.: Strong scaling of right-looking *LU* decomposition using LAPACK and OpenBLAS

repository of numerous scientific computing software maintained by AT&T Bell Laboratories, the University of Tennessee, Oak Ridge National Laboratory and other scintific communities [**netlib-overview**]. The libraries are built on top of Basic Linear Algebra Subprograms (BLAS) library. Figure 1.4 shows how these three libraries are coupled together.

place for figure 1.4

It is worth noting that BLAS can be considered as a foundation of LAPACK and ScaLAPACK libraries and, thus, it is the primary source of performance improvement. In particular we have to consider DGEMM, DTRSM BLAS subroutines performance because they, together with unblocked *LU* factorization, compose the core of parallel *LU* factorization algorithms as we discussed earlier.

There exist special-purpose, hardware-specific BLAS implementations developed by the hardware vendors i.e. IBM, Cray, Intel, AMD as well as open-source tuned implementations such as ATLAS, OpenBLAS, etc. We will come back to that discussion later and pay our close attention to a specific choice of a tuned BLAS library in subsection 1.11.

In spite of all advantages of the direct dense solvers i.e. numerical stability and high ratio of floating point operations per memory access, we cannot consider this group of methods as a solver for time integration due to high complexity and storage costs.

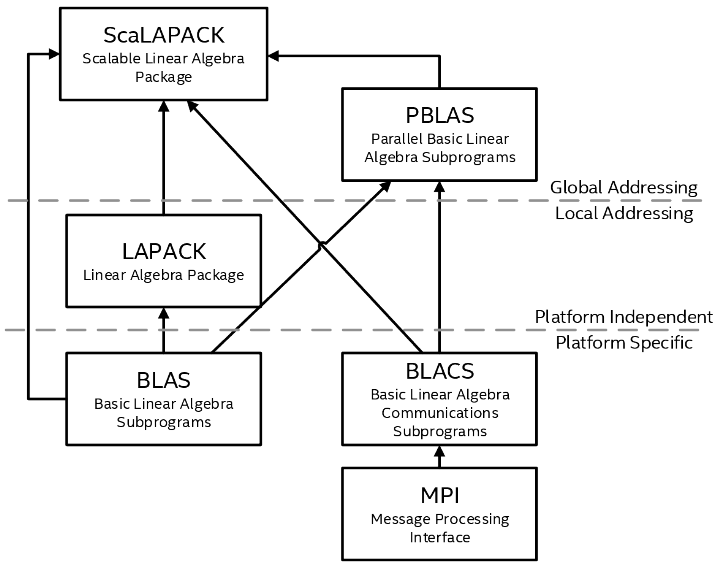

ScaLAPACK
Scalable Linear Algebra Package
PBLAS
Parallel Basic Linear Algebra Subprograms
Global Addressing
Local Addressing
LAPACK
Linear Algebra Package
Platform Independent
Platform Specific
BLAS
Basic Linear Algebra Subprograms
BLACS
Basic Linear Algebra Communications Subprograms
MPI
Message Processing Interface

Figure 1.4.: A general view on BLAS, LAPACK and ScaLAPACK libraries [**netlib:lapack-scalapack-general-view**]

## 1.4. Iterative methods

Iterative methods, especially Krylov subspace methods that we are going to discuss in this section, are well known for their relatively low storage requirements $O(nnz)$ and computation cost $O(N^2)$ in case of sparse linear systems of equations and good condition number. It turns out that sometimes it might be only one way to solve huge systems with millions unknowns.

The most well known methods are Conjugate Gradient (CG) for symmetric positive definite matrices, Minimal Residual Method (MINRES) for symmetric indefinite systems, Generalized Minimal Residual Method (GMRES) for non-symmetric systems of linear equations as well as different variants of GMRES such Biconjugate Gradient Method (BiCG), Biconjugate Gradient Stabilized Method (BiCGSTAB) and so on.

All Krylov methods solve a system of equation as a minimization problem. For example, the goal of CG algorithm is to minimize the energy functional $f(x) = 0.5x^T Ax - b^T x + c$, whereas, MINRES and GMRES tries to minimize residual norm $r_j$ for $x_j$ in a subspace.

The methods construct an approximate solution of a system as a linear combination of vectors $b$, $Ab$, $A^2b$, $A^3b$ and so on which defines the Krylov subspace. At each iteration we expand the subspace adding and evaluating a next vector in the combination.

Let's consider GMRES, as the most popular and general iterative solver, without preconditioning to just analyze its strong scaling behavior and potential problems.

As we mentioned above GMRES minimizes the residual norm in a subspace $U_m$.

$$\underset{x \in U_m}{min} ||Ax - b||^2 \tag{1.9}$$

We can consider a solution vector $x$ in the subspace $U_m$ in a form $x = U_m y$. Thus, equation 1.9 can be written as following:

$$\underset{x \in U_m}{min} ||AU_m y - b||^2 \tag{1.10}$$

The most natural way to choose a proper subspace $U_m$ is the corresponding Krylov subspace $\mathcal{K}_m$ because it can be easily generated on the fly. However, decomposition of vector $x$ in that subspace can be a problem. Since the subspace $\mathcal{K}_m$ is spanned by the sequence of $b$, $Ab$, $A^2b$, ..., $A^{m-1}b$ and due to round-off error the sequence can become linear dependent. Therefore, we have to compute and use the orthonormal base of the given Krylov subspace. Saad and Schultz in their work [**sparse-la:gmrese-origin**] used Arnoldi process for constructing an $l_2$-orthogonal basis. As the results equation 1.10 can be written in the following form:

$$\underset{x \in U_m}{min} ||U_{m+1} H_{m+1,m} y - ||b|| u_1||^2 = \underset{x \in U_m}{min} ||H_{m+1,m} y - ||b|| e_1||^2 \tag{1.11}$$

where $H_m$ is an upper Hessenberg matrix. We can apply Givens rotation algorithm to compute $QR$ decomposition to convert $H_m$ to a strictly upper triangular matrix. Thus,

$$\underset{x \in K_m}{min} ||Ax - b||^2 = \underset{x \in U_m}{min} ||Q^T R y - ||b|| e_1||^2 = \underset{x \in U_m}{min} || \begin{pmatrix} R_m \\ 0 \end{pmatrix} y - \begin{pmatrix} \tilde{b_m} \\ \tilde{b_{n-m}} \end{pmatrix} ||^2 \tag{1.12}$$

Given 1.12, we can compute the solution as following:

$$R_m y = \tilde{b_m} \tag{1.13}$$

$$x_m = U_m y \tag{1.14}$$

Because of large computational and storage costs, in case of evaluation of the full Krylov subspace, only small a subspace is computed, typically first 20 - 50 column vectors. Then the algorithm is restarted using the computed approximate solution as a initial guess for the next iteration.

We can see that some operations, for example 1.14, can be efficiently done in parallel. However, operatiopns like sparse triangular solve 1.13 can introduce some effect on strong scaling behavior. Figure 1.5 shows strong scaling performance results of a sparse parallel triangular solver with a two dimensional matrix distribution. Performance considerations of the solver can be found in [**sparse-la:triangular-solve**].

place for figure 1.5

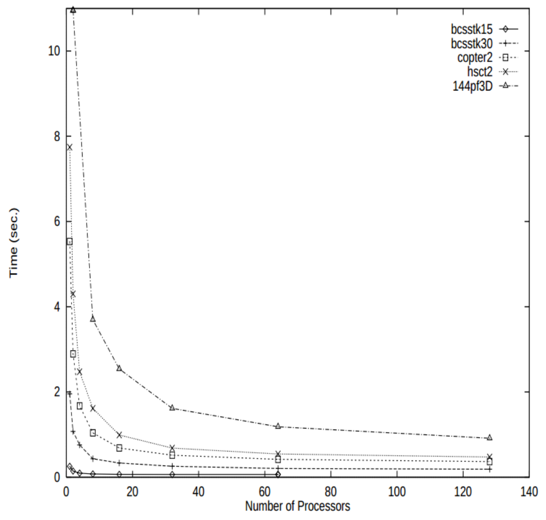


Figure 1.5.: Performance of sparse triangular solve [**sparse-la:triangular-solve**]

It is interesting to notice that performance of the triangular solver depends on a matrix sparsity structure as well as the matrix size.

Triangular solve 1.13 can be computed in a single processor because matrix $R_m$ is usually small and depends on the number iterations before the restart. In this case the triangular solve can become a bottleneck again.

Figure 1.6 shows strong scaling performance results of the default GMRES solver from the PETSc library. The solver was set up without any preconditioner and 50 iterations as the restart. Additionally no stop criteria was specified except the maximum number iterations which was equal to 100. The *spread* process pinning strategy, described in section 1.2, was used. It is well-known that all iterative methods are memory bound due to indirect memory addressing caused by sparse matrix storage schemes. Hence equal process distribution can help reduce the load on memory channels since it is an obvious bottleneck for this type of applications.

Four matrices were chosen form the GRS matrix set for the tests, namely: cube-64, cube-645, k3-2 and k3-18. The information about the matrices is summarized in table 1.1. As we expected, we can observe strong deviation of our results from the ideal speed-up line when the number of processes exceeds 10.

place for figure 1.6

It should be mentioned that parallelization overheads, introduced by such MPI operations as MPI_Send, MPI_Recv, MPI_Allreduce, etc., also have their impact on performance of the algorithm.

Other Krylov methods such as CG, for example, scales much better than GMRES. Because of the nature of the CG algorithm the next search direction can be found using a recurrent expression and the algorithms boils down to simple operations such as dot products and matrix vector multiplications. These operations can be easily parallelized and drop of performance comes only from MPI overheads. A quite comprehensive study about parallel CG algorithm performance can be found in [**sparse-la:cg**]. The authors also introduced a deeply pipelined version of CG algorithm that scales even better due to overlapping the time-consuming global communication phase, induced by parallel dot product computations, with useful independent computations [**sparse-la:cg**].

place for figure 1.6

The most important criteria of Krylov methods is convergence rate. The convergence rate of iterative methods strongly depends on a matrix and, in particular, on its condition number. For instance, equation 1.15 shows dependence of the convergence rate from the matrix condition number. It can be clearly seen that a big condition number leads to very slow error reduction and, as the results, to huge number of iterations.

$$||e^i||_A \leq 2(\frac{\sqrt{k}-1}{\sqrt{k}+1})^i||e^0||_A \tag{1.15}$$

where $k = \frac{\lambda_{max}}{\lambda_{min}}$ - condition number of the corresponding matrix.

An obvious solution of such a problem is to reduce the condition number of the original system 1.1. A general method is to transform the original system in such a way that the conditional number of the transformed system gets significantly smaller. The

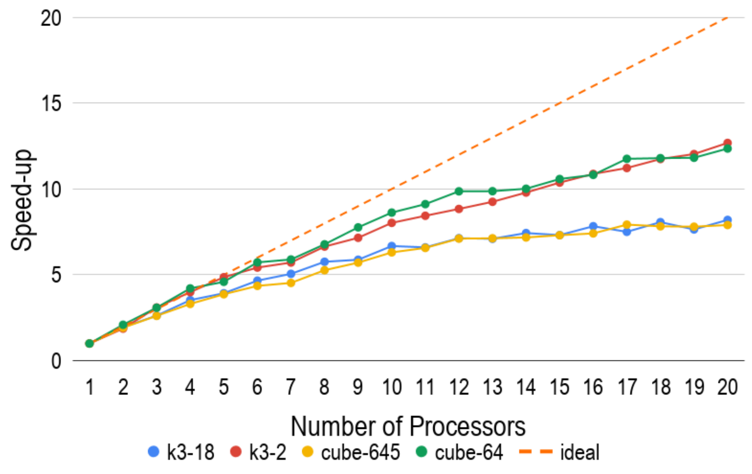


Figure 1.6.: GMRES strong scaling speed-up

transformation of 1.1 can be done from the left side 1.16 or from the right one 1.17.

$$PAx = Pb \tag{1.16}$$

$$AP(P^{-1}x) = b \tag{1.17}$$

where matrix $P$ is called preconditioner.

As an extreme example we can consider the inverse matrix $A^{-1}$ as the best preconditioner since it directly leads to the solution of the problem 1.1 and, thus, it requires only one iteration. However, it is obvious that computation of inverse $A^{-1}$ is extremely expensive operation and it is not an objective of any iterative methods. That example helps understand and set requirements for preconditioners, namely:

1. cheap to compute e.g. a 5-10 iterations of the corresponding Krylov solver
2. should lead to a small conditioner number of the transposed system
3. should be sparse, otherwise storage requirements will considerably increase

There exist numerous techniques to compute preconditioners given a matrix $A$ e.g. (point) Jacobi, Block-Jacobi, incomplete $LU$ decomposition (ILU), multilevel ILU (ILU(p)), threshold ILU (ILUT), incomplete Cholesky factorization (IC), sparse approximate inverse (SPAI), multigrid as a preconditioner, etc. Almost all methods listed above have some tuning parameters which allow to get a better preconditioner i.e. a smaller condition number of the transformed system. However, it usually leads to increase of computational and storage costs.

Some methods can works particularly well for matrices derived from certain PDEs e.g. Poisson, NavierStokes, etc. problems discretized using the cartesian grid. However sometimes it can take a considerable amount of time to choice right parameters for a certain preconditioning algorithm. It can become a challenge to fulfill all requirements 1, 2, 3 mentioned above.

Table

Table with comparisons of different preconditioning for our test cases

Table [] shows results of different preconditioning algorithms application to our test case. It can be seen that some algorithms failed even after tuning.

It interesting to notice that **wsmp** came to approximately the same results working on their set of matrices in their work [**wsmp**]. They observed that preconditioned iterative solvers worked efficiently only for 2 out 5 cases in contrast to direct sparse solvers.

We can summarize that it is vital to perform careful parameter tuning of any preconditioning algorithms combining results from [table] and [**wsmp**]. In general the search can take a considerable amount of time. Moreover, it becomes impractical for time integration problems where topology of an underlying problem and, as the results, the computational mesh, discretization, Jacobian matrix can be changed over time of a simulation. It is obvious that parameters chosen for a particular time step can become not optimal for consecutive steps and, at the end, it can lead to divergence. If divergence happens at any time step the entire time integration algorithm fails and the simulation has to be restarted with different preconditioning parameters or with a different preconditioning algorithm.

By and large we come to a conclusion that preconditioned iterative solvers are not robust and thus cannot fully fulfill requirements listed in section 1.

## 1.5. Direct sparse methods

Direct sparse methods combine main advantages of direct and iterative methods i.e. numerical robustness and usage of sparsity structures. As a results, there is no need for preconditioning and the computation complexity is $O(n^2)$ [**complexity-of-spdm**]. The problem is that storage cost can significantly increase during factorization i.e. the inverse of a sparse matrix can be sufficiently dense. To reduce storage space of $LU$ decomposition this group of methods performs fill-in reduction reodering as a pre-processing step before actual factorization. If storage space is still huge even after fill-in reduction reodering out-of core factorization can be used where partial results are stored in the secondary memory.

The most widely known sparse direct method is multifrontal method introduced by **mult-frontal-original:1** in their work [**mult-frontal-original:1**]. Multifrontal method is an improved extension of a frontal method [**frontal-original**] that can compute independent fronts in parallel. A front, or also called frontal matrix, can be considered as small dense matrix which is a result of Gaussian Elimination for a particular column. The algorithm, in fact, is as a variant of Gaussian Elimination process. There also exist left-looking and right-looking sparse direct methods. The difference between all of them is explained and can be found in [**elimination-tree**].

In order to understand and analyze strong scaling behavior of the algorithm we have to briefly discuss the theory of the method. For simplicity we will assume that matrix $A$ is real symmetric and $LU$ decomposition boils down to the Cholesky factorization 1.18. It allows us to focus on the Cholesky factor $L$ and its sparsity pattern only.

$$A = LDL^T \tag{1.18}$$

The algorithm usually starts with symbolic factorization to predict sparsity pattern of $L$. Once it is done the corresponding elimination tree has to be constructed.

place for figure 1.7

Figure 1.7 shows an illustrative example of a sparse matrix and its Cholesky factor from [**mult-frontal-original:2**]. The solid circles represent original non-zero elements whereas hollow ones define fill-in factors of $L$.

The elimination tree is a crucial part of the method. It can be considered as a structure of $n$ nodes that node $p$ is the parent of $j$ if and only if it satisfies equation 1.19. It is worth pointing out the definition 1.19 is not only one possible and one can define a strucutre of an elimination tree in a different way as well. As an example one can find a definition of a general assembly tree in [**mult-frontal-original:2**] proposed by **mult-frontal-original:2**.

$$A = \begin{array}{c} 1\\2\\3\\4\\5\\6\\7\\8\\9 \end{array} \begin{pmatrix} a & & & & & & \bullet & \bullet & \bullet \\ & b & & \bullet & & \bullet & & & \\ & & c & & \bullet & & & \bullet & \\ & \bullet & & d & & & & \bullet & \bullet \\ & & \bullet & & e & \bullet & & \bullet & \\ & \bullet & & & \bullet & f & & & \bullet \\ \bullet & & & & & & g & \bullet & \bullet \\ \bullet & & \bullet & \bullet & \bullet & & \bullet & h & \\ \bullet & & & \bullet & & \bullet & \bullet & & i \end{pmatrix} \qquad L = \begin{array}{c} 1\\2\\3\\4\\5\\6\\7\\8\\9 \end{array} \begin{pmatrix} a & & & & & & & & \\ & b & & & & & & & \\ & & c & & & & & & \\ & \bullet & & d & & & & & \\ & & \bullet & & e & & & & \\ & \bullet & & \circ & \bullet & f & & & \\ \bullet & & & & & & g & & \\ \bullet & & \bullet & \bullet & \bullet & \circ & \bullet & h & \\ \bullet & & & \bullet & & \bullet & \bullet & \circ & i \end{pmatrix}$$

Figure 1.7.: An example of a sparse matrix and its Cholesky factor [**mult-frontal-original:2**]

$$p = min(i > j | l_{ij} \neq 0) \tag{1.19}$$

It is important to notice that node $p$ represents elimination process of the corresponding column $p$ of matrix $A$ as well as all dependencies of column $p$ factorization on the results of its descendants.

Given definition 1.19 we can build the corresponding elimination tree as it is shown in figure 1.8.

place for figure 1.8

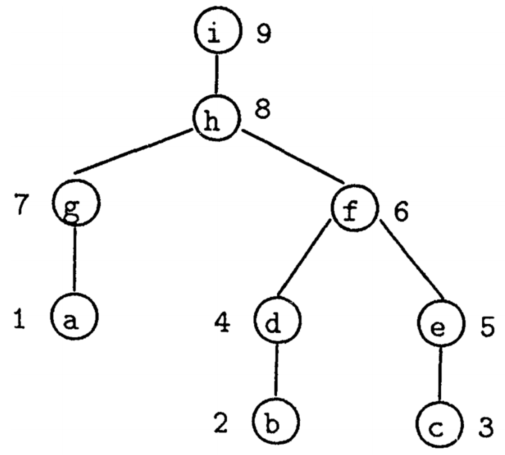


Figure 1.8.: The elimination tree for the matrix example in Figure 1.7 [**mult-frontal-original:2**]

The fundamental idea of multifrontal method spins around frontal and update matrices. A frontal matrix is used to perform Gaussian Elimination for a specific column $j$. It is a sum of a frame and update matrices as it can be seen from equation 1.20

.

$$F_j = Fr_j + \hat{U}_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} & a_{j,i_2} & \dots & a_{j,i_r} \\ a_{i_1,j} & & & & \\ a_{i_1,j} & & & & \\ \vdots & & & 0 & \\ a_{i_r,j} & & & & \end{bmatrix} + \hat{U}_j \tag{1.20}$$

where $i_0$, $i_1$, ..., $i_r$ are the row subscripts of non-zeros in $L_{*j}$ with $i_0 = j$ and $r$ is number of off-diagonal non-zero elements.

The frame matrix $Fr_j$ is filed with zeros except the first row and column. The first row and column contain non-zeros elements of the $j$th row and column of the original matrix $A$. Because we consider matrix $A$ to be symmetric the frame matrix is square and symmetric as well.

In order to describe parts of the elimination tree we will use the notation $T[j]$ to represent all descendants of the node $j$ in the tree and node $j$ itself. In this way we can define the update matrix $\hat{U}_j$ as following:

$$\hat{U}_j = - \sum_{k \in T[j]-j} \begin{bmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_1,k} \end{bmatrix} \begin{bmatrix} l_{j,k} & l_{i_1,k} & \dots & l_{i_1,k} \end{bmatrix} \tag{1.21}$$

The update matrix $\hat{U}_j$ is, in fact, can be considered as the second term of the Schur complement i.e. update contributions from already factorized columns of $A$.

The subscript $k$ represents descendant columns of node $j$. Thus we include and consider only those elements of descendant columns which correspond to the non-zero pattern of the $j$th column that we are currently factorizing.

Let's consider the partial factorization of 2-by-2 block dense matrix to better understand essence of update matrix $\hat{U}_j$.

$$A = \begin{bmatrix} B & V^T \\ V & C \end{bmatrix} = \begin{bmatrix} L_B & 0 \\ VL_B^{-T} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & C - VB^{-1}V^T \end{bmatrix} \begin{bmatrix} L_B^T & L_B^{-1}V^T \\ 0 & I \end{bmatrix} \tag{1.22}$$

Again we assume that $B$ has already been factorized and can be expressed as:

$$B = L_B L_B^T \tag{1.23}$$

The Schur complement from equation 1.22 can be viewed as the original sub-matrix $C$ and update $-VB^{-1}V^T$. It can be written in a vector form as well:

$$-VB^{-1}V^T = -(VL_B^{-T})(L_B^{-1}V^T) = -\sum_{k=1}^{j-1} \begin{bmatrix} l_{j,k} \\ \vdots \\ l_{n,k} \end{bmatrix} \begin{bmatrix} l_{j,k} & \dots & l_{n,k} \end{bmatrix} \tag{1.24}$$

As it can be easily seen that equations 1.24 and 1.21 are identical. The difference is that equation 1.21 exploits sparsity of the corresponding row and column of $L$ and thus masks unnecessary information.

We can also notice from equation 1.22 that the frame matrix $Fr_j$ corresponds to the block matrix $C$ and brings information from the original matrix $A$ whereas matrix $\hat{U}_j$ adds information about the columns that have already been factorized.

As soon as the frontal matrix $F_j$ is assembled i.e. we have the complete update of column $j$, we can perform elimination of the first column and get non-zero entries of factor column $L_{*j}$.

Let's denote $\hat{F}_j$ as a result of the first column factorization of the frontal matrix $F_j$. Then we can express the results as following:

$$\hat{F}_j = \begin{bmatrix} l_{j,j} & \dots & 0 \\ \vdots & I & \\ l_{i_r,j} & & \end{bmatrix} \begin{bmatrix} 1 & \dots & 0 \\ \vdots & U_j & \\ 0 & & \end{bmatrix} \begin{bmatrix} l_{j,j} & \dots & l_{i_r,j} \\ \vdots & I & \\ 0 & & \end{bmatrix} \tag{1.25}$$

where sub-matrix $U_j$ represents the full update from all descendants of node $j$ and node $j$ itself. Equation 1.26 express the sub-matrix $U_j$ in a vector form.

$$\hat{U}_j = -\sum_{k \in T[j]} \begin{bmatrix} l_{i_1,k} \\ \vdots \\ l_{i_1,k} \end{bmatrix} \begin{bmatrix} l_{i_1,k} & \dots & l_{i_1,k} \end{bmatrix} \tag{1.26}$$

Together with the frontal $F_j$ and update $\hat{U}_j$ matrices, the update column matrix $U_j$ (also called contribution matrices) forms the key concepts of the multifrontal method. To consider the importance of sub-matrix $U_j$ let's consider and example illustrated in Figure 1.9.

place for figure 1.9

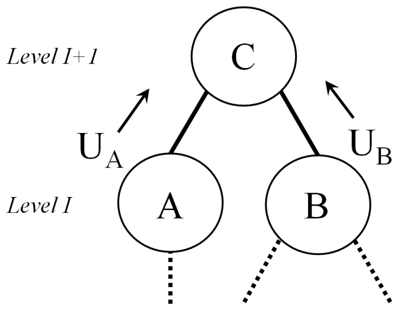


Figure 1.9.: Information flow of the multifrontal method

We assume that factorization of columns A and B have already been done and corresponding contribution matrices $U_A$ and $U_B$ have been computed. From equation 1.26 we have already known that both $U_A$ and $U_B$ contain the full updates of all their descendants including updates from factorization of columns $A$ and $B$ as well. Therefore update column matrices $U_A$ and $U_B$ have already got all necessary information to construct update matrix $\hat{U}_C$. The detailed proof and careful explanation can be found in [**mult-frontal-original:2**].

It might happen that we do not need all rows and columns of $U_A$ and $U_B$ i.e. we need only some subset of them, because of sparsity of column $C$ . It is also important to place all necessary rows and columns of matrices $U_A$ and $U_B$ in a right place within matrix $\hat{U}_C$. For that reason, an additional matrix operation, called ***extend-add***, must be introduced.

Let's consider an example from [**mult-frontal-original:2**] of an extend-add operation for 2-by-2 matrices $R$ and $S$ which correspond to the indices $5, 8$ and $5, 9$ of some matrix $B$, respectively.

$$R = \begin{bmatrix} p & q \\ u & v \end{bmatrix}, S = \begin{bmatrix} w & x \\ y & z \end{bmatrix} \tag{1.27}$$

The result of the operation is going to be a 3-by-3 $K$ matrix which looks as following:

$$K = R \text{✥} S = \begin{bmatrix} p & q & 0 \\ u & v & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} w & 0 & x \\ 0 & 0 & 0 \\ y & 0 & z \end{bmatrix} = \begin{bmatrix} p+w & q & x \\ u & v & 0 \\ y & 0 & z \end{bmatrix} \tag{1.28}$$

Hence we can express formation of the frontal matrix $F_j$ using the extend-add operation and all direct children of node $j$ in the following way:

$$F_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} & a_{j,i_2} & \dots & a_{j,i_r} \\ a_{i_1,j} & & & & \\ a_{i_1,j} & & & & \\ \vdots & & & 0 & \\ a_{i_r,j} & & & & \end{bmatrix} \text{✥} U_{c_1} \text{✥} \dots \text{✥} U_{c_s} \tag{1.29}$$

where $c_1$, $c_2$, ... $c_n$ are indices of direct children of the node $j$.

Now it can be clearly seen that the resultant frontal matrix $F_j$ is a small dense one and it can be efficiently computed using BLAS level 3 subroutines.

After factorization we have to build the contribution matrix $U_j$ i.e. add columns and rows of $U_{c_1}, , U_{c_2}, \dots, U_{c_s}$ to $U_j$ that have not been used in factorization of $F_j$ due to sparsity of column $j$. After that we can continue to move up along the tree. The complete update matrices grow in size as we move to the top of the tree. Therefore they have to be stored in a sparse matrix format to stay within memory constrains of the computer.

Another important aspect is storage and manipulation of frontal and contribution matrices. Sometimes we have to store contribution matrices produced in previous steps into some temporary buffer and efficiently retrieve them later during factorization. This can require some matrix re-ordering. In case of symmetric matrices, one can apply postordering on a tree to be able to use the stack data structure to alleviate the process of contribution matrix manipulations during factorization. A tree postordering is based on topological ordering and it has been proven that it is equivalent to the original matrix ordering and thus leads to the same filled graph [**mult-frontal-original:2**]. We refer to the original matrix ordering as the ordering received from fill-in reduction operation.

sentence refactoring

A tree postordering means that a node is ordered before its parent and, additionally, nodes in each subtree are numbered consecutively. Figure 1.10 shows an example of posrordering applied to the elimination tree of the matrix from figure 1.7. The results of this can be see in figure 1.11 where consecutive *push* and *pop* operations are efficiently used during factorization and thus simplify the program logic.

place for figure 1.10

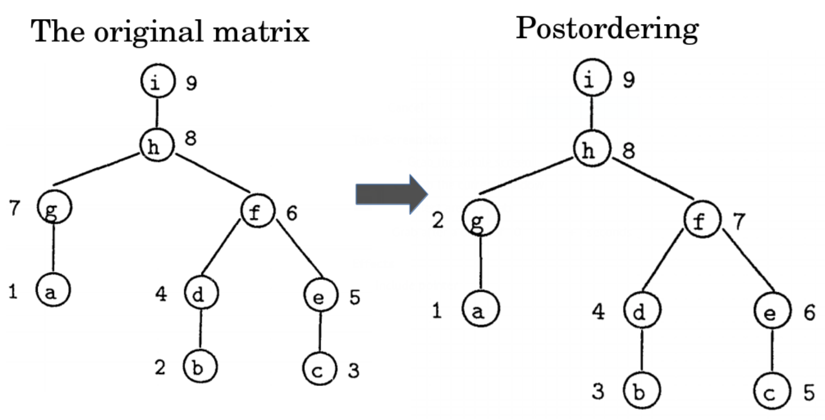


Figure 1.10.: An example of matrix postordering from [**mult-frontal-original:2**]

place for figure 1.11

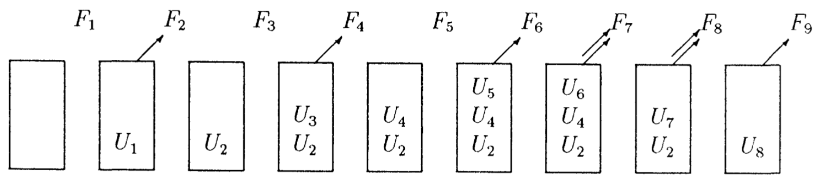


Figure 1.11.: The stack contents for the postordering [**mult-frontal-original:2**]

We can see the algorithm requires to perform some preprocessing steps in order to estimate the size of working space for matrix manipulations. If the working space has not been predicted correctly the algorithm will terminate during factorization. Additionally it can happen that even with the correct estimation we can be run out of space in the main memory, in case of huge sparse matrices. This fact can require to use the secondary memory and, as a result, the execution time will increase significantly. Therefore, different optimal postordering schemes have been proposed which allow to shrink the amount of space needed during factorization [**mm:optimal-tree-postordering**] [**mm:elimination-tree-rotations**]. Some schemes, for example elimination tree rotations [**mm:elimination-tree-rotations**], can lead to deep and unbalanced trees which might have their negative effect on task parallelism as we will see later.

In general, the estimation of working space can be tricky due to pivoting. Because pivoting happens only during the numerical factorization it is not always possible to estimate enough space correctly beforehand. There exist some heuristics which allow to use some numerical matrix information during symbolic factorization to better predict the amount of required space [**wsmp:direct-solution-of-general-system**].

It can be clearly observed the method consists of three distinct phases, namely: analysis, numerical factorization and solution. The analysis phase includes all pre-processing steps that have been discussed above i.e. fill-in reduction, postordering, symbolic factorization, building elimination tree an so on. During the numerical factorization phase the $L$ and $D$ (or $U$) parts of a matrix $A$ are computed based on sequence of dense factorization on frontal matrices. At the solution step, the solution vector $x$ is computed by means of backward and forward substitutions (equations 1.2 and 1.3).

In practice, an improved version of multifrontal method, called supernodal method, is used. The idea of the supernodal method is to shrink the final elimination tree by grouping some particular nodes/columns in one computational unit. As a result, more useful floating point operations per memory access can be performed by eliminating few columns at once within the same frontal matrix.

A supernode is formed by a set of contiguous columns with identical off-diagonal sparsity structure forms. Thus, a supernode has few important properties. Firstly, it can be expressed as a set of indices, namely: $\{j, j+1, \ldots, j+t\}$, where node $j+k$ is parent of $j+k-1$ in the elimination tree. Secondly, the size of the supernodal frontal matrix is equal to the frontal matrix of the $j$th column within a supernode. As an example, Figure 1.12 shows a postordered matrix $A$ and its Cholesky factor $L$ as well as the corresponding supernodal elimination tree.

place for figure 1.12

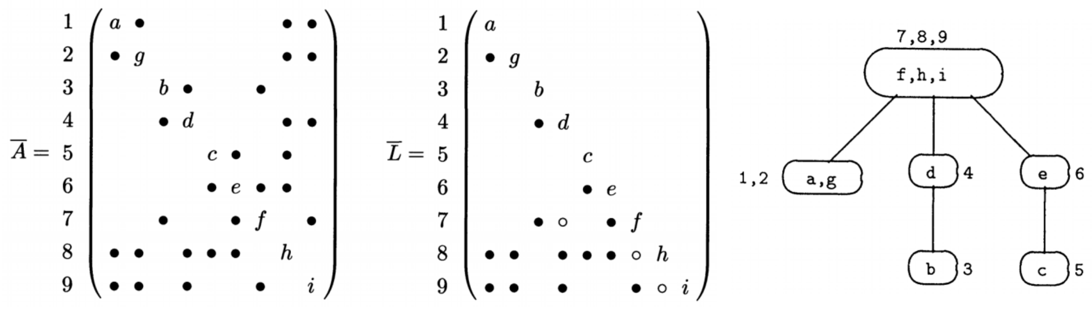


Figure 1.12.: An example of a supernodal elimination tree [**mult-frontal-original:2**]

Equation 1.30 expresses the building process of a frontal matrix of a supernode. In contrast to 1.29, the frame matrix $\mathcal{F}_j$ contains more dense rows and columns. As before, we use *extend-add* operation to get the full block update from children contribution matrices.

check grammar

It should be mentioned there exist more sophisticated variants of supernodes. Most of the time, it intends to improve efficiency of the algorithm. **mult-frontal-original:2** pointed out that supernodes could be defined without using the contiguous constrains [**mult-frontal-original:2**]. On another hand, **complexity-of-spdm** defines supernodes corresponded to separators from the nested dissection step [**complexity-of-spdm**] which was used for fill-in reduction.

$$\mathcal{F}_j = \begin{bmatrix} a_{j,j} & a_{j,j+1} & \dots & a_{j,j+t} & a_{j,i_1} & \dots & a_{j,i_r} \\ a_{j+1,j} & a_{j+1,j+1} & \dots & a_{j+1,j+t} & a_{j+1,i_1} & \dots & a_{j+1,i_r} \\ \vdots & \vdots & \dots & \vdots & & & \\ a_{j+t,j} & a_{j+t,j+1} & \dots & a_{j+t,j+t} & a_{j+t,i_1} & \dots & a_{j+t,i_r} \\ a_{i_1,j} & a_{i_1,j+1} & \dots & a_{i_1,j+t} & & & \\ \vdots & \vdots & \dots & \vdots & & 0 & \\ a_{i_r,j} & a_{i_r,j+1} & \dots & a_{i_r,j+t} & & & \end{bmatrix} \text{✣} U_{c_1} \text{✣} \dots \text{✣} U_{c_s} \quad (1.30)$$

Up to this point we have already seen all key concepts of the multifrontal method and discussed how the algorithm works. We will move to the discussion of parallelization of the method.

The elimination tree, in fact, represents dependencies among columns. Conversely, the tree also shows independent steps of elimination process. Hence the tree forms independent problems that can be executed in parallel. Task parallelism is the main and primary source the algorithm parallelisation. Figure 1.13 shows task parallelism, for the example given in Figure 1.12, where each color represents a set concurrent tasks.

place for figure 1.13

For example, nodes on separate branches of the tree are totally independent and can processed in parallel. However, as soon as at least two branches run into the same node it forms a dependency and we have to wait all contribution matrices of its children and cannot proceed further.

We can observe the amount of task parallelism is rapidly decreasing while moving towards the root along the tree. Once we reach the root of the tree the algorithm becomes totally sequential. This fact can play the significant role in strong scaling behavior of the method.

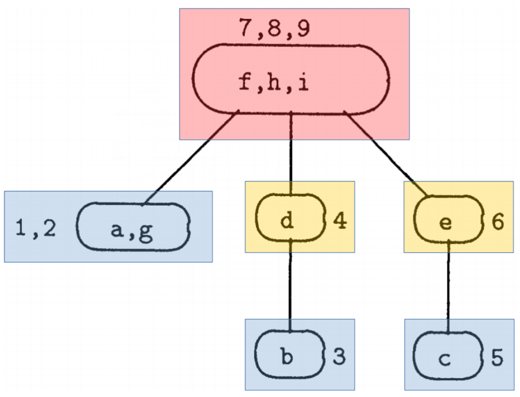
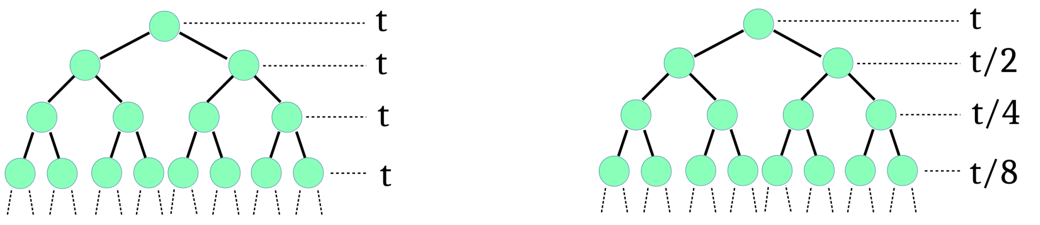


Figure 1.13.: Parallel steps of the multifrontal method based on the example in Figures 1.12



(a) Model 1: equal cost per level (b) Model 2: quadratically decreasing cost per level

Figure 1.14.: Simple parallel models of the multifrontal method

We developed two simple models based on perfectly balanced binary trees to better understand strong scaling of the algorithm. The main concept of the models is so-called cost per level or cost per node. This idea is similar to the recursion trees in [**recursion-tree**] which explains and computes complexity of recurrent algorithms.

Figure 1.14a represents the first model where we keep the same cost per level whereas the second model (Figure 1.14b) simulates quadratic cost decay from level to level. Additionally we assume that computational cost distributed uniformly between nodes at the same level for both models.

We have to say that our models mimic only numerical factorization and do not include time spent on any per-processing steps, for example, fill-in reduction reodering. A cost per level can be interpreted in different ways e.g. increase of partial factorization

place for figure 1.14

sentence refactoring

time due to growth of frontal matrices in size, time increase spent on numerical pivoting, increase of MPI communication overheads due to growth of contribution matrices, etc. It should be mentioned that real computer implementations of the multifrontal algorithm (MUMPS, SuperLU, etc.) are quite sophisticated in many aspects and our models do not have any intention to analyze performance of a particular package. Instead the objective of these models is to show possible strong scaling behavior and possible bottlenecks.

We will consider only task parallelism at the beginning to a first approximation and later we will discuss how additional data parallelism can affect algorithm performance.

Instead of coloring given in Figure 1.13, we assume that each level has the same color and thus can be executed fully in parallel if we have enough processing elements. We cannot go to the next level till the current one has not been completed yet i.e. free processing elements, that do not have nodes to execute at the current level, have to wait.

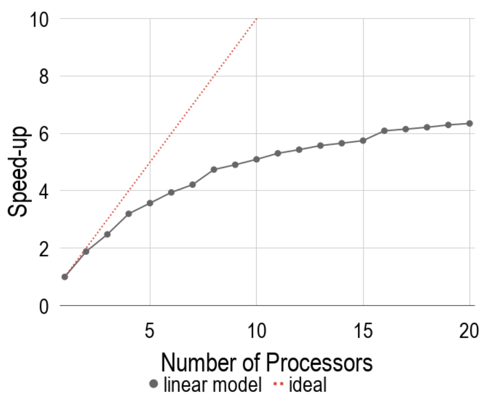
As we mentioned above the root of the tree can be processed purely sequentially if we only consider task parallelism. As a first approximation, time spent on the root factorization determines the minimal execution time according to the Amdahl's low [**wiki:amdahls-low**]. More precisely, the minimal execution time is equal to a sum of time spent on single node partial factorization at each level. This time determines the asymptote on the corresponding speed-up graph.

We considered a perfectly balanced tree with 16 levels, 65535 nodes and the maximum of 20 processing elements as an example. The numerical results of linear and quadratic models can be viewed in Figures 1.14a and 1.14b, respectively. The figures show a rapid drop of performance, especially in case of the quadratic model. Table 1.4 demonstrates the maximum potential speed-up, having 32768 processing elements which is equal to the number of leaves of the tree, against the speed-up we have got using only 20 processing elements.

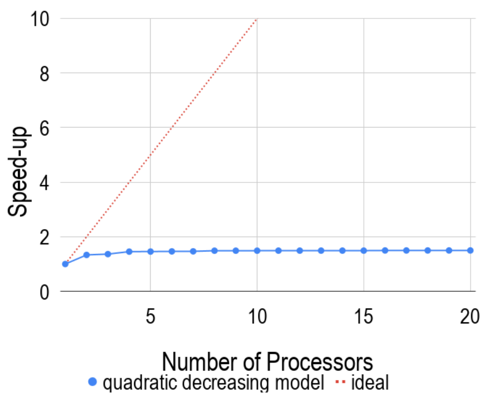
place for figure 1.15

|  | 20 PEs | 32768 PEs |
|---|---|---|
| Model 1 | 6.3492 | 8.0000 |
| Model 2 | 1.4972 | 1.5000 |

Table 1.4.: Potential speed-up of linear and quadratic models



(a) Theoretical speed-up of Model 1



(b) Theoretical speed-up of Model 2

Figure 1.15.: Theoretical speed-up

We can see that model 1 still has some potential to grow whereas the second model has already reached its asymptote and further increase of processing elements does not make sense. In spite of a potential growth of the first model, both models have very low parallel efficiency even with 20 processing elements which can be observed from table 1.5.

| | 20 PEs |
|---|---|
| Model 1 | 0.3175 |
| Model 2 | 0.0749 |

Table 1.5.: Efficiency of linear and quadratic models using 20 PEs

Both models shows that computational intensity per node grows from bottom to top. It is easy to conclude from Figure 1.14 that intensity per node is equal $t/2^i$ and $t/2^{2i}$ for the first and second models, respectively (where $i$ is a level of the tree). It reflects that the most intensive part of the method is centered on the top part of the tree i.e. first few level. **mult-frontal-original:2** discussed application of the multifrontal method to a $k-by-k$ regular model problem with nine-point difference operator in his paper [**mult-frontal-original:2**]. He observed that factorization of the last 6 nodes took slightly more than 25% of the total amount of arithmetical operations. As a comparison, table 1.6 shows fractions of time spent on processing first few top levels of our models: 1 and 2.

| | Model 1 | Model 2 |
|---|---|---|
| Level 0 | 6.25% | 50.00% |
| Level 1 | 12.50% | 75.00% |
| Level 2 | 18.75% | 87.50% |

Table 1.6.: Distribution workload per level in case model 1 and 2

As we can see, the result of our first model is relatively close to 25% and, therefore, it looks quite optimistic. However, the second model shows that 87% of workload is focused on the top part of the tree and, as a result, we can consider that model as extremely pessimistic.

By and large, reduction of time spent on the top nodes is a way to improve strong scaling behavior. To do so, data parallelism can be additionally exploited for these nodes. It is worth noting that data parallelism at bottom levels does not make sense because it leads to increase of granularity there and thus increase communication overheads which can lead to significant performance drop.

Figure 1.16 shows an example of two types of parallelism applied to the algorithm. First of all, we can see the leaves are grouped in subtrees and a single PE is assigned to each subtree. Other nodes are distributed among three different types. Nodes of the first type uses task parallelism only, which is induced by the tree, and each node is executed in a single processor. The second type exploits data parallelism with 1D block row distribution among the processors. The root belongs to the third type where data parallelism is used with 2D block cyclic distribution. The details of MUMPS parallelism management is carefully explained and can be found in [**mumps:task-data-parallelism**].

place for figure 1.16

All the techniques mentioned above were designed to improve strong scaling behavior by splitting the most intensive parts among all available processors. Going back to our models, we can also think about that in a slightly different way, namely: *data parallelism helps to re-distribute cost per node/level on the corresponding elimination tree*. However, we have to notice that efficiency of data parallelism totally depends on sizes of frontal matrices at the top part of the tree. In case of skinny sparse matrices, oversubscription of processing elements can lead to strong performance penalties as we could see from section 1.3. A machine-dependent minimal frontal matrix size was introduced in MUMPS in order to control whether to use ScaLAPACK at the root node or not [**mumps-manual**]. It can happen that the algorithm uses only task parallelism, due to the threshold, and, as a results, scaling will only depend on the tree structure

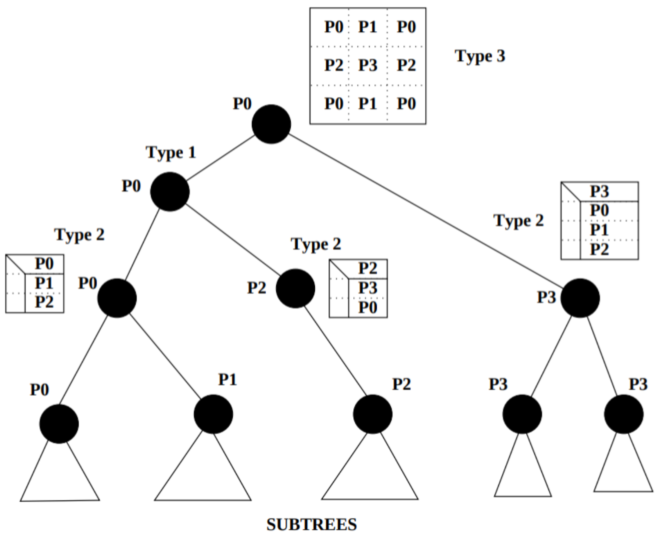


Figure 1.16.: MUMPS parallelism management in case of 4 PEs [**mumps:task-data-parallelism**]

that can be deep and unbalanced.

Figure 1.17 shows comparison of strong scaling between model 1 and parallel numerical factorization of the matrix ***memchip*** (Table 1.2) done with using MUMPS library. The sparsity pattern before and after fill-in reduction is shown in figure 1.18.

As we can see, our model and the experiment show the same trend and the results are pretty much close to each other. However, our model takes into consideration only task parallelism whereas MUMPS exploits both data and task parallelism. Additionally, we have to mention that our model 1 works well only for relatively big sparse matrices. It can be quite inaccurate in case of small/skinny sparse systems.

add some examples to the appendix

place for figure 1.17. Show standard deviation

place for figure 1.18

In general, it is possible to refine our models and make them more accurate, using Bulk Synchronous Parallel (BSP) approach, for example. However, it will require to possess a real postordered elimination tree, extracted from a specific implementation of multifrontal method, together with information about supernodes sizes. We strongly believe the new enhanced model can explain the jagged strong scaling behavior of the MUMPS solver that we can observe in figure 1.17. But, this approach seems to be quite cumbersome and requires to delve into the source code of a particular library. It is needless to say that data can be retrieved only during run time and only after the analysis phase. This makes it less valuable and we can see that it is definitely a wrong

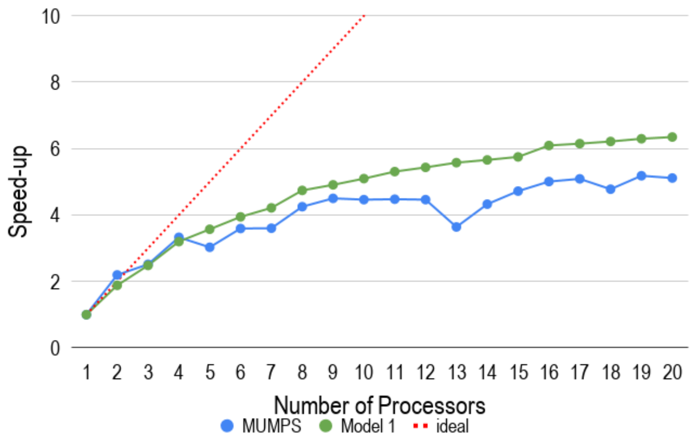


Figure 1.17.: Comparison between model 1 and numerical factorization of the matrix ***memchip*** using MUMPS library

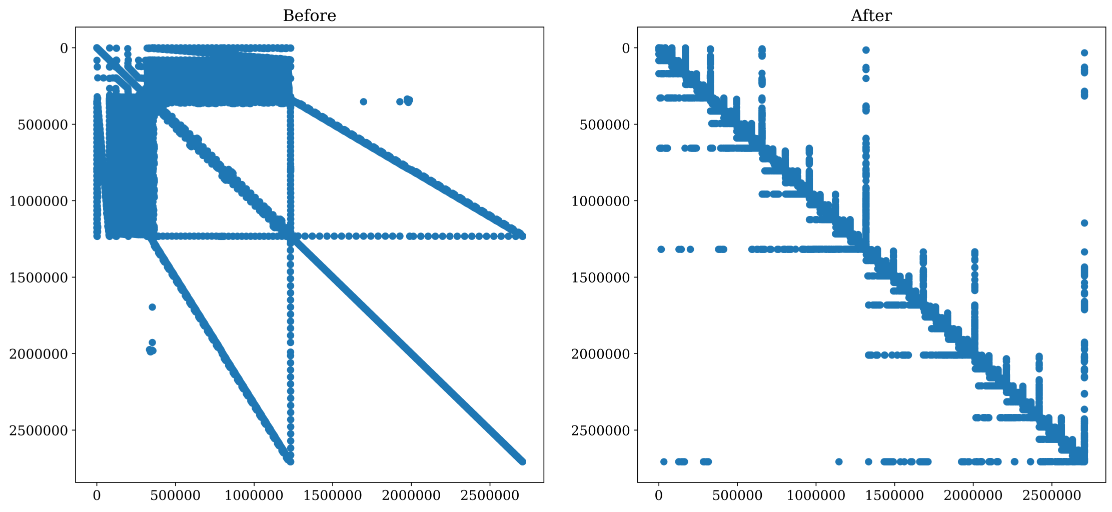


Figure 1.18.: Sparsity structure of the matrix ***memchip*** before and after fill-in reduction

way to go.

There are few important aspects to discuss at the end of the section. Numerical robustness is the main advantage of the multifrontal method. It does not require any preconditioner to solve a system of equations. As we discussed in the previous section, tuning a specific preconditioning algorithm can take a considerable amount of time, especially in case of our systems. As another advantage, the method (heavily) exploits matrix sparsity which lowers computational complexity up to $O(n^2)$. In case of massively huge matrices, the algorithm can utilize the secondary memory which sometimes is only one way to solve a system.

We can conclude, from the analysis above, the method has inherently bad scaling behavior and it is quite sensitive to a matrix structure. We will see later that it is almost impossible to predict the saturation point i.e. a point after which performance either drops or stays at the same level. We assume that scaling becomes better with growth of a matrix size. However, we cannot expect such behavior for small and medium systems.

Secondly, we can see the algorithm requires many pre-processing steps to be done before numerical factorization phase. All these steps must run in parallel and be highly scalable. Apart from performance constrains of the steps, they must lead to wide and well balanced elimination trees which becomes crucial during the numerical phase.

Lastly, the algorithm can fail due to incorrect working space prediction. As a results, factorization has to be restarted with some modification of input solver parameters.

## 1.6. Hybrid method

We have observed almost all available methods and we could see that none of them can fully cover all our requirements at once, namely:

- robustness
- numerical stability
- parallel efficiency
- open source licenses

The analysis from sections 1.4 and 1.4 shows that iterative methods scale much better in contrast to sparse direct ones. However, they are only efficient in case of very well preconditioned systems. We showed in section 1.4 that search of preconditioning parameters usually takes lots of time and efforts. Additionally, we cannot guarantee that the settings fond for our GRS matrix set will always work well in subsequent steps of time integration or for other different simulations.

Sparse direct methods do not have such a problem. They always produce the right solution. The methods can only fail in case of underestimation of the working space due to numerical pivoting during the numerical factorization phase. In order to cope with that problems, some implementations of direct sparse methods provide two options to the user, namely: to increase predicted working space by some factor e.g. 2, 3, 4, etc. or to lower constrains of numerical pivoting which allows small numerical values to stay on the diagonal.

The drawback of the second option is that it can lead to out-of-core execution with using the secondary memory which makes numerical factorization significantly slow. While the second option has lower chance of out-of-core factorization it can lead to a numerically inaccurate solution.

After many considerations we decided to stick to the sparse direct solvers because **robustness** criteria had the highest priority in our case. To circumvent problems mentioned above, we proposed a so-called hybrid solver, in spite of the fact that the definition of *hybrid linear solvers* had already been used in scientific computing literature in a slightly different way [**shylu-hybrid-solver**]. ***The idea is to switch off numerical pivoting (or significantly lower the constrains) of sparse direct solvers and use the resultant* LU *decomposition as a preconditioner for an iterative method, for example GMRES*.**

```
# compute LU decomposition with a sparse direct solver
LU = SparseDirectSolver(matrix=A, pivoting="switch_off")

# compute inverse of A using backward-forward substitutions
# sovle: LU * A_inv = I
A_inv = ComputeInverse(decomposition=LU)

# apply a Krylov method to a preconditioned system
# i.e A_inv * A * x = A_inv * b
GMRES(matrix=A, rhs=b, preconditione=A_inv)
```

Listing 1.1: A pseudo-code of the Hybrid approach

According to our primary tests, the hybrid approach showed us that it required from

write which solver we used

1 to 5 iterations of the GMRES solver on average to converge to a desired residual.

The main problem of our approach is parallel efficiency because sparse *LU* decomposition takes the most of computational time. We discussed reasons of possible bad strong scaling behavior of sparse direct solvers in section 1.5. We could see, in case of the multifrontal method, these methods consist of multiple steps and implementation of each step has its strong effect on parallel performance. We also mentioned that the main source of performance improvement is data parallelism and it can be achieved in many different ways. Hence, performance of the same method can vary form library to library.

In the next section we are going to investigate all available open-source implementations of sparse direct solvers, compare their efficiency and choose one of them. At the beginning, we will only consider libraries that have their direct interface to PETSc [**petsc-web-page**]. PETSc is a scientific numerical library that contains various algorithms and methods, especially the Krylov methods. It is highly efficient in parallel and provides numerous interfaces to other libraries such as MUMPS, SuperLU, Hypre, PaStiX, ViennaCL and etc.

proofreading

The subsequent sections will be dedicated to tuning and optimization of a specific library with the aim to reduce execution time.

## 1.7. Choice of a Sparse Direct Solver Library

Fair to say, there is no single algorithm or software that is best for all types of linear systems [**list-of-sparse-direct-solvers**]. Nowadays there exist many different sparse direct solvers on the market. Some of them are tunned for specific linear systems i.e. symmetric positive definite, systems with symmetric sparsity pattern, system with complex numbers, etc., some are targeted for the most general cases. Some packages can handle data parallelism in different ways even within the same library depending on the system size and other criteria. Hence, parallel performance highly depends on a specific implementation of a method. Table 1.7 displays a short summary of almost all available and well-known packages, at the time of writing, in this field based on works [**list-of-sparse-direct-solvers**] and [**petsc-web-page**].

add footnotes of Open*

We only listed libraries that can run on distributed memory parallel machines. Almost all of them also support shared memory environment in some degree. Nonetheless there also exist libraries that run either only sequentially (UMFPACK, SPARSE, TAUCS,

| Package | Method | Matrix Types | PETSc Interface | License |
|---|---|---|---|---|
| Clique | Multifrontal | Symmetric | Not Officially | Open |
| MF2 | Multifrontal | Symmetric pattern | No | - |
| DSCPACK | Multifrontal | SPD | No | Open* |
| MUMPS | Multifrontal | General | Yes | Open |
| PaStiX | Left looking | General | Yes | Open |
| PSPASES | Multifrontal | SPD | No | Open* |
| SPOOLES | Left-looking | Symmetric pattern | No | Open* |
| SuperLU_DIST | Right-looking | General | Yes | Open |
| symPACK | Left-Right looking | SPD | No | Open |
| S+ | Right-lookin | General | No | - |
| PARDISO | Multifrontal | General | No | Commercial |
| WSMP | Multifrontal | General | No | Commercial |

Table 1.7.: List of packages to solve sparse linear systems using direct methods on distributed memory parallel machines [**list-of-sparse-direct-solvers**], [**petsc-web-page**]

SuperLU) or only on shared memory machines (PanelLLT, SuperLU_MT).

We can see, from table 1.7, that only MUMPS, PaStiX and SuperLU_DIST cover all our initial requirements: open-source license and direct interface to the PETSc library. However, these libraries implement different sparse direct methods, namely: multifrontal, left-looking and right-locking, respectively. Moreover, they handle partial pivoting in different ways.

It is known that partial pivoting is necessary to achieve a good numerical accuracy during Gaussian Elimination. It interchanges rows and columns of a matrix in such a way to avoid small numerical values along the diagonal. In case of sparse direct solvers, the numerical pivoting, in run-time, usually distorts all predictions that have been made during the analysis phase and can lead to significant fill-in and load unbalanced, with respect to floating-point operations, during factorization. Hence, implementation of numerical pivoting, especially during parallel execution, plays the most important role in performance for this group of methods.

Both PaStiX and SuperLU_DIST libraries use so-called static pivoting where the pivot order is chosen before numerical factorization and kept fixed during factorization.

The main advantage of static pivoting is that it allows to better optimize the data layout, load balance, and communication scheduling [**superlu-manual**]. However, it leads to a higher risk of numeric instability. Therefore, both PaStiX and SuperLU_DIST provide a few ways to perform the solution refinement.

For instance, SuperLU_DIST uses diagonal scaling, setting very tiny pivots to larger values, and iterative refinement (listing 1.3). While PaStiX allows the user to choose a refinement strategy between GMRES, CG (for SPD systems) and iterative refinement as well. At this point it is interesting to notice that we came to the same conclusion as the PasTiX developers with respect to the solution refinement using Krylov iterative methods.

Iterative refinement, shown in listing 1.3, is claimed to converge to a decent precision within 2 or 3 steps in work [**mm-backward-error**]. However, in practice, we noticed that the iterative refinement can work not as expected, especially in case of lowered partial pivoting constrains.

can we use decent here?

For completeness, we have to mention that the variable *too_large* is, in fact, an estimation of the backward error [**mm-backward-error**] which can be expressed as following:

$$\frac{|b - A\hat{x}|_i}{(|b| + |A||\hat{x}|)_i} \tag{1.31}$$

where $\hat{x}$ is the computed solution and $| \bullet |$ is the element-wise module operation.

```
# perform analysis and numerical factorization
# phases
LU = SparseDirectSolver(matrix=A)

# compute initial solution
x = Solve(factorization=LU, rhs=b)

# compute initial residual
r = A * x − b

while r > too_large:
  # find correction
  d = Solve(factorization=LU, rhs=r)

  # update solution
  x = x − d

  # update residual
  r = A * x − b
```

Listing 1.2: A simple iterative refinement

listing of iterative refinement

In contrast to PaStiX and SuperLU_DIST, MUMPS performs partial pivoting in run-time during the numerical factorization phase. To limit the amount of numerical pivoting, and stick better to the sparsity predictions done during the symbolic factorization, partial pivoting can be relaxed, leading to the partial threshold pivoting strategy [**mumps-manual**].

A pivot $|a_{i,i}|$ is accepted if it satisfies:

$$|a_{i,i}| \geq u \times max_{k=i \cdots n} |a_{k,j}| \tag{1.32}$$

where $u$ is value between 0 and 1.

To improve solution accuracy, MUMPS, as PasTiX and SuperLU_DIST, provides the iterative refinement as a post-processing step as well.

The most important feature which MUMPS introduces is so-called delayed pivots. It can happen that equation 1.32 cannot be satisfied within a fully-summed block of a frontal matrix (equation 1.30) and we also cannot consider elements outside the block since the corresponding rows are not fully-summed. In this case, some rows and columns will remain unfactored, or delayed, in the front. They are going to be sent the frontal matrix of the parent, as part of the contribution block and the process will repeat. The delayed pivot approach helps to improve numerical accuracy, however, it causes additional fill-in in the parent node.

In spite of obvious complexity of dynamic partial pivoting, MUMPS allows the user to explicitly control run-time behavior of the algorithm due to partial threshold pivoting strategy. This provides an opportunity for optimization and tuning in some degree.

data parallelism difference between MUMPS and SuperLU

PETSc (version 3.10) provides the full interface to both SuperLU_DIST and MUMPS, whereas the interface to the PasTiX library is quite limited. In fact, in case of PasTiX, the user can only control the number of threads per MPI process and a level of verbosity, which makes this library to be less interesting for our subsequent (following) research.

In order to evaluate the overall parallel performance of the libraries, we performed a few flat-MPI tests with the GRS matrix using the HW1 machine. Before testing, we downloaded and configured the libraries, MUMPS version 5.1.2, PasTiX version 6.0.0, SuperLU_DIST version 5.4, within the PETSc environment with their **default settings**. As a profiling tool, we used the internal PETSc profiler. A time limit of 15 minutes was set up for each test case to prevent blocking of a compute node in case if out-of-core execution. Results of the tests are summarized in tables 1.8, 1.9, 1.10 and in appendix A. Numerical values in tables are given in **seconds**.

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 1 | 7.02E-02 | 8.72E-02 | 3.17E+00 |
| 2 | 6.73E-02 | 7.10E-02 | 1.43E+00 |
| 3 | 6.36E-02 | 7.01E-02 | 1.07E+00 |
| 4 | 6.28E-02 | 7.11E-02 | 8.17E-01 |
| 5 | 6.50E-02 | 7.15E-02 | 7.51E-01 |
| 6 | 6.72E-02 | 7.62E-02 | 6.15E-01 |
| 7 | 6.91E-02 | 7.69E-02 | 6.48E-01 |
| 8 | 6.89E-02 | 8.17E-02 | 5.41E-01 |
| 9 | 7.50E-02 | 8.28E-02 | 5.02E-01 |
| 10 | 7.22E-02 | 8.52E-02 | 4.64E-01 |
| 11 | 7.55E-02 | 8.89E-02 | 5.82E-01 |
| 12 | 7.61E-02 | 1.06E-01 | 4.37E-01 |
| 13 | 7.84E-02 | 9.72E-02 | 5.43E-01 |
| 14 | 8.06E-02 | 1.02E-01 | 4.22E-01 |
| 15 | 8.20E-02 | 1.19E-01 | 3.91E-01 |
| 16 | 8.07E-02 | 1.19E-01 | 4.44E-01 |
| 17 | 8.38E-02 | 1.22E-01 | 5.19E-01 |
| 18 | 8.40E-02 | 1.26E-01 | 3.77E-01 |
| 19 | 8.58E-02 | 1.33E-01 | 5.47E-01 |
| 20 | 8.64E-02 | 1.49E-01 | 3.39E-01 |

Table 1.8.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **cube-5** (9352 equations)

We ran into a few problems with the SuperLU_DIST library during the tests. Firstly, factorization exceeded the set time limit in case of **cube-64** and **k3-2** matrices. Secondly,

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 1 | 1.36E+00 | 1.39E+00 | time-out |
| 2 | 1.00E+00 | 9.82E-01 | time-out |
| 3 | 8.83E-01 | 1.06E+00 | time-out |
| 4 | 8.17E-01 | 8.74E-01 | time-out |
| 5 | 7.85E-01 | 8.50E-01 | time-out |
| 6 | 8.06E-01 | 8.52E-01 | time-out |
| 7 | 7.71E-01 | 8.33E-01 | time-out |
| 8 | 7.66E-01 | 8.33E-01 | time-out |
| 9 | 7.93E-01 | 8.35E-01 | time-out |
| 10 | 8.07E-01 | 8.15E-01 | time-out |
| 11 | 7.75E-01 | 8.15E-01 | time-out |
| 12 | 7.81E-01 | 8.10E-01 | time-out |
| 13 | 7.85E-01 | 8.35E-01 | time-out |
| 14 | 7.85E-01 | 8.18E-01 | time-out |
| 15 | 7.88E-01 | 8.46E-01 | time-out |
| 16 | 7.81E-01 | 8.23E-01 | time-out |
| 17 | 6.83E-01 | 8.49E-01 | time-out |
| 18 | 7.96E-01 | 8.44E-01 | time-out |
| 19 | 8.04E-01 | 8.65E-01 | time-out |
| 20 | 6.85E-01 | 8.87E-01 | time-out |

Table 1.9.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **cube-64** (100657 equations)

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 1 | 1.55E+02 | 6.44E+01 | crashed |
| 2 | 6.28E+01 | 4.84E+01 | crashed |
| 3 | 5.06E+01 | 5.02E+01 | crashed |
| 4 | 4.17E+01 | 4.50E+01 | crashed |
| 5 | 2.52E+01 | 3.98E+01 | crashed |
| 6 | 2.58E+01 | 4.29E+01 | crashed |
| 7 | 2.65E+01 | 4.30E+01 | crashed |
| 8 | 2.59E+01 | 3.73E+01 | crashed |
| 9 | 1.95E+01 | 4.08E+01 | crashed |
| 10 | 1.91E+01 | 3.81E+01 | crashed |
| 11 | 1.77E+01 | 3.81E+01 | crashed |
| 12 | 1.60E+01 | 3.75E+01 | crashed |
| 13 | 1.42E+01 | 3.58E+01 | crashed |
| 14 | 1.45E+01 | 3.59E+01 | crashed |
| 15 | 1.47E+01 | 3.57E+01 | crashed |
| 16 | 1.41E+01 | 3.52E+01 | crashed |
| 17 | 1.54E+01 | 3.45E+01 | crashed |
| 18 | 1.52E+01 | 3.31E+01 | crashed |
| 19 | 1.52E+01 | 3.31E+01 | crashed |
| 20 | 1.38E+01 | 3.16E+01 | crashed |

Table 1.10.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **k3-18** (1155955 equations)

we noticed the library crashed during processing of **k3-18**, **cube-645** and (partially) **pwr-3d** matrices. A debugging process showed that a segmentation fault occurred in *pdgstrf* function during the numerical factorization phase. We still keep working on this problem together with the PETSc team in order to find a solution.

To complete and perform a fair comparison, an additional flat-MPI test was conducted with a 5-point stencil Poisson matrix with 100000 equations. We think this test can partially allow us to estimate parallel performance of systems like **k3-18**, **cube-645** where SuperLU_DIST crashed. The results of the test are given in figure 1.19.

place for figure 1.19

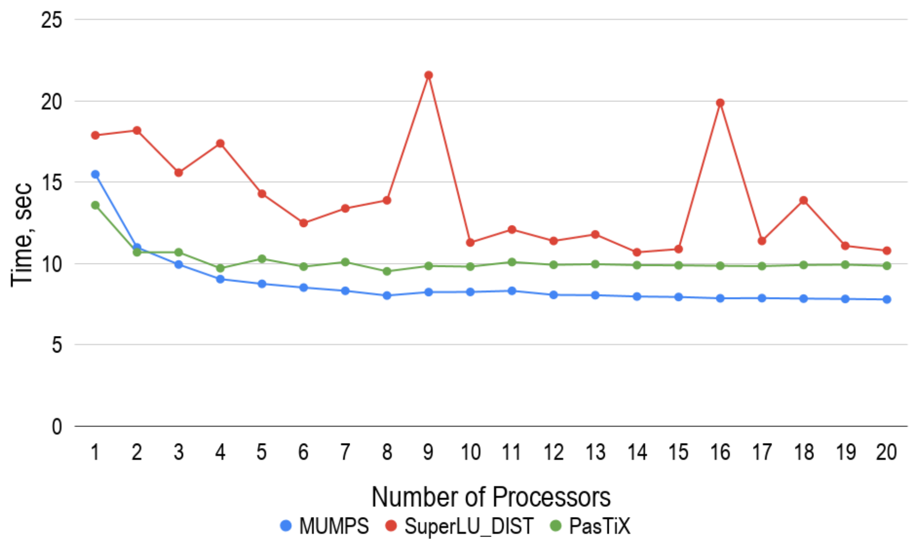


Figure 1.19.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and a 5 point-stencil Poisson matrix (1000000 equations)

According to the test results, it is clear that MUMPS significantly outperforms both SuperLU_DIST and PaStiX. A literature review showed that **wsmp**, in paper [**wsmp**], came to nearly the same results comparing parallel performance of WSMP, MUMPS and SuperLU_DIST libraries with respect to execution time for their matrix set. However, paper [**mm-comparison-of-packages**] showed an almost opposite outcome. According to **mm-comparison-of-packages**, SuperLU_DIST spent less time on factorization and solution phases, which almost always determines the total execution time, and, even more interesting, it scaled much better overall. We must note that both research groups used different machines and matrix sets. This fact actually seconds our idea that a choice of a suitable library can depend heavily on data and hardware.

Taking into consideration the results of our primary flat-MPI tests, the MUMPS library was chosen as a sparse direct solver for our hybrid approach and the following study. Furthermore, an overview of the MUMPS documentation also showed some room for performance tuning that we were going to discuss in detail in sections [bla], [bla] and [bla].

However, it should be mentioned that we cannot exclude that SuperLU_DIST and

PaStiX can perform similar, or even better, as the MUMPS library with appropriate parameters tuning or for another matrix set.

## 1.8. Review of MUMPS Library

Originally, MUMPS library was a part of the PARASOL Project. The project was an ESPRIT IV Long Term Research whose main goal was to build and test a portable library for solving large sparse systems of equations on distributed memory systems [**PARASOL**]. An important aspect of the researh was the strong link between the developers of the sparse solvers and the industrial end users, who provided a range of test problems and evaluated the solvers [**MUMPS:description**]. Since 2000 MUMPS had continued as an ongoing project and, by the moment of writing, the library have contained almost 5 main releases.

It was mentioned in section 1.7 that MUMPS is an implementation of the multifrontal method. Hence, MUMPS sequentially performs all three phases: analysis, numerical factorization and solution. The numerical factorization and solution phases were fully described in detail in section 1.5. It is important to examine the analysis phase of MUMPS library because this phase varies from library to library and plays a significant role on parallel performance.

According to the documentation, the MUMPS analysis phases consists of several pre-processing steps:

1. Fill-reducing pivot order
2. Symbolic factorization
3. Scaling
4. Amalgamantion
5. Mapping

1) To handle both symmetric and unsymmetric cases, MUMPS performs fill-in reordering based on $A + A^T$ sparsity pattern. The library provides numerous sequantial algorithms for reordering such as Approximate Minimum Degree (AMD) [**reordering:AMD**], Approximate Minimum Fill (AMF), Approximate Minimum Degree with automatic quasi-dense row detection (QAMD) [**reordering:QAMD**], Bottom-up

and Top-down Sparse Reordering (PORD) [**reordering:PORD**], Nested Dissection coupled with AMD (Scotch) [**reordering:SCOTCH**], Multilevel Nested Dissection coupled with Multiple Minimum Degree (METIS) [**reordering:METIS**]. Additionally, MUMPS can work together with ParMETIS and PT-Scotch which are extensions of METIS and Scotch libraries for parallel execution. MUMPS also provides the user with an automatic choice option where an appropriate reordering algorithm is selected in run-time based on matrix type and size and the number of processors [**mumps-manual**].

2) Sparsity structures of factors $L$ and $U$ are computed during the step, based on permuted matrix $A$ after fill reducing reordering, in order to build the corresponding elimination tree. All computations are performed on a directed graph $G(A)$ associated with the matrix $A$.

3) At this step, matrix $A$ is scaled in such a way to get absolute values of *one* along the main diagonal and *less than one* for all off-diagonal entries. Scaling algorithms are based on sudies described in detail in works [**mm:scaling:duff1999design**], [**mm:scaling:duff2001algorithms**] (for the unsymmetric case) and [**mm:scaling:duff2005strategies**] (for the symmetric case). This preprocessing step is supposed to improve numerical accuracy and makes all estimations performed during analysis more reliable [**mumps-manual**]. MUMPS also provides an option to switch off scaling or perform it during the factorization phase.

4) During amalgamantion, sets of columns with the same off-diagonal sparsity pattern are group together to create bigger nodes, also known as supernodes. The process leads to restructuring of the initial elimination tree to an amalgamated tree of supernodes which is also know as the *assembly tree*. The main purpose of that step is to improve efficiency of dense matrix operations. An example of the amalgamantion process is shown in section 1.5.

5) A host process, chosen by MUMPS, creates a pool of tasks where each task can be either a subtree or type 2 or type 3 node (figure 1.16). Then each task is mapped by the host among all available processes in such a way to achieve good memory and compute balance.

Type 1 nodes are grouped in subtrees, according to the Geist-Ng algorithm [**geist1989task**], and each subtree is processed only by one single process to avoid the finest granularity, which can cause high communication overheads.

In case of type 2 nodes, the host process assigns each node to one process, which is called the master, which holds fully summed rows and columns of the node and perform pivoting and factorization of these rows. During the numerical factorization phase, in run-time, the master process first receives symbolic information which describes the structure of the contribution blocks sent by its children. At the next step, the master collects information concerning the load of all other processes and decides which of them (*slaves*) are going to participate. Then the master informs the chosen slaves that a new task has been allocated for them and sends them the frontal matrix distribution. After that, the slaves communicates the the children of the master process and collects the corresponding numerical elements. The slaves are in charge of all the assembly and computation of the partly summed rows.

The root node belongs to the type 3. The host statically assigns the master for the root, as it is in case of type 2 nodes, to hold all the indices describing the structure of the frontal matrix. Before factorization, the structure of the frontal matrix of the root is statically mapped onto a 2D grid of processes using block cyclic distribution. This allows to determine, during the analysis phase, which process an entry of the root is assigned. Hence, the original matrix entries and the part of the contribution blocks can be assembled as soon as they are available. Due to partial pivoting, the master process collects the index information for all delayed variables of its sons, builds the final structure of the root frontal matrix and broadcast the corresponding symbolic information to all slave processes. The slave, in turn, adjust their local data structure. After that numerical factorization can be perform in parallel.

It is important to mention that if the size of the root node is less than a certain computer depended parameter, defined by MUMPS, the root node will be treated as the type 2.

An illustrative example of process a mapping together with a combination of static and dynamic scheduling is given in figure 1.20.

place for figure fig:mumps:map and-scheduling

Another outstanding feature of MUMPS is treatment of partial pivoting during the numerical factorization phase. To handle this, MUMPS uses threshold pivoting and delayed pivots approaches which are fully described in section 1.7 where different implementations of direct sparse solvers are compared.

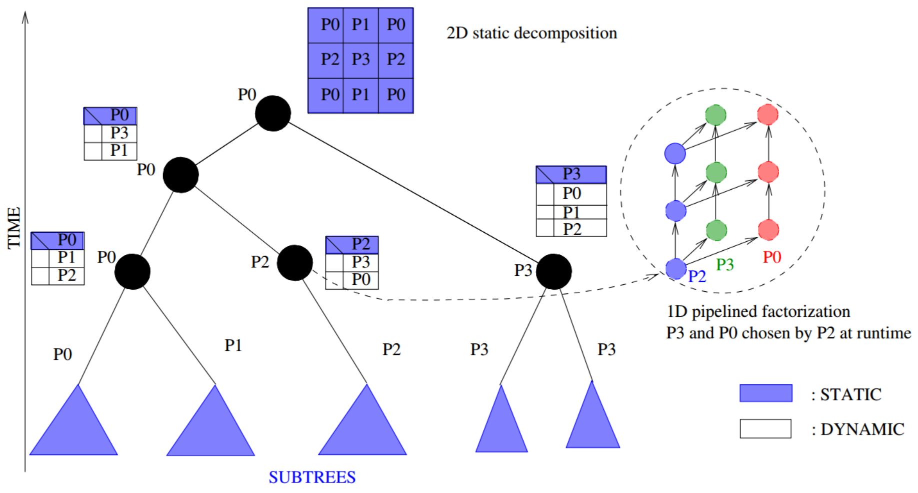


Figure 1.20.: MUMPS: static and dynamic scheduling [**l2012multifrontal**]

## 1.9. Choice of Fill Reducing Reordering

Fill reducing reordering is the first and the most important step of sparse matrix decomposition since it has its direct impact on the assembly tree structure. As we mentioned above, the tree structure defines the task parallelism as well as sizes of frontal matrices and thus performance of the method.

MUMPS provides various algorithms for reordering described in section 1.8. A detailed study and comparison between different methods were done by **guermouche2003memory** in work [**guermouche2003memory**] for sequential execution of the analysis phase. **guermouche2003memory** noticed that the trees generated by METIS and SCOTCH were rather wide (because of the global partitioning performed at the top), while the trees generated by AMD, AMF and PORD tend to be deeper. In addition, they came to two important conclusions. Firstly, they noticed both SCOTCH and METIS generated much better balanced trees in contrast to other methods. Secondly, according to their results, SCOTCH and METIS produced trees with bigger frontal matrices than those generated by the other reorderings [**guermouche2003memory**].

In this section we are going to investigate influence of two different parallel fill reducing packages, namely: PT-Scotch and ParMETIS, on parallel performance of MUMPS. The algorithmic difference between PT-Scotch and ParMETIS was explained in section 1.8.

To perform a test, the default PETSc, MUMPS, PT-Scotch and ParMETIS libraries were downloaded, compiled, configured and link together. The test was carried out using only flat-MPI mode without explicit process pinning. The results are shown in figure 1.21 as well as in appendix B.

According to the results, parallel performance of MUMPS can vary significantly between the libraries. In average, the difference between different fill-in reducing algorithms can achieve almost BRA%.

place for figure 1.21

rewrite the results

It is important to mention that both packages, PT-Scotch and ParMetis, use heuristic approaches with the main aim to reduce fill-in of the factors and thus do not directly consider quality of the resultant elimination tree. It is relevant to assume that efficiency of a particular heuristic can be very sensitive to a matrix structure and size. This fact makes it difficult to predict which algorithm is better to use for a specific case in advance. Taking GRS matrix set as an example, we can observe that PT-Scotch is the best choice for small and medium matrices, namely: *cube-5*, *cube-64*, *k3-2* and *pwr-3d* cases. However, at the same time PerMetis tends to work better for relatively big systems such as *cube-645* and *k3-18*.

insert example of k3-2 and show that expensive reordering can help to get better tree

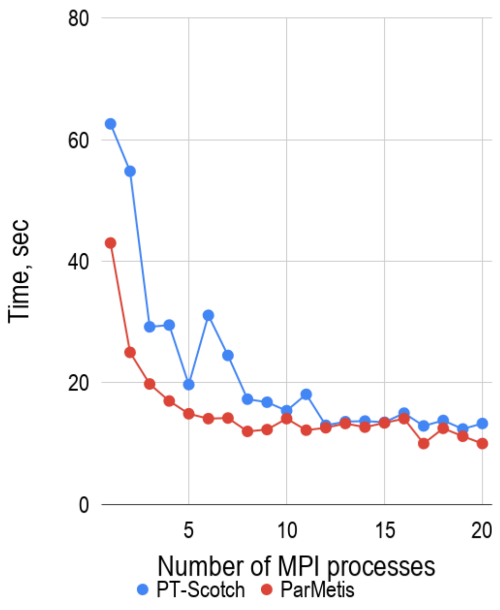
During the test, application of ParMetis in case of small systems of equations showed a strong negative effect on parallel performance of MUMPS. The execution time for factorization of *pwr-3d* and *cube-5* matrices grew with the increase of processing units (figure 1.22).

A simple profiling showed two important things. Firstly, numerical factorization time and time spent on the analysis phase had the same order in case of sequential execution i.e. 1 MPI process. Secondly, while numerical factorization time barely decreased with increase of number of processing elements, time spent on analysis phase time significantly grew. The results of profiling are shown in figure 1.23.

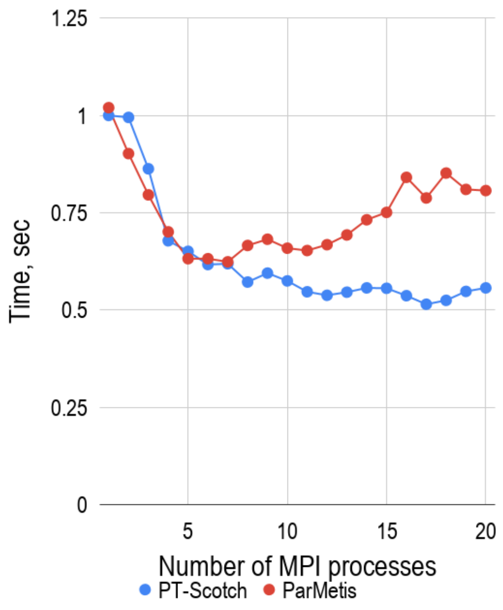
place for figure 1.22

In this section, we have presented the influence of different fill-in reducing algorithms on parallel performance of MUMPS. We have observed the right choice of the algorithm can lead to significant improvements in terms of the overall execution time. We have showed there is no a single algorithm that performs the best for all test cases. At the moment of writing, we came to conclusion there was no an indirect metric to predict the best algorithm in advance for a specific system of equations and only a flat-MPI test could be used for that purposed. Sometimes PT-Scotch and ParMetis can perform nearly the same like it is in case of *CurlCurl_3* and *cant*, for

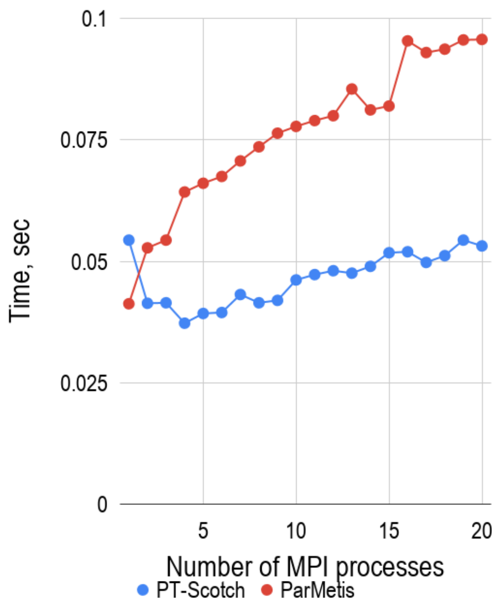
place for figure 1.23



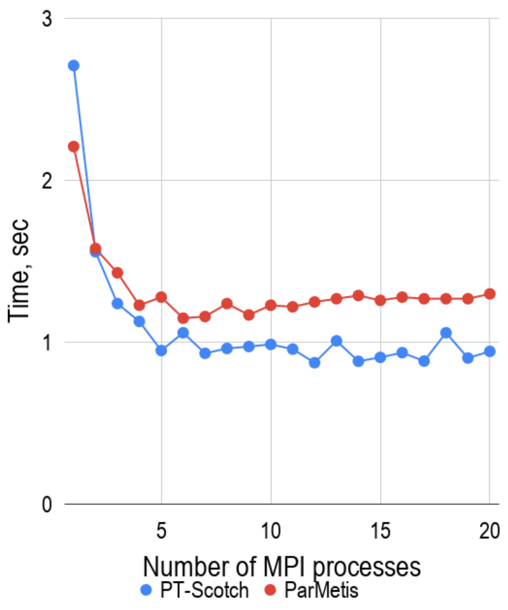
(a) k3-18



(b) cube-64

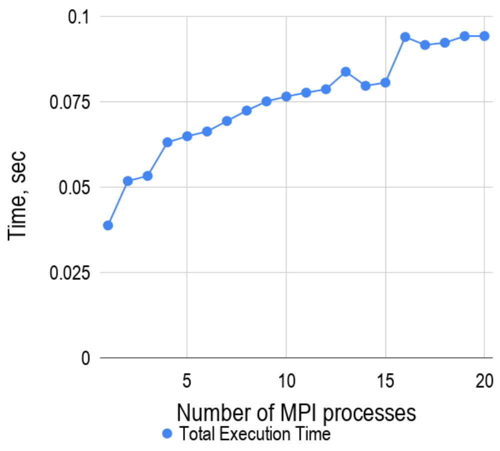


(c) pwr-3d

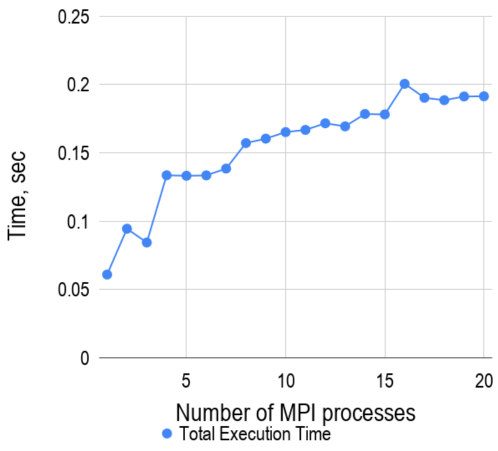


(d) k3-2

Figure 1.21.: Comparison of different fill-reducing algorithms

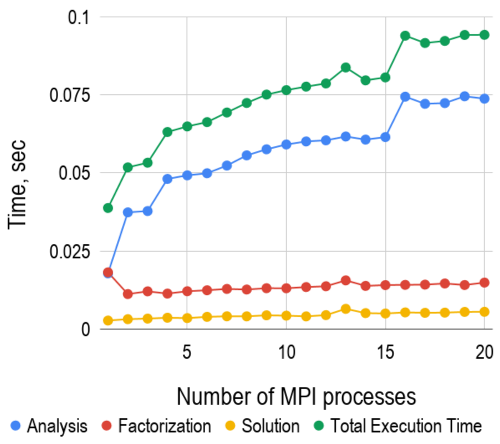


(a) pwr-3d

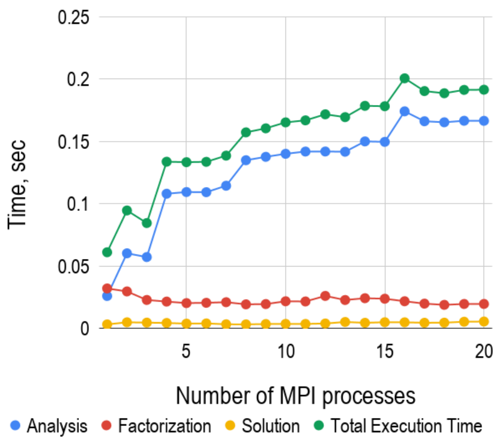


(b) cube-5

Figure 1.22.: MUMPS-ParMetis parallel performance in case of relatively small matrices



(a) pwr-3d



(b) cube-5

Figure 1.23.: Profiling of MUMPS library with using ParMetis as a fill-in reducing algorithm in case of factorization of relatively small matrices

example (see appendix B). Therefore, from time to time, it can be quite difficult to make a decision which package to use. At the end, we have assigned each test case to a specific fill reducing reordering method based on results of the conducted experiments and our subjective opinion. The results are summarized it in tables 1.11 and 1.12.

| Matrix Name | Ordering | n | nnz | nnz / n |
|---|---|---|---|---|
| cube-5 | PT-Scotch | 9325 | 117897 | 12.6431 |
| cube-64 | PT-Scotch | 100657 | 1388993 | 13.7993 |
| cube-645 | ParMetis | 1000045 | 13906057 | 13.9054 |
| k3-2 | PT-Scotch | 130101 | 787997 | 6.0568 |
| k3-18 | ParMetis | 1155955 | 7204723 | 6.2327 |
| pwr-3d | PT-Scotch | 6009 | 32537 | 5.4147 |

Table 1.11.: GRS matrix set: assignment of matrices to a specific fill-in reducing algorithm based on parallel performance during flat-MPI tests

| Matrix Name | Ordering | n | nnz | nnz / n |
|---|---|---|---|---|
| cant | ParMetis | 62451 | 4007383 | 64.1684 |
| consph | PT-Scotch | 83334 | 6010480 | 72.1252 |
| memchip | PT-Scotch | 2707524 | 13343948 | 4.9285 |
| PFlow_742 | PT-Scotch | 742793 | 37138461 | 49.9984 |
| pkustk10 | PT-Scotch | 80676 | 4308984 | 53.4110 |
| torso3 | ParMetis | 259156 | 4429042 | 17.0903 |
| x104 | PT-Scotch | 108384 | 8713602 | 80.3956 |
| CurlCurl_3 | PT-Scotch | 1219574 | 13544618 | 11.1060 |
| Geo_1438 | ParMetis | 1437960 | 63156690 | 43.9210 |

Table 1.12.: SuiteSparse matrix set: assignment of matrices to a specific fill-in reducing algorithm based on parallel performance during flat-MPI tests

From now onwards, we will use this assignment for the rest of the study to keep consistency between tests and show the overall effect of *optimal parameters* on the default MUMPS settings at the end.

## 1.10. MUMPS: Process Pinning

Due to intensive and complex manipulations with frontal and and contribution matrices, we can assume that MUMPS belongs to memory bound applications. In this case memory access can be a bottleneck for the library. A common way to improve performance of memory bound applications running on distributed memory machines is to distribute processes equally among sockets of a node.

However, because MUMPS uses both task and data parallelism as well as a complex hybrid, both static and dynamic, task scheduling, it becomes difficult to decide which pinning strategy is better i.e. *close* or *spread*, described in section 1.2.

Therefore, a couple of tests were conducted with both GRS and SuiteSparse matrix sets in order to investigate influence of different strategies on MUMPS performance. For this group of tests, only MUMPS default settings together with a specific fill-in reducing algorithm for each test case, mentioned in section 1.9, were used. The tests were performed on HW1 machine using only flat-MPI mode. Results are shown in figures 1.24, 1.25, 1.26 and in appendix C. The graphs depict the total time spent, i.e. analysis, factorization and solution.

place for figure 1.24

place for figure 1.25

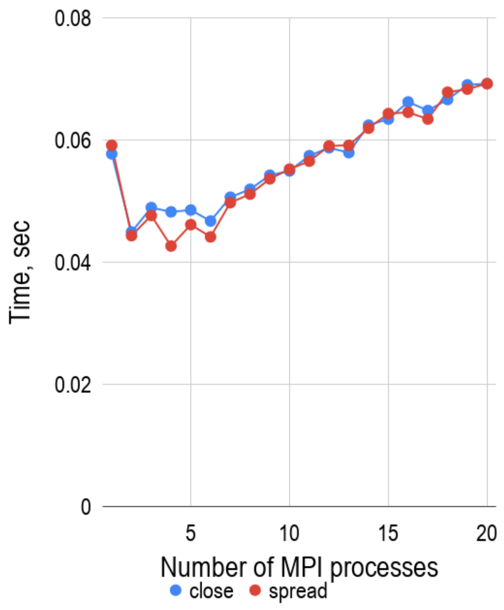
place for figure 1.26

The tests revealed that *spread*-pinning performed better and allowed to reduce runtime by approximately BRA% in average in contrast to the *close* strategy. As expected, the points with 1 and 20 MPI processes show the same performance because they basically represent the same process distribution. Additionally, almost BRA% improvement can be observed around the saturation point in case of *spread* strategy application.

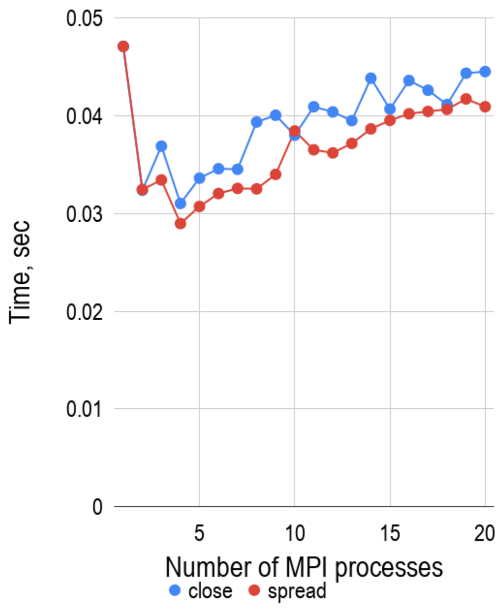
Taken into account results of the tests, *spread*-pinning has been chosen for the rest of the study because it shows better overall performance in the intermediate range, in terms of the number MPI processes, where a saturation point can occur. As an example, we would like to point out this process distribution can be particular useful for matrices such as *pwr-3d* and *cube-5* where saturation happens at the number of processes equal to 4.

In general, such process distribution can be easily achieved by means of some advanced OpenMPI options, for example *–map-by*, as following:.

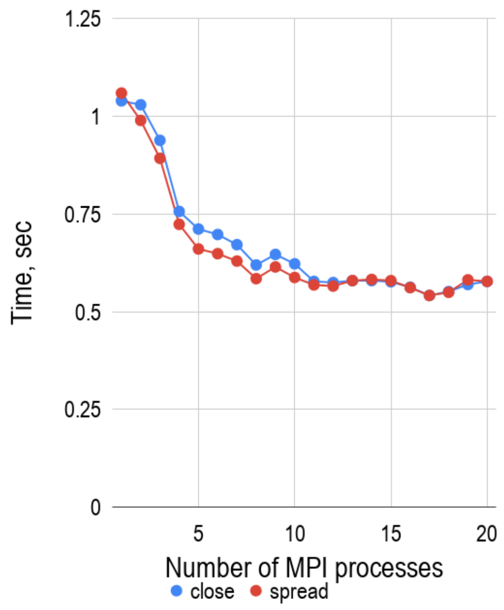
```
mpiexec --map-by socket -n $num_proc $executable_name $parameters
```



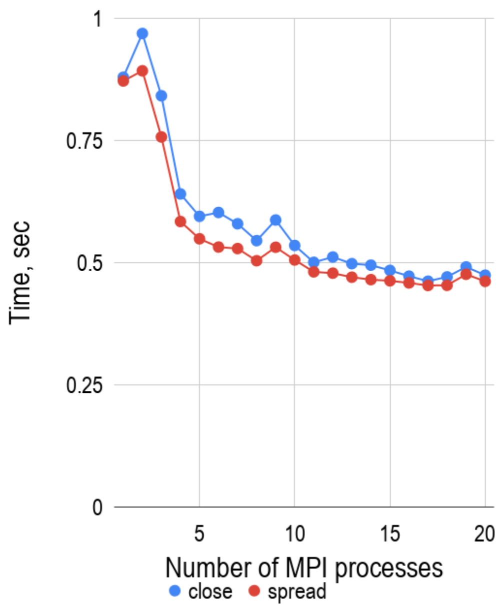
(a) HW1 - pwr-3d



(b) HW2 - pwr-3d

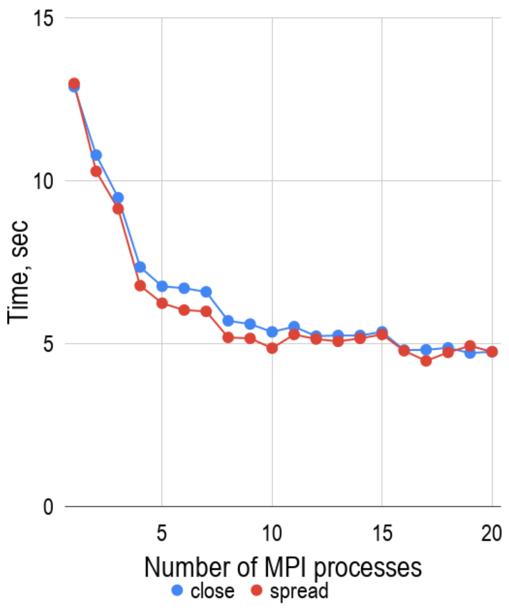


(c) HW1 - cube-64

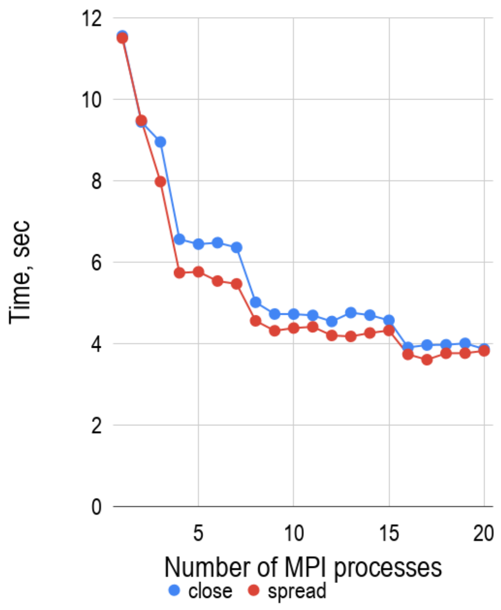


(d) HW2 - cube-64

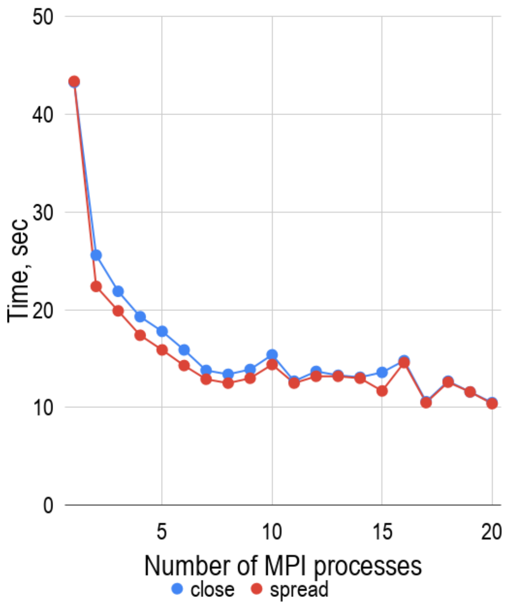
Figure 1.24.: Comparison of *close* and *spread* pinning strategies



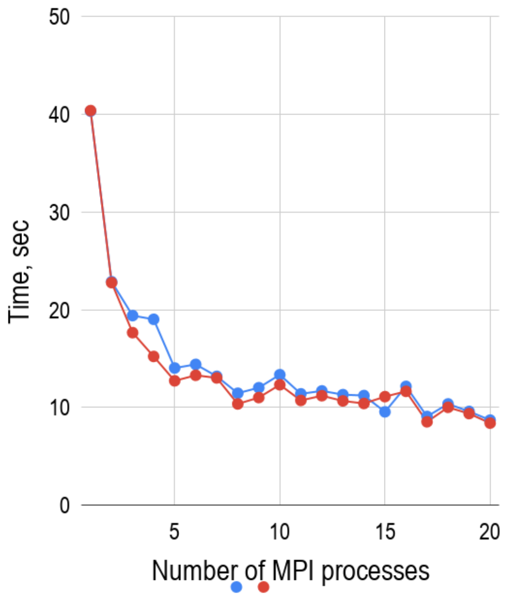
(a) HW1 - cube-645



(b) HW2 - cube-645

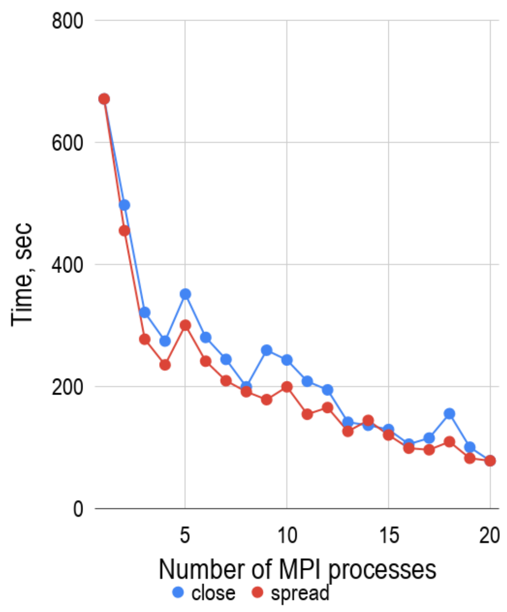


(c) HW1 - k3-18

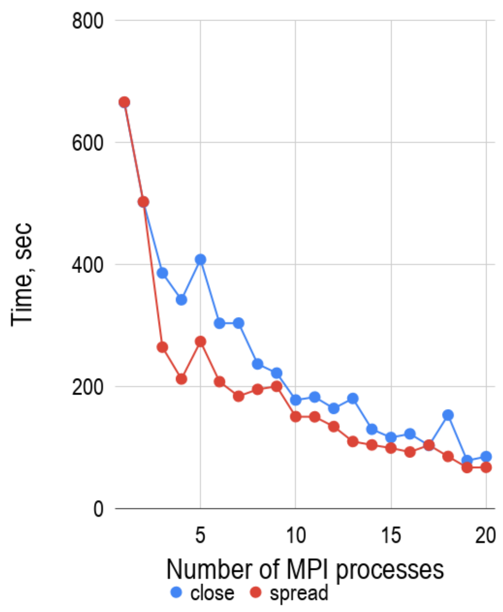


(d) HW2 - k3-18

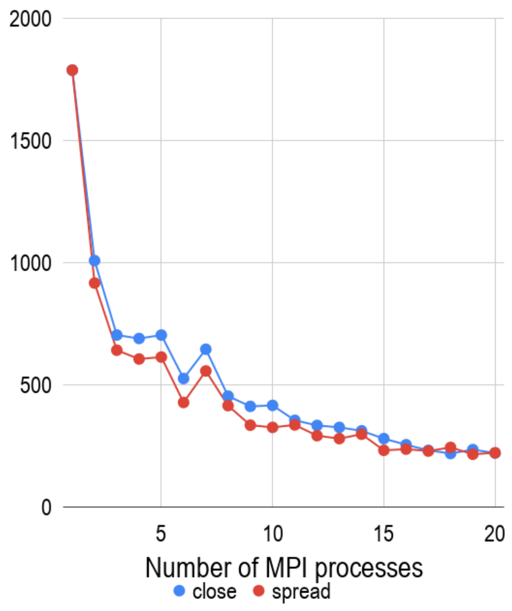
Figure 1.25.: Comparison of *close* and *spread* pinning strategies



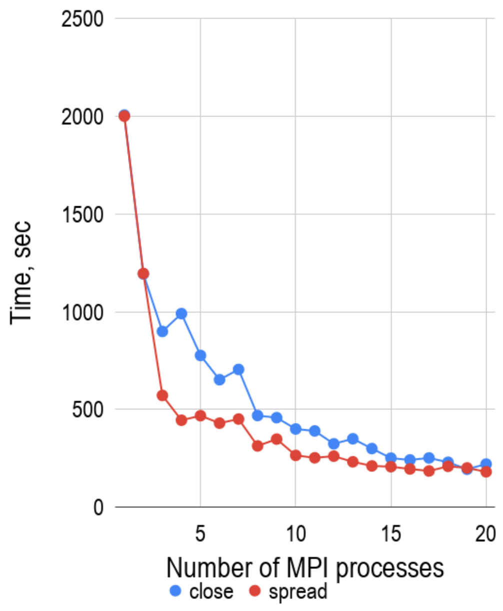
(a) HW1 - PFlow_742



(b) HW2 - PFlow_742



(c) HW1 - CurlCurl_3



(d) HW2 - CurlCurl_3

Figure 1.26.: Comparison of *close* and *spread* pinning strategies

Listing 1.3: An example of *spread*-pinning with using OpenMPI options in case of a flat-MPI run

## 1.11. Choice of BLAS Library

To perform columns elimination of fully summed block of frontal matrices, MUMPS intensively uses GEMM, TRSM and GETRF subroutines which are parts of BLAS and LAPACK libraries. As an example, figure 1.28 demonstrates factorization of a type 2 node.

place for figure 1.27

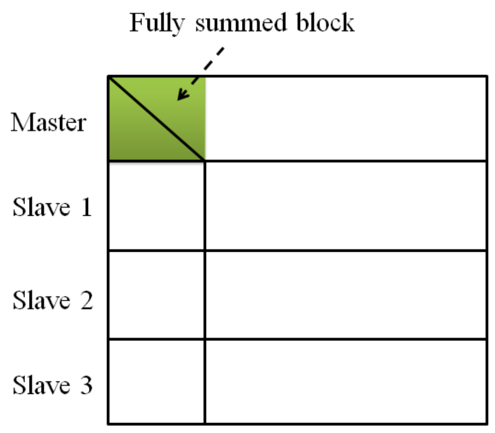


Figure 1.27.: MUMPS: static and dynamic scheduling

place for figure fig:mumps:step-of-type-2-factorization

Both BLAS and LAPACK originate from the Netlib project which is a repository of numerous scientific computing software maintained by AT&T Bell Laboratories, the University of Tennessee, Oak Ridge National Laboratory and other scintific communities [**netlib-overview**].

The goal of BLAS library is to provide a high efficient implementation of common dense linear algebra kernels by means of high rate of floating point operations per memory access, low cache and Translation Lookaside Buffer (TLB) miss rates.

In its turn, LAPACK is designed in such a way so that as much as possible computations is performed by calls to BLAS library. This allows to achieve high efficiency

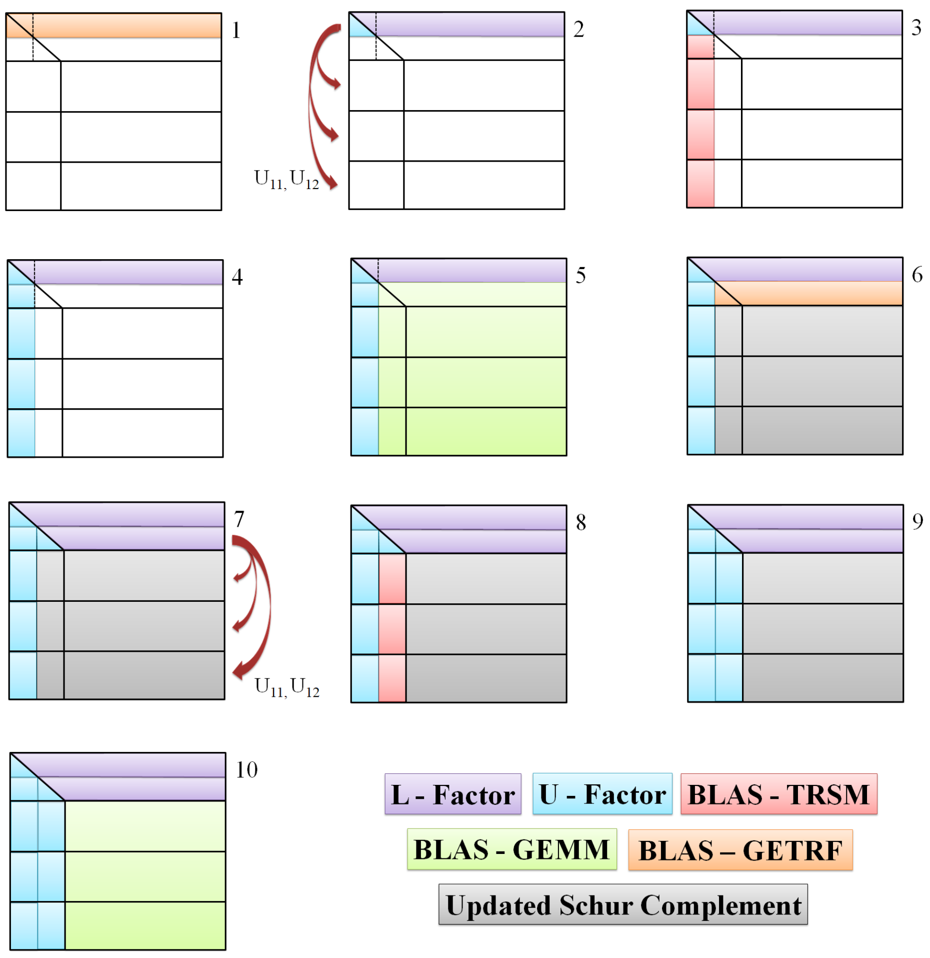


Figure 1.28.: MUMPS: An example of a type 2 node factorization

for operations such as $LU$, $QR$, $SVD$ decompositions, triangular solve, etc. on modern computers. However, the Netlib BLAS implementation is written for an abstract general-purpose central processing unit, in mind, where hardware parameters are based on market statistics. Hence, it is not possible to achieve the maximum possible performance on a specific machine.

Hence, there exist special-purpose, hardware-specific implementations of the library developed by hardware vendors i.e. IBM, Cray, Intel, AMD, etc., as well as open-source tuned implementations such as ATLAS, OpenBLAS, etc. To achieve full compatibility, the developers consider the Netlib implementation of BLAS library as the standard (or reference) and thus overwrite all subroutines with additional tuning and optimization. This approach makes it possible to easily replace different BLAS implementations during object files linking without any modifications of the source code.

Table 1.13 shows commercial and open-source tunned BLAS implementations available on the market today.

Among all libraries listed in table 1.13 there were only four available on HW1 machine, namely: Netlib BLAS, Intel MKL, OpenBLAS and ATLAS. However, installation of ATLAS requires to switch off dynamic frequency scaling, also called CPU throttling, to allow an ATLAS configuration routine to find the best loop transformation parameters for a specific hardware. In order to turn off CPU throttling, one has to reboot the entire machine and make appropriate changes in Basic Input/Output System (BIOS). This fact made ATLAS library not suitable for the rest of the study and we excluded it from the primary list of candidates. Moreover, during installation, one has to explicitly provide the number of OpenMP threads that are going to be used once a BLAS subroutine is called. This means there is no way to change the number of threads per MPI process in run-time without re-installation of ATALS library. Thus, only 3 versions of MUMPS-PETSc (Netlib BLAS, Intel MKL and OpenBLAS) library were compiled, installed and tested with using both GRS and SuiteSparse matrix sets and 1 thread per MPI process. The test results are shown in figure 1.29 and appendix D.

The tests show that OpenBLAS significantly outperforms both Netlib and Intel MKL libraries in case of GRS matrix set. In average, OpenBLAS is about **10%** faster than the default Netlib implementation and approximately **17%** faster than Intel MKL library. It is interesting to notice that parallel performance of OpenBLAS is a bit better around the saturation points in contrast to the competitors. It turns out that OpenBLAS is **11%** and **18%** faster than Netlib and Intel MKL, respectively.

place for figure 1.29

place for figure 1.30

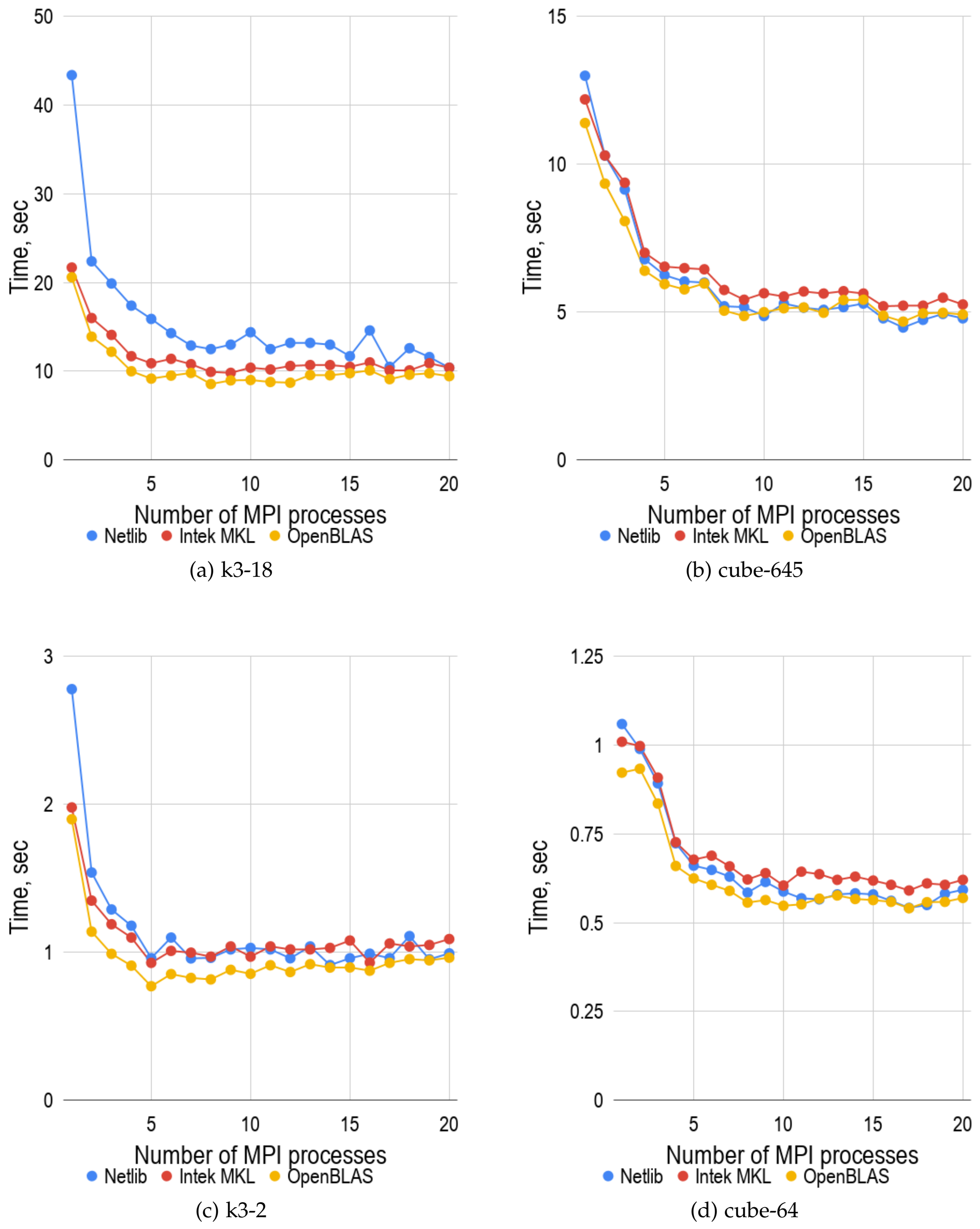
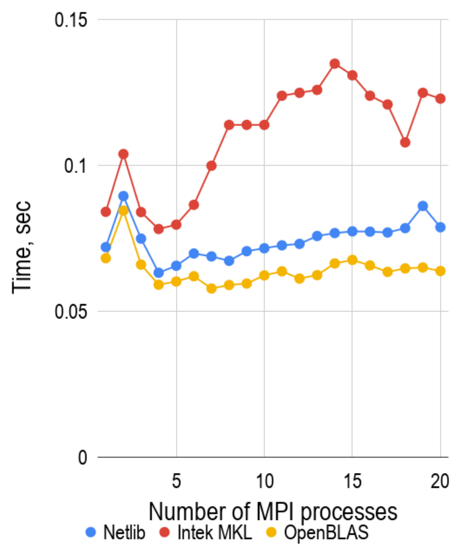
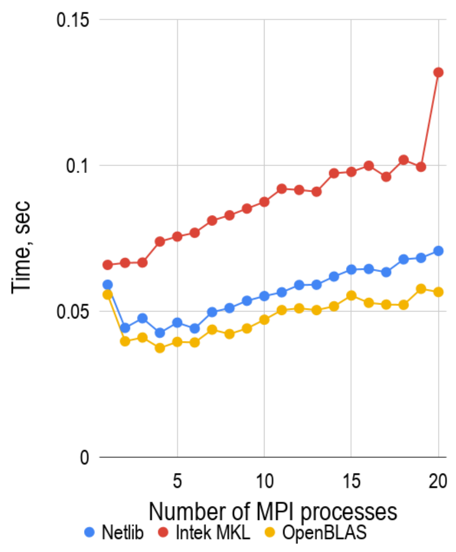


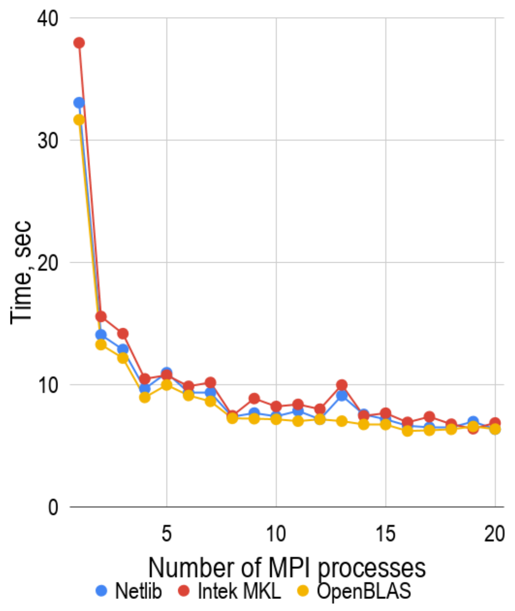
Figure 1.29.: MUMPS: comparison of different BLAS libraries with using GRS matrix set



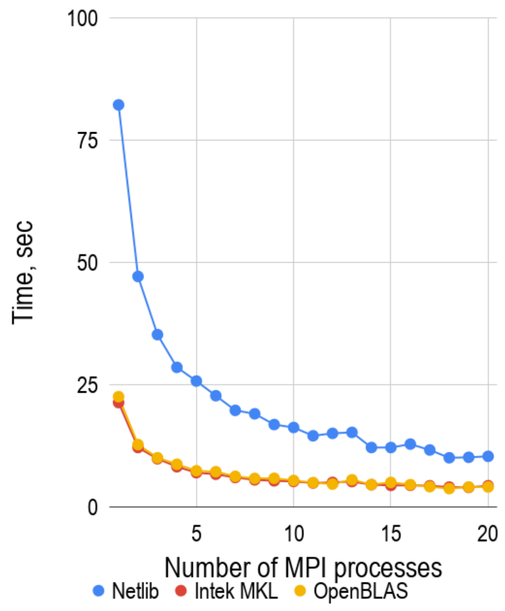
(a) cube-5



(b) pwr-3d



(c) memchip



(d) torso3

Figure 1.30.: MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets

| Name | Description | License |
|---|---|---|
| Accelerate | Apple's implementation for macOS and iOS | proprietary license |
| ACML | BLAS implementation for AMD processors | proprietary license |
| C++ AMP | Microsoft's AMP language extension for Visual C++ | open source |
| ATLAS | Automatically tuned BLAS implementation | open source |
| Eigen BLAS | BLAS implemented on top of the MPL-licensed Eigen library | open source |
| ESSL | optimized BLAS implementation for IBM's machines | proprietary license |
| GotoBLAS | Kazushige Goto's implementation of BLAS | proprietary license |
| HP MLIB | BLAS implementation supporting IA-64, PA-RISC, x86 and Opteron architecture | proprietary license |
| Intel MKL | Intel's implementation of BLAS optimized for Intel Pentium, Core, Xeon and Xeon Phi | proprietary license |
| Netlib BLAS | The official reference implementation on Netlib | open source |
| OpenBLAS | Optimized BLAS library based on GotoBLAS | open source |
| PDLIB/SX | BLAS library targeted to the NEC SX-4 system | proprietary license |
| SCSL | BLAS implementations for SGI's Irix workstations | proprietary license |
| Sun Performance Library | Optimized BLAS and LAPACK for SPARC, Core and AMD64 architectures under Solaris 8, 9, and 10 as well as Linux | proprietary license |

Table 1.13.: Comerrcial and open source BLAS implementations **[wiki:blas-implementations]**

Slightly different results were observed in case of SuiteSparse matrix set. In general, Intel MKL and OpenBLAS demonstrate approximately the same efficiency and considerably outperform Netlib BLAS implementation. In opposite to GRS matrix set, it appears that Intel MKL is often faster than OpenBLAS for almost all test cases from the set. However, the difference between them is negligibly small.

All in all, we can conclude that OpenBLAS implementation is the best choice for matrices generated by GRS software in case of flat-MPI execution strategy.

## 1.12. MPI-OpenMP Tuning of MUMPS Library

As it was mentioned in section 1.8, the development of MUMPS began in 1996 when message-passing programming paradigm dominated in parallel computing. Therefore, the library originally was designed only for distributed-memory machines.

In 2010, **chowdhury2010some** published their first experiments and some issues, in [**chowdhury2010some**], of exploiting shared memory parallelism in MUMPS. The authors showed that it was possible to achieve some improvements in multicore systems with multithreading, given a purely MPI application. However, later **l2013introduction** mentioned, in [**l2013introduction**], that adaptation of existing code for NUMA architecture was still a challenge because of memory allocation, memory affinity, thread pinning and other related issues.

In spite of natural data locality of message-passing applications which is always beneficial, a general motivation of switching to a hybrid mode, mixed MPI/OpenMP, is to reduce communication overheads between processes. According to the profiling results done by **chowdhury2010some**, MUMPS contained four main initial sources of shared-memory parallelization, namely:

1. BLAS Level 1, 2, 3 operations during both factorization and solution phases
2. Assembly operations, where contribution blocks of children nodes of the assembly tree are assembled at the parent level
3. Copying contribution blocks during stacking operations
4. Pivot search operations

Almost all customized BLAS libraries, for example Intel MKL and OpenBLAS, are multi-threaded and can efficiently work in shared-memory environment. Thus, parallelization of region 1 can be achieved by linking a suitable BLAS library whereas regions 2, 3 and 4 are multithreaded by inserting appropriate OpenMP directives above the corresponding loop statements.

A detailed review of works [**l2013introduction**] and [**chowdhury2010some**] reveals that, in general, a pure OpenMP or mixed MPI/OpenMP strategy can reduce run-time of MUMPS. In average, factorization time is reduced by **14.3%** and in some special cases improvement reaches about **50.4%**, according to the data provided in the papers. However, at the same time, the results demonstrate that pure-MPI mode sometimes can significantly outperform any hybrid mixed strategy.

By and large, the results show two important aspects. Firstly, performance of a specific strategy depends heavily on the resultant assembly tree and thus on the matrix sparsity pattern and applied fill reducing algorithm. Secondly, it is not possible to guess in advance which strategy gives the best parallel performance without detailed information about the tree structure and computational cost per node. **l2013introduction** showed that performance of a particular mode depended on the ratio of large and small fronts. For example, they noticed more threads per MPI process leaded to better parallel performance when the ratio was high. On the other hand, they observed the absolutely opposite result with relatively small ratios. Unfortunately, **l2013introduction** did not provide any quantitative measure for that in their work [**l2013introduction**].

It is also interesting to notice that parallelization of region 1 with using a multithreaded BLAS library brings most of parallel performance for mixed or pure OpenMP strategies, according to the results from[**l2013introduction**]. But, at the same time, performance of regions 2, 3, 4 multithreaded by OpenMP directives is marginal. In average, it allows to reduce numerical factorization run-time by only **0.66%**.

This outcome is expected because BLAS subroutines, especially level 3, have a high ratio of floating point operations per memory access and thus PEs re-use data stored in caches as much as possible. Meanwhile, regions 2, 3, 4 mainly perform initialization, data movement and execution of *if-statements* which lead to a low compute ratio.

Additionally, it is worth noticing that both works, [**chowdhury2010some**] and [**l2013introduction**], were mainly focused on the numerical factorization phase assuming that both analysis and solution phases do not take lots of time. In spite of credibility of this assumption, it still should be pointed out the solution phase runs faster in case of flat-MPI mode. This fact becomes even more interesting because, in our case, a system with multiple right-hand sides has to be solved in order to generate a preconditioner.

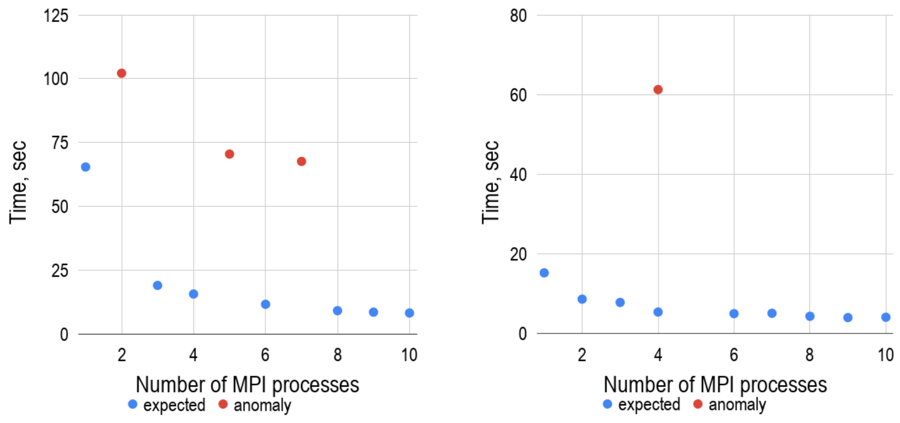
We have to admit that both works, [**chowdhury2010some**] and [**l2013introduction**], are relatively old and the analysis above may be not complete and full. Because MUMPS is a dynamic developing project, we can expect that adaptation of shared-memory parallelization in MUMPS has been significantly advanced since that time. Since the release of MUMPS version 4, the developers have persistently recommended to use only a hybrid mode like*one MPI process per socket and as many threads as the number of cores* [**mumps-manual**].

As an initial test, we decided to compare influence of both Intel MKL and OpenBLAS libraries on parallel performance of MUMPS using GRS matrix set only. In order to pin OpenMP threads in a right way, without any conflict between them, the following OpenMP environment variables were set as:

- OMP_PLACES=cores
- OMP_PROC_BIND=spread

During the test, we found that run-time of MUMPS-OpenBLAS configuration abnormality increased for some test cases. For instance, in case of matrix *cube-645*, the increase reached almost 450% in contrast to the pure sequential execution.

place for figure 1.31

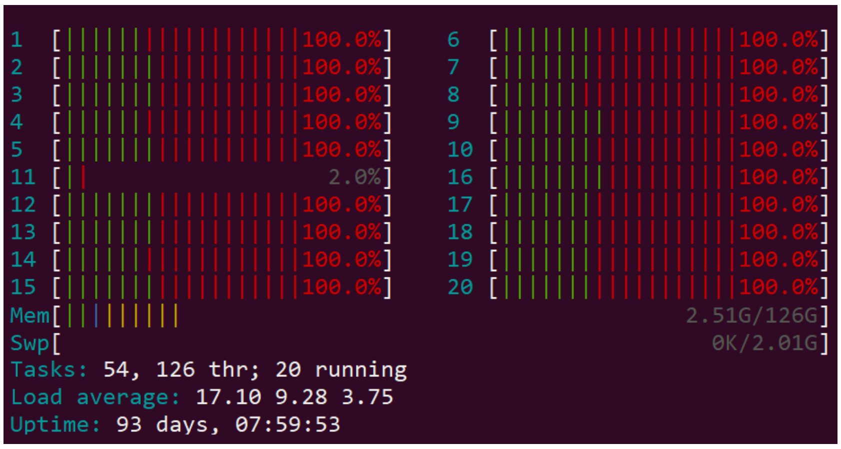


(a) k3-18 (b) cube-645

Figure 1.31.: Anomalies of MUMPS-OpenBLAS configuration running with 2 OpenMP threads per MPI process

Multiple conflicts between application and system threads were observed using *htop* as an interactive process viewer. Figure 1.32 shows a snapshot taken during factorization of matrix *k3-18* running with 1 MPI process and 20 threads.

place for figure 1.32



```
1  [||||||||||||||||||100.0%]   6  [||||||||||||||||||100.0%]
2  [||||||||||||||||||100.0%]   7  [||||||||||||||||||100.0%]
3  [||||||||||||||||||100.0%]   8  [||||||||||||||||||100.0%]
4  [||||||||||||||||||100.0%]   9  [||||||||||||||||||100.0%]
5  [||||||||||||||||||100.0%]   10 [||||||||||||||||||100.0%]
11 [||                  2.0%]   16 [||||||||||||||||||100.0%]
12 [||||||||||||||||||100.0%]   17 [||||||||||||||||||100.0%]
13 [||||||||||||||||||100.0%]   18 [||||||||||||||||||100.0%]
14 [||||||||||||||||||100.0%]   19 [||||||||||||||||||100.0%]
15 [||||||||||||||||||100.0%]   20 [||||||||||||||||||100.0%]
Mem[|||||||||                                    2.51G/126G]
Swp[                                               0K/2.01G]
Tasks: 54, 126 thr; 20 running
Load average: 17.10 9.28 3.75
Uptime: 93 days, 07:59:53
```

Figure 1.32.: A MUMPS-OpenBLAS thread conflict in case of *k3-18* matrix factorization (green - application threads, red - system threads)

It is difficult to say what exactly caused such behavior. However, **chowdhury2010some** also reported about the same problem with using GotoBLAS (OpenBLAS). They assumed that GotoBLAS created and kept some threads active even after the main threads returned to the calling application which could lead to interference with threads created in other OpenMP regions [**chowdhury2010some**]. For that reason, we decided to stick only to Intel MKL library for the rest of the study in this section because there were no conflicts detected during initial runs.

At the beginning, only common mixed MPI/OpenMP modes were tested in order to check influence of shared-memory parallelism on parallel performance of MUMPS as well as to limit the amount of testing. The following strategies were chosen, namely: 20 MPI - 1 thread (flat-MPI), 10 MPI - 2 threads, 4 MPI - 5 threads, 2 MPI - 10 threads, 1 MPI - 20 threads (flat-OpenMP). Additionally, MUMPS was set as a preconditioning algorithm for the GMRES solver with only *one iteration*. This allowed to force MUMPS library to solve systems multiple right-hand sides. According to our assumption, time spent on one GMRES iteration is negligible in contrast time spent on sparse direct matrix factorization. The test results are shown in tables BRA, BRA and BRA as well as in appendix BRA. Numerical values in tables are given in seconds.

| Matrix Name | 20 MPI 1 thread | 10 MPI 2 threads | 4 MPI 5 threads | 2 MPI 10 threads | 1 MPI 20 threads | Gain w.r.t. flat-MPI |
|---|---|---|---|---|---|---|
| k3-18 | **12.520** | 12.630 | 14.010 | 18.020 | 19.170 | - |
| k3-2 | 1.341 | **1.250** | 1.470 | 1.671 | 2.052 | 1.073 |
| cube-645 | **6.585** | 6.859 | 8.552 | 12.010 | 14.080 | - |
| cube-64 | 0.756 | **0.749** | 0.874 | 1.178 | 1.354 | 1.010 |
| cube-5 | 0.181 | 0.132 | **0.104** | 0.126 | 0.117 | 1.744 |
| pwr-3d | 0.130 | 0.114 | 0.0972 | **0.077** | 0.109 | 1.691 |

Table 1.14.: GRS - HW1

| Matrix Name | 20 MPI 1 thread | 10 MPI 2 threads | 4 MPI 5 threads | 2 MPI 10 threads | 1 MPI 20 threads | Gain w.r.t. flat-MPI |
|---|---|---|---|---|---|---|
| k3-18 | 8.558 | **7.819** | 8.165 | 11.330 | 14.320 | 1.095 |
| k3-2 | 1.168 | **0.788** | 0.956 | 1.131 | 1.651 | 1.482 |
| cube-645 | 5.735 | **4.859** | 6.069 | 9.360 | 11.040 | 1.180 |
| cube-64 | 0.805 | **0.541** | 0.664 | 0.947 | 0.918 | 1.490 |
| cube-5 | 0.241 | 0.121 | **0.093** | 0.129 | 0.126 | 2.582 |
| pwr-3d | 0.234 | 0.095 | 0.098 | **0.070** | 0.094 | 3.341 |

Table 1.15.: GRS - HW2

## 1.13. Conclusions

## 1.14. Outlook

- enhance the GRS matrix set to include more cases. Creation of database or collection
- create automatic testing environment. Since solvers to test solver updates
- Optimization engine for preconditioner parameter search.
- invasive computing (resource-aware programming)

| Matrix Name | 20 MPI 1 thread | 10 MPI 2 threads | 4 MPI 5 threads | 2 MPI 10 threads | 1 MPI 20 threads | Gain w.r.t. flat-MPI |
|---|---|---|---|---|---|---|
| cant | 1.400 | **0.990** | 1.050 | 1.605 | 2.019 | 1.414 |
| consph | 3.495 | **2.652** | 3.015 | 3.706 | 3.714 | 1.318 |
| memchip | **7.470** | 9.080 | 13.301 | 20.198 | 45.800 | - |
| PFlow_742 | 26.802 | 24.204 | **21.897** | 30.389 | 54.501 | 1.224 |
| pkustk10 | **0.748** | 0.879 | 0.972 | 1.459 | 1.280 | - |
| torso3 | **3.922** | 4.285 | 4.642 | 5.603 | 8.144 | - |
| x104 | **1.597** | 1.644 | 2.024 | 3.208 | 2.167 | - |
| CurlCurl_3 | 49.250 | 44.120 | **39.909** | 43.311 | 63.001 | 1.234 |
| Geo_1438 | 478.101 | 234.697 | **151.603** | 157.697 | 158.102 | 3.154 |

Table 1.16.: SuiteSparse - HW1

| Matrix Name | 20 MPI 1 thread | 10 MPI 2 threads | 4 MPI 5 threads | 2 MPI 10 threads | 1 MPI 20 threads | Gain w.r.t flat-MPI |
|---|---|---|---|---|---|---|
| cant | 2.128 | **0.955** | 1.011 | 1.577 | 2.058 | 2.229 |
| consph | 3.840 | **2.852** | 3.111 | 3.695 | 3.897 | 1.346 |
| memchip | **7.811** | 7.816 | 9.811 | 15.160 | 31.969 | - |
| PFlow_742 | 24.190 | 29.241 | **19.686** | 27.530 | 55.431 | 1.230 |
| pkustk10 | 1.373 | **0.904** | 1.022 | 1.421 | 1.403 | 1.520 |
| torso3 | 4.733 | **4.080** | 4.483 | 5.648 | 8.217 | 1.160 |
| x104 | 2.676 | **1.597** | 2.025 | 3.204 | 2.133 | 1.676 |
| CurlCurl_3 | 39.890 | **34.579** | 38.620 | 41.171 | 67.760 | 1.154 |
| Geo_1438 | xxx | yyy | zzz | aaa | bbb | ccc |

Table 1.17.: SuiteSparse - HW2

# Appendices

# A. Choice of a Sparse Direct Solver Library

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 1 | 4.58E-02 | 5.60E-02 | 4.64E+00 |
| 2 | 4.31E-02 | 5.14E-02 | 1.89E+00 |
| 3 | 4.51E-02 | 5.28E-02 | 1.22E+00 |
| 4 | 4.61E-02 | 5.64E-02 | 9.13E-01 |
| 5 | 4.92E-02 | 5.97E-02 | 7.70E-01 |
| 6 | 5.37E-02 | 6.14E-02 | 6.04E-01 |
| 7 | 5.42E-02 | 6.51E-02 | crashed |
| 8 | 5.41E-02 | 6.60E-02 | 4.81E-01 |
| 9 | 5.69E-02 | 6.84E-02 | 4.35E-01 |
| 10 | 5.86E-02 | 7.22E-02 | 4.08E-01 |
| 11 | 5.93E-02 | 8.97E-02 | crashed |
| 12 | 6.07E-02 | 9.20E-02 | 3.61E-01 |
| 13 | 6.26E-02 | 8.25E-02 | crashed |
| 14 | 6.28E-02 | 9.75E-02 | crashed |
| 15 | 6.43E-02 | 1.03E-01 | 3.05E-01 |
| 16 | 6.55E-02 | 1.05E-01 | 2.99E-01 |
| 17 | 6.61E-02 | 9.46E-02 | crashed |
| 18 | 6.73E-02 | 1.24E-01 | 2.65E-01 |
| 19 | 6.84E-02 | 1.14E-01 | crashed |
| 20 | 7.02E-02 | 1.32E-01 | 2.60E-01 |

Table A.1.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **pwr-3d** (6009 equations)

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 1 | 1.55E+02 | 6.44E+01 | time-out |
| 2 | 6.28E+01 | 4.84E+01 | time-out |
| 3 | 5.06E+01 | 5.02E+01 | time-out |
| 4 | 4.17E+01 | 4.50E+01 | time-out |
| 5 | 2.52E+01 | 3.98E+01 | time-out |
| 6 | 2.58E+01 | 4.29E+01 | time-out |
| 7 | 2.65E+01 | 4.30E+01 | time-out |
| 8 | 2.59E+01 | 3.73E+01 | time-out |
| 9 | 1.95E+01 | 4.08E+01 | time-out |
| 10 | 1.91E+01 | 3.81E+01 | time-out |
| 11 | 1.77E+01 | 3.75E+01 | time-out |
| 12 | 1.60E+01 | 3.58E+01 | time-out |
| 13 | 1.42E+01 | 3.59E+01 | time-out |
| 14 | 1.45E+01 | 3.57E+01 | time-out |
| 15 | 1.47E+01 | 3.52E+01 | time-out |
| 16 | 1.41E+01 | 3.45E+01 | time-out |
| 17 | 1.54E+01 | 3.31E+01 | time-out |
| 18 | 1.52E+01 | 3.31E+01 | time-out |
| 19 | 1.52E+01 | 3.16E+01 | time-out |
| 20 | 1.38E+01 | 3.15E+01 | time-out |

Table A.2.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **k3-2** (130101 equations)

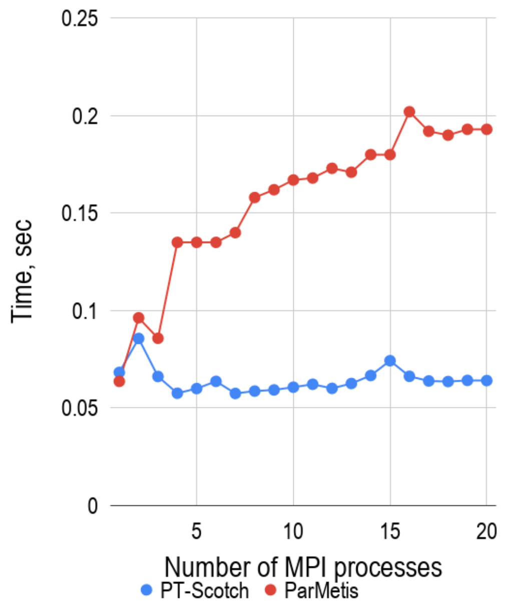
| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 1 | 1.52E+01 | 1.61E+01 | crashed |
| 2 | 1.13E+01 | 1.13E+01 | crashed |
| 3 | 1.00E+01 | 1.03E+01 | crashed |
| 4 | 9.29E+00 | 1.05E+01 | crashed |
| 5 | 8.85E+00 | 9.84E+00 | crashed |
| 6 | 8.43E+00 | 8.99E+00 | crashed |
| 7 | 8.64E+00 | 9.69E+00 | crashed |
| 8 | 8.70E+00 | 9.12E+00 | crashed |
| 9 | 8.91E+00 | 8.94E+00 | crashed |
| 10 | 8.76E+00 | 9.26E+00 | crashed |
| 11 | 8.62E+00 | 9.09E+00 | crashed |
| 12 | 8.53E+00 | 8.92E+00 | crashed |
| 13 | 8.44E+00 | 9.13E+00 | crashed |
| 14 | 8.52E+00 | 9.00E+00 | crashed |
| 15 | 8.54E+00 | 9.19E+00 | crashed |
| 16 | 8.56E+00 | 9.05E+00 | crashed |
| 17 | 8.65E+00 | 9.12E+00 | crashed |
| 18 | 8.62E+00 | 8.96E+00 | crashed |
| 19 | 8.66E+00 | 9.30E+00 | crashed |
| 20 | 8.66E+00 | 9.16E+00 | crashed |

Table A.3.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **cube-645** (1000045 equations)

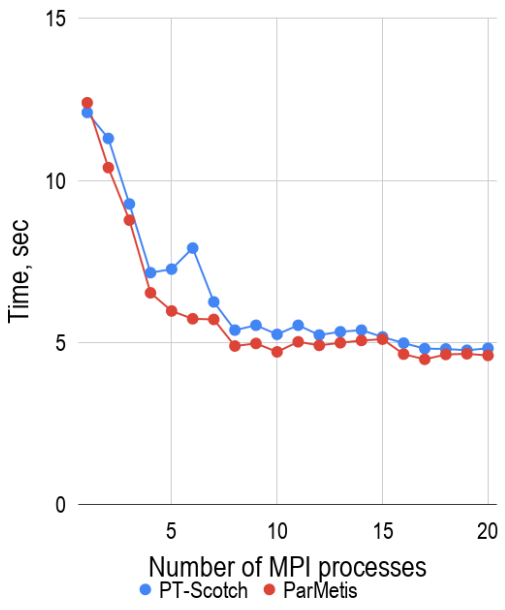
# B. Choice of Fill Reducing Reordering

place for figure B.1

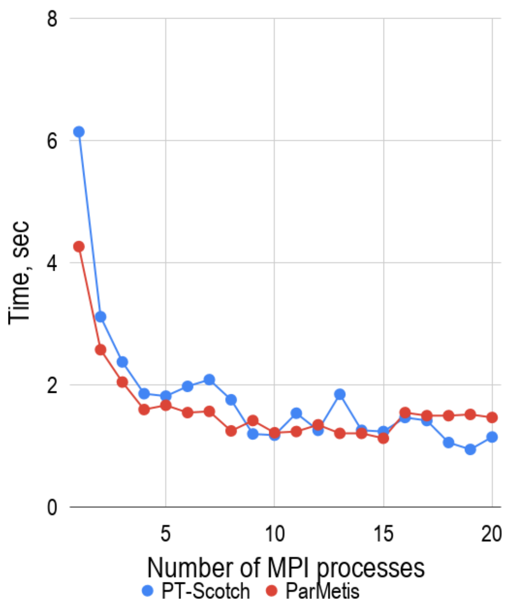
place for figure B.2



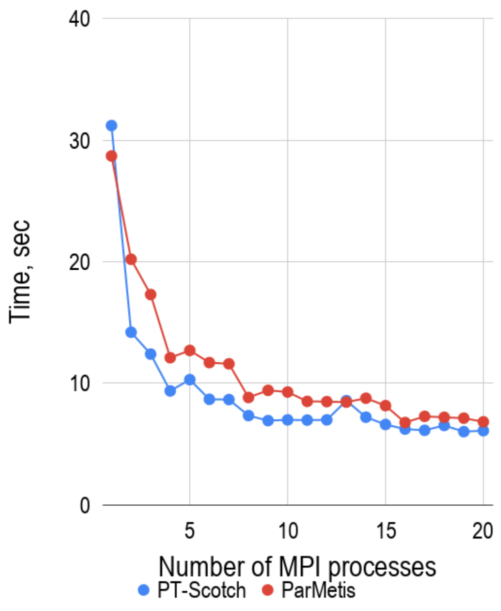
(a) cube-5



(b) cube-645

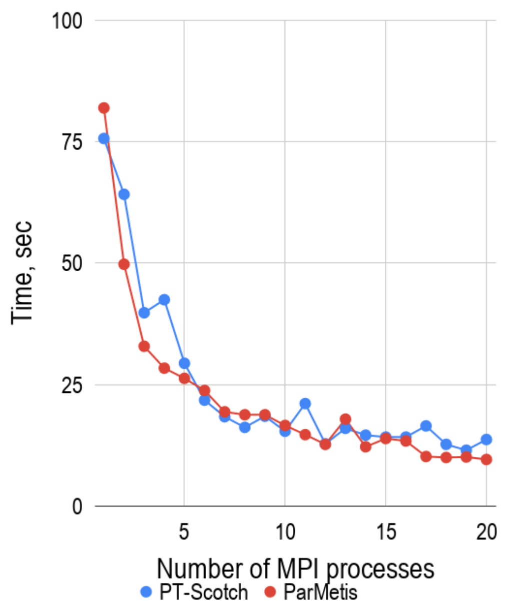


(c) cant

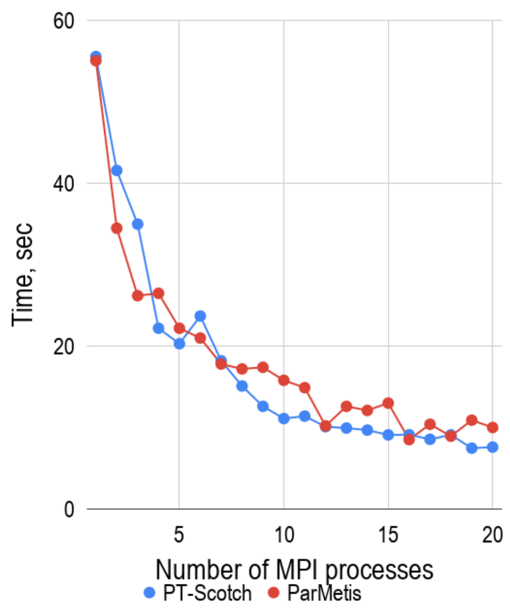


(d) memchip

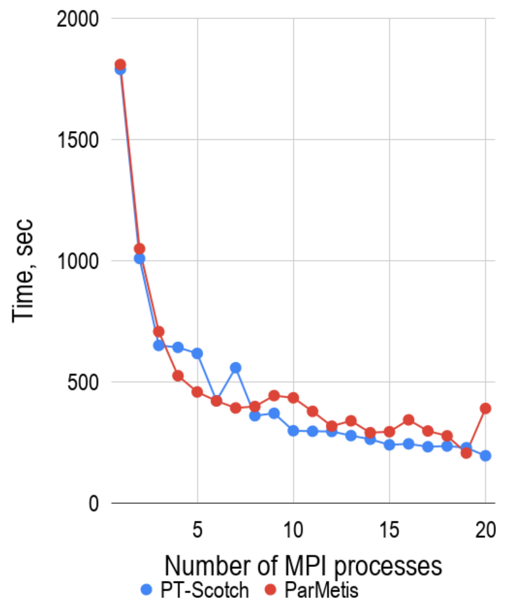
Figure B.1.: Comparison of different fill-reducing algorithms



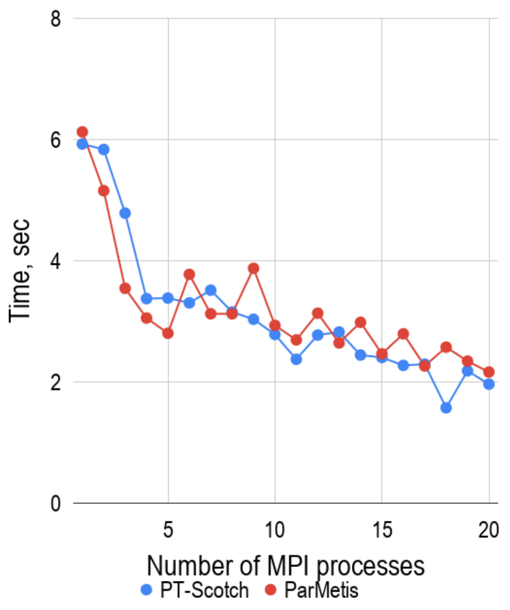
(a) torso3



(b) consph



(c) CurlCurl_3



(d) x104

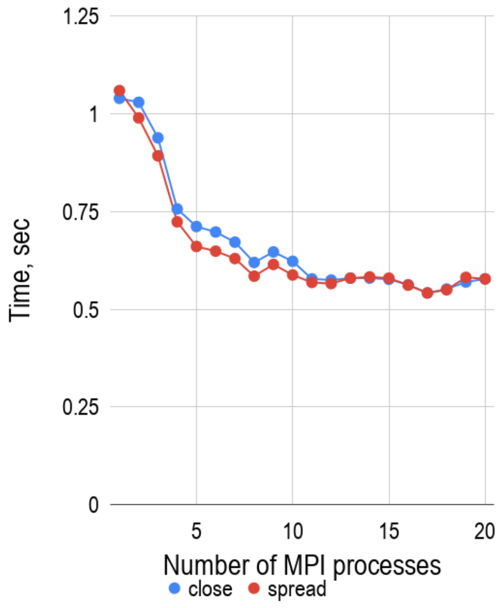
Figure B.2.: Comparison of different fill-reducing algorithms

# C. MUMPS: Process Pinning

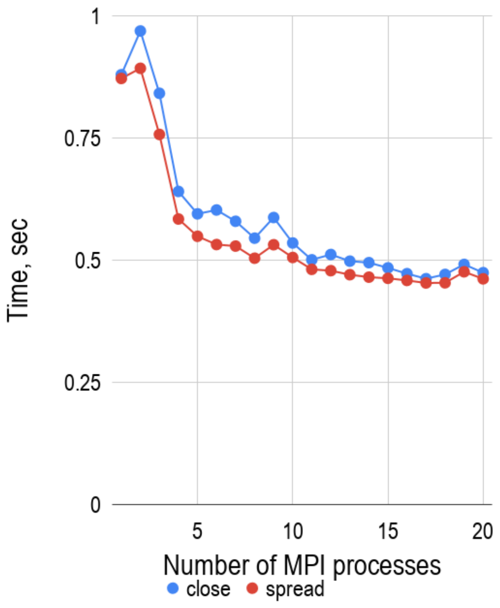
replace cube-64 by Geo matrix

place for figure C.1

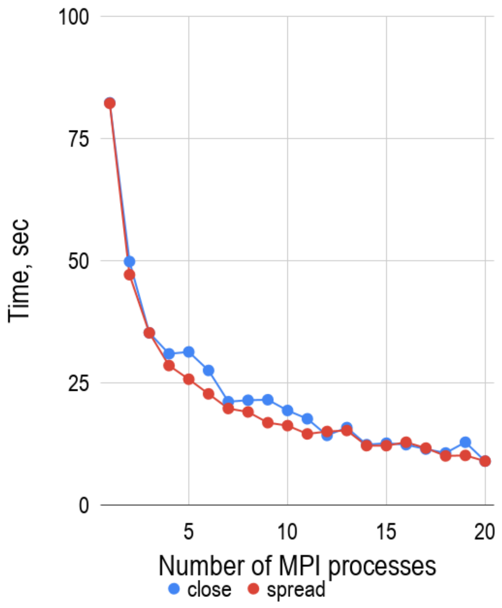
place for figure C.2



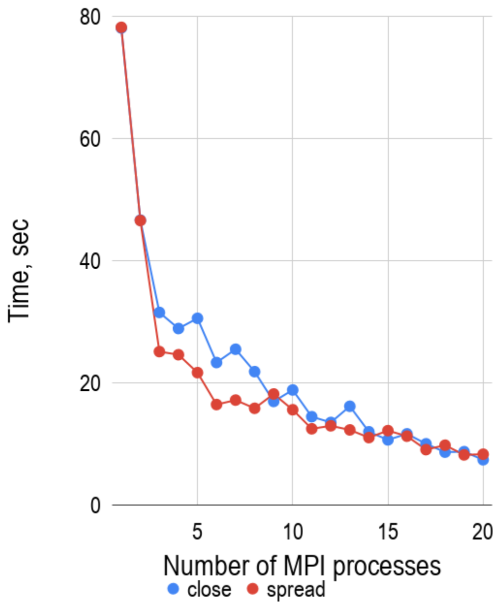
(a) HW1 - cube-64



(b) HW2 - cube-64

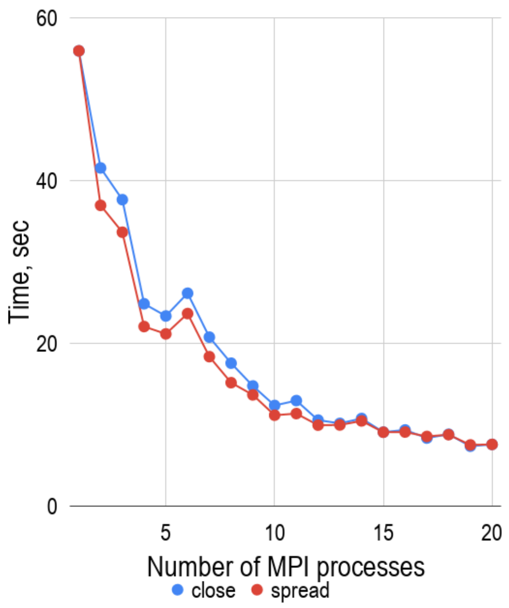


(c) HW1 - torso3

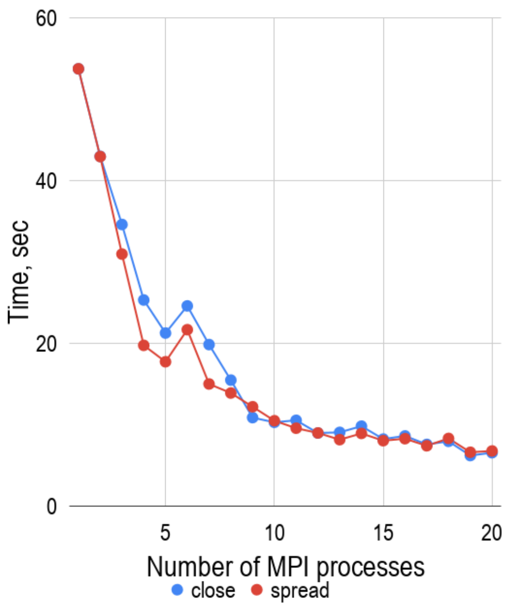


(d) HW2 - torso3

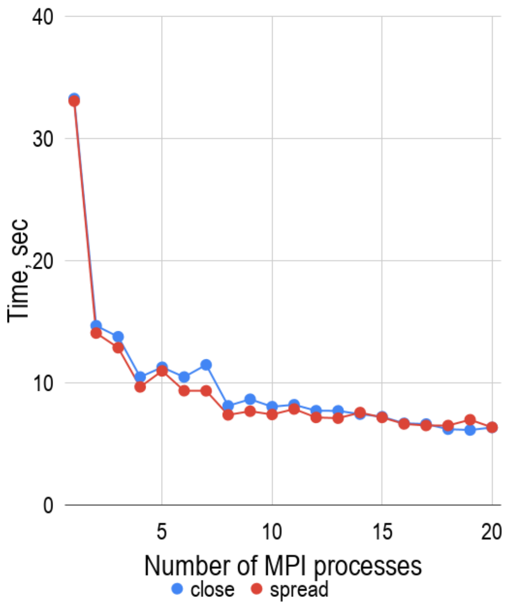
Figure C.1.: Comparison of *close* and *spread* pinning strategies



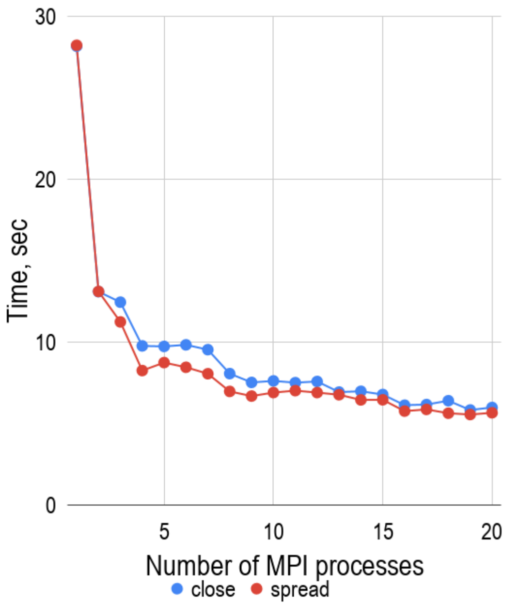
(a) HW1 - consph



(b) HW2 - consph



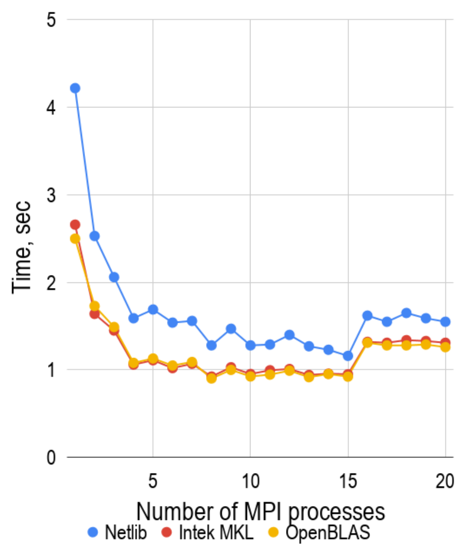
(c) HW1 - memchip_3



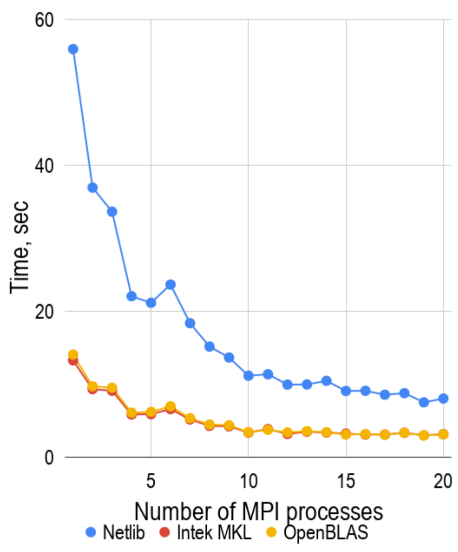
(d) HW2 - memchip

Figure C.2.: Comparison of *close* and *spread* pinning strategies

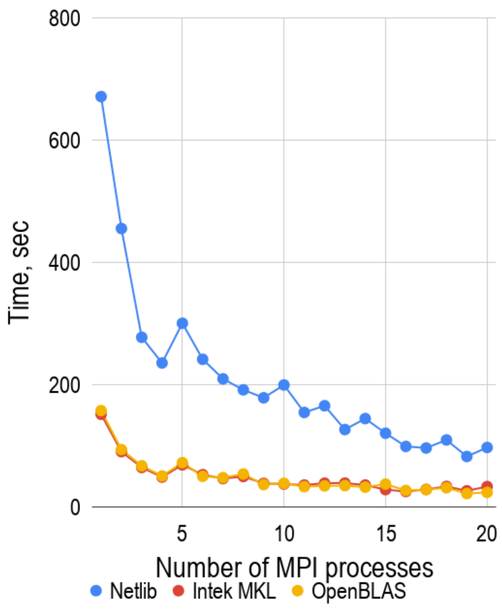
# D. Choice of BLAS Library



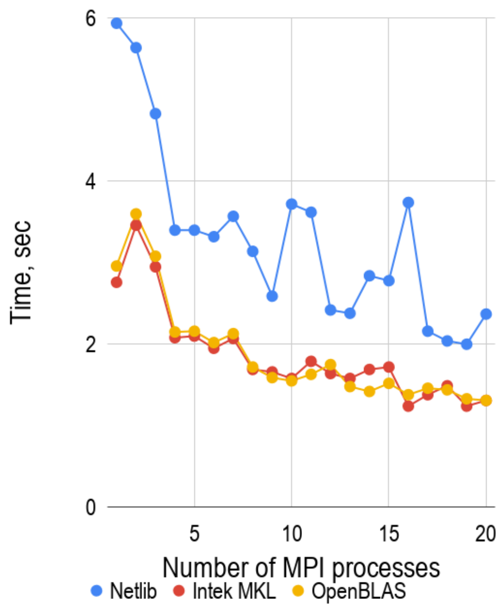
(a) cant



(b) consph

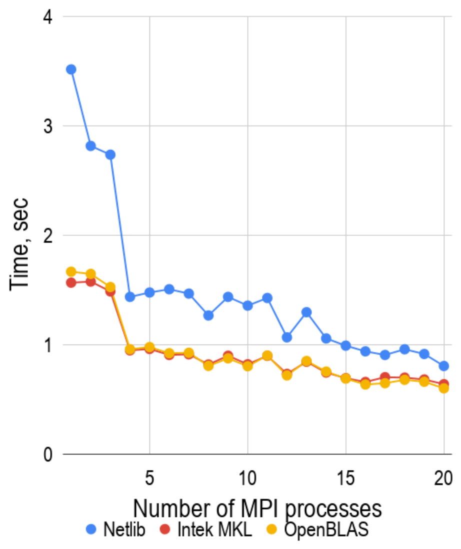


(c) PFlow_742

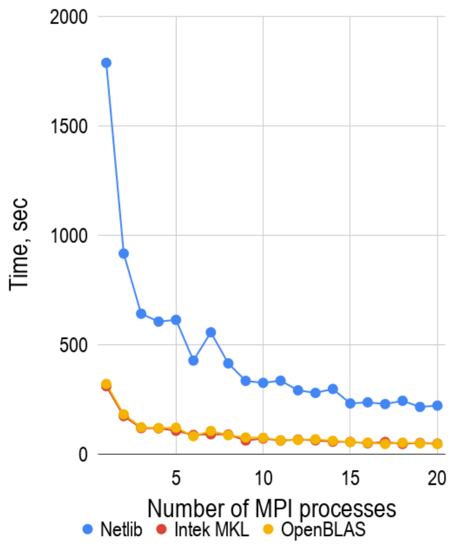


(d) x104

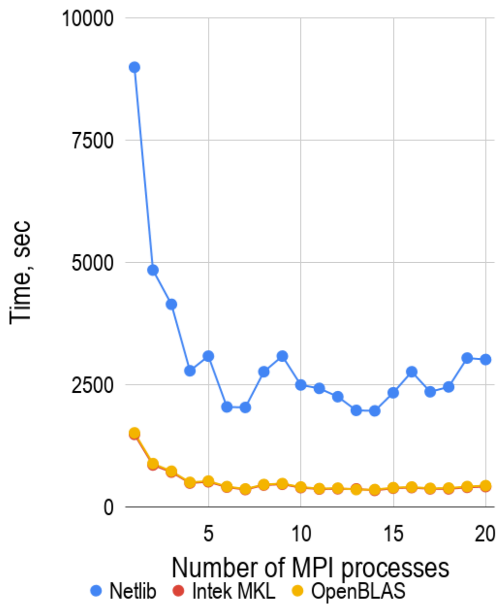
Figure D.1.: MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets



(a) pkustk10



(b) CurlCurl_3



(c) Geo_1438

Figure D.2.: MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets

# List of Figures

# List of Tables