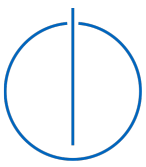
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Computational Science and Engineering

# Configuration of a linear sparse solver for a linear implicit time integration method and application of non-blocking MPI communication in parts of thermo-hydraulic computations for efficient data transfer

**Ravil Dorozhinskii**



# Nomenclature

ATHLET Analysis of THermal-hydraulics of LEaks and Transients

NUT Numerical Toolkit

PETSc Portable, Extensible Toolkit for Scientific Computation

MPI Message Passing Interface library

OpenMP Open Multi-Processing library

BLAS Basic Linear Algebra Subprograms

LAPACK Linear Algebra Package

ScaLAPACK Scalable Linear Algebra Package

GEMM General Matrix-matrix Multiplication (BLAS subroutine)

TRSM Triangular Solver with Multiple right-hand sides (LAPACK subroutine)

GETRF General Triangular Factorization (LAPACK subroutine)

NUMA non-uniform memory access

MUMPS MUltifrontal Massively Parallel sparse direct Solver

WSMP Watson Sparse Matrix Package

PDE Partial Differential Equation

SPD Symmetric Positive Definiten

PE Processing Element

GRS Die Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH

LRZ Leibniz-Rechenzentrum (Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities)

HW1 Hardware installed on the GRS cluster

HW2 Hardware installed on a LRZ CoolMUC-2 Linux cluster

# Contents


**1. Introduction** **1**

**2. Overview of ATHLET and NuT software** **3**
2.1. ATHLET . . . . . 3
2.2. NuT . . . . . 6
2.3. ATHLET-NuT coupling . . . . . 6

**3. Problem Statement** **9**

**4. Experimental Setup and Matrix Sets** **12**

**5. Overview of Sparse Linear Solver Types** **16**
5.1. Iterative Methods . . . . . 16
5.1.1. Theory Overview . . . . . 16
5.1.2. Parallelization Aspects . . . . . 17
5.1.3. Preconditioners . . . . . 20
5.2. Direct Sparse Methods . . . . . 22
5.2.1. Theory Overview . . . . . 22
5.2.2. Parallelization Aspects . . . . . 30
5.2.3. Numerical Pivot Handling . . . . . 38
5.3. Conclusion . . . . . 38

**6. Selection of a Sparse Direct Linear Solver Library** **40**

**7. Configuration of MUMPS library** **47**
7.1. Review of MUMPS Library . . . . . 47
7.2. Choice of Fill Reducing Reordering . . . . . 50
7.3. MUMPS: Process Pinning . . . . . 55
7.4. Choice of BLAS Library . . . . . 61
7.5. MPI-OpenMP Tuning of MUMPS Library . . . . . 68
7.6. Conclusion . . . . . 74
7.7. Recommendations . . . . . 79

**8. Improvement of ATHLET-NuT Communication** **81**
8.1. Jacobian Matrix Compression . . . . . 81
8.2. Accumulator Concept . . . . . 86
8.3. Benchmark and Test Data . . . . . 88
8.4. Results . . . . . 91
8.5. Conclusion . . . . . 95

**Appendices** **97**

**A. Sparsity Patterns of Matrix Sets** **98**

**B. Choice of a Sparse Direct Solver Library** **100**

**C. Choice of Fill Reducing Reordering** **102**

**D. MUMPS: Process Pinning** **105**

**E. Choice of BLAS Library** **108**

**List of Figures** **111**

**List of Tables** **114**

**Bibliography** **116**

# 1. Introduction

Nowadays, nuclear energy is one of the main sources of electricity. It comes from splitting atoms in a reactor which, as a result, heats water up to the point where it is converted into pressurized steam. In its turn, the steam rotates turbines which, finally, produces electricity. According to the recent estimations, thermal efficiency of modern nuclear power plants lies in the range of 35-45% which is comparable to conventional fossil fueled power plants [Jor19]. In spite of considerable initial investment, nuclear power plants have low operating costs and longevity which makes them particularly cost effective.

In recent years, nuclear power plants have become attractive means of power generation because of relatively low emission of carbon dioxide. As a result, the green house gase emissions to the atmosphere and thus the contribution of nuclear power plants to global warming is relatively less [Tim07].

Today, nuclear power plants generate almost 30% of the electricity produced in the European Union (EU). There are almost 130 nuclear reactors in operation in 14 EU countries, namely: Belgium, Bulgaria, Czech Republic, Finland, France, Germany, Hungary, Netherlands, Romania, Slovakia, Slovenia, Spain, Sweden, and the United Kingdom [Eur18].

The main problem assosiated with nuclear power is radioactive waste which is extremely dangerous for people and environment and has to be carefully looked after for several thousand years after utilization. Any accident in a plant can lead to grave consequences at a scale similar to Chernobyl disaster. For this reason, nuclear safety is one of the most important topics in this area. It demands a huge amount of testing and analysis to be performed before and during operation of a nuclear power plant in order to predict any possiblity of unwanted outcomes and devise preventive measures against such accidents. The topic has become even more prominent after 2011 Fukushima accident. In response to the disaster, numerous stress tests were conducted to measure the ability of the EU nuclear industry to withstand any kind of natural disaster [Eur18].

Since 1977, Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) has been the main

German scientific research institute in the field of nuclear safety and radioactive waste management [Ges]. Today, the organization carries out advanced research and analysis in the field of reactor safety, radioactive waste management as well as radiation and environmental protection [Ges]. Due to the inability to create various nuclear accident test scenarios, which by their very nature could be catastrophic, GRS develops and provides numerous simulation software products to cope with this problem. A short description of the main software packages developed by GRS is provided in table 1.1.

| Name | Description |
|---|---|
| ATHLET | Thermohydraulic safety analyses for the primary circuit of LWRs |
| ATHLET-CD | Analyses of accidents with core meltdown and fission product release for LWRs |
| ATLAS | Analysis simulator for interactive handling and visualisation of several computer codes |
| COCOSYS | Analyses of severe incidents in the containment of LWRs |
| DORT/TORT | Solution of time-dependant neutron transport equations for 2D/3D transients analyses |
| QUABOX/CUBBOX | 3-D neutron kinetics core model |
| SUSA | Uncertainty and sensitivity analyses |
| TESPA-ROD | Core rod code for design basis accidents |

Table 1.1.: A general overview of software developed by GRS [Ges]

The main focus of this study is dedicated to ATHLET software package as well as its Numerical Toolkit. The goal of the study is to identify the most compute-intensive parts of the ATHLET-NuT code and possibly accelerate its execution time.

# 2. Overview of ATHLET and NuT software

## 2.1. ATHLET

The thermal-hydraulic system code ATHLET (Analysis of THermal-hydraulics of LEaks and Transients) is developed by GRS for the analysis of the whole spectrum of operational conditions, incidental transients, design-basis accidents and beyond design-basis accidents without core damage anticipated for nuclear energy facilities [Ges16]. The code provides specific models and methods for the simulation of many types of nuclear power plants comprising current light water reactors (PWR, BWR, VVER, RBMK), advanced Generation III+ and IV reactors as well as Small Modular Reactors [Ges16].

Physical processes inside of hydraulic circuits of light-water reactors can be naturally described by a two-phase thermo-fluiddynamic model based on conservation equations of mass, momentum and energy for liquid and vapor.

1. Liquid mass
$$\frac{\partial((1-\alpha)\rho_l)}{\partial t} + \nabla((1-\alpha)\rho_l\vec{w}_l) = -\psi \tag{2.1}$$
2. Vapor mass
$$\frac{\partial(\alpha\rho_v)}{\partial t} + \nabla(\alpha\rho_v\vec{w}_v) = \psi \tag{2.2}$$
3. Liquid momentum
$$\frac{\partial((1-\alpha)\rho_l\vec{w}_l)}{\partial t} + \nabla((1-\alpha)\rho_l\vec{w}_l\vec{w}_l) + \nabla((1-\alpha)p) = \vec{F}_l \tag{2.3}$$
4. Vapor momentum
$$\frac{\partial(\alpha\rho_v\vec{w}_v)}{\partial t} + \nabla(\alpha\rho_v\vec{w}_v\vec{w}_v) + \nabla(\alpha p) = \vec{F}_v \tag{2.4}$$
5. Liquid energy
$$\frac{\partial\Big[(1-\alpha)\rho_l(h_l + \frac{1}{2}\vec{w}_l\vec{w}_l - \frac{p}{\rho_l})\Big]}{\partial t} + \nabla\Big[(1-\alpha)\rho_l\vec{w}_l(h_l + \frac{1}{2}\vec{w}_l\vec{w}_l)\Big] = -p\frac{\partial(1-\alpha)}{\partial t} + E_l \tag{2.5}$$

6. Vapor energy

$$\frac{\partial\left[\alpha\rho_v(h_v+\frac{1}{2}\vec{w}_v\vec{w}_v-\frac{p}{\rho_v})\right]}{\partial t}+\nabla\left[\alpha\rho_v\vec{w}_v(h_v+\frac{1}{2}\vec{w}_v\vec{w}_v)\right]=-p\frac{\partial\alpha}{\partial t}+E_v \tag{2.6}$$

7. Volume vapor fraction

$$\alpha=\frac{V_v}{V} \tag{2.7}$$

where $p$ - pressure of mixture, $\psi$ - mass source term, $\vec{F}$ - external composite force acted on a control volume, $E$ - external composite energy source term within a control volume, subscripts $l$ and $v$ denote liquid and vapor phases, respectively.

Spacial integration of the conservation equations, the ssystem 2.1 - 2.7, is performed on the basis of finite-volume method with using one dimensional formulation, figure 2.1.

place for figure 2.1

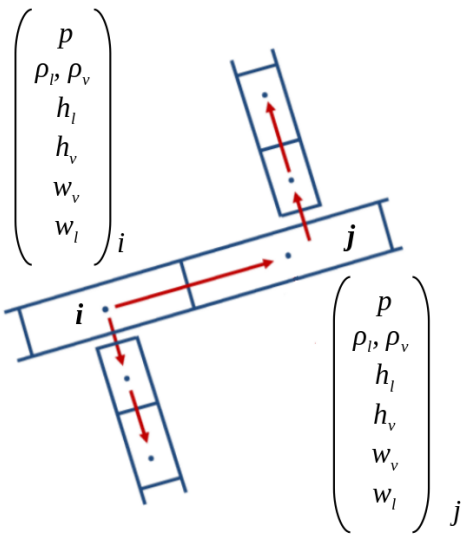



Figure 2.1.: ATHLET: one dimensional finite volume formulation of the problem

Finally, the system is transformed to a non-autonomous system of ordinary differential equations and expressed as an initial value problem, equation 2.8, after spatial finite-volume integration and many mathematical transformations with the aim of decoupling the initial system REF.

$$\frac{dy}{dt}=f(t,y),\ t_0\le t\le t_F\ y(t_0)=y_0 \tag{2.8}$$

where $y \in \mathbb{R}^N$ is a composite vector of variables, $f$ is a non-linear function such that $f : \mathbb{R} \times \mathbb{R}^N \supset \Omega \to \mathbb{R}^N$ .

Analysis of system 2.8 shows the problem is rather stiff and thus must to be solved with an implicit solver. Rosenbrock methods are a class of linear implicit methods which is capable of solving such stiff systems of ODEs efficiently. The methods replace non-linear systems with a sequence of linear ones, however, some stability and accuracy properties are usually lost [Blo+13]. An additional drawback of the methods is evaluation of the exact Jacobian at every time step which affects computational cost.

To decrease the cost and preserve sufficient accuracy of numerical integration, ATHLET, instead, uses a W-method of the third order. W-methods belong to the family of Rosenbrock methods, however, calculate the Jacobian matrix occasionally. The ATHLET developers spent much of their time and efforts to develop heuristics to identify instances of time when evaluation of the Jacobian must be performed. In other words, the algorithm can re-use the same Jacobian matrix approximation between steps with some partial matrix updates. However, when a hydraulic circuit state drastically changes due to transitivity, the evaluation of the full Jacobian is required.

place for figure 2.2

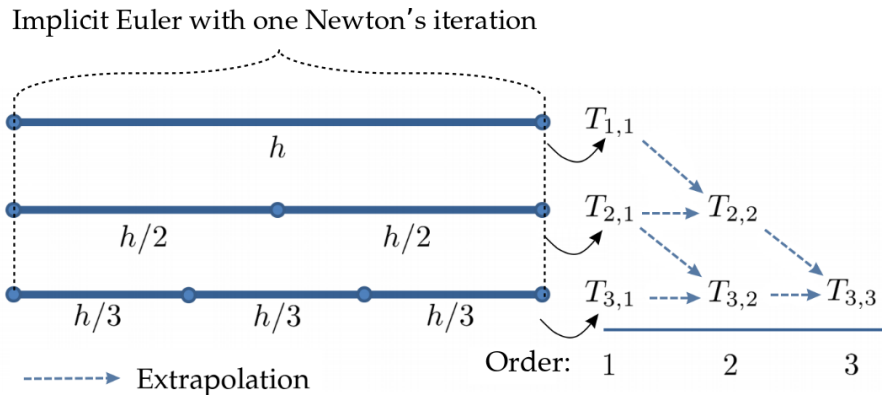



Figure 2.2.: A general view on the W-method solver implemented in ATHLET

In the general case, a step of the W-method method, implemented in ATHLET, can be viewed as a sequence of three stages in the following way. Each stage uses implicit Euler method with different sub-steps and exactly one Newton's iteration to evaluate the value of vector $y$ at the next integration step with different accuracy. Then, the obtained values are extrapolated, in order explained in figure 2.2, to achieve desired

order of integration. By and large, the algorithm can be expressed in a compact form of equation 2.9.

$$((h\gamma)^{-1}I - J)\Delta z_i^l = -h^{-1}z_i^l + f(t_0 + \tau_i h, y_0 + z_i^l) \tag{2.9}$$

where $\Delta z_i^l = z_i^{l+1} - z_i^l$, $z_i^l = y_i^l - y_{i-1}^l$, $J \approx \frac{\partial f}{\partial y}$ - approximation of Jacobian matrix, $l = 1, 2$ - Newton's iteration index, $i = 1, 2, 3$ - integration step index.

## 2.2. NuT

Numerical Toolkit, or just NuT, can be viewed as a container of various dense and sparse linear algebra subroutines which can run in parallel on distributed-memory machines. NuT design follows the paradigm of *Adapter/Wrapper* pattern which provides a uniform common interface for its services to various GRS simulation tools (outlined in table 1.1) and thus helps to achieve re-usability, flexibility and extensibility properties of the code.

Currently, NuT is based heavily on Portable, Extensible Toolkit for Scientific Computation, known as PETSc library. It is one of the most widely used parallel numerical software library [Wik18c]. It includes a large suite of parallel linear and nonlinear equation solvers as well as its software-infrastructure to handle computations on distributed-memory machines by means of Message Passing Interface (MPI) and specific data structures. Fortunately, though a careful selection of the design pattern, NuT can be easily extended to provide an extra service or an external library access which has not been implemented in PETSc yet.

## 2.3. ATHLET-NuT coupling

place for figure 2.3

Coupling of NuT with GRS tools is based on the client-server architecture where NuT acts as a server and the tools can be viewed as clients. Communication between two parts is done via MPI.

To provide a clear and concise external interface, NuT contains a client module called "NuT Plug-in". It can be considered as a socket, from the client side, using the analogy of Transmission Control Protocol (TCP). The plug-in hides all MPI calls to the sever

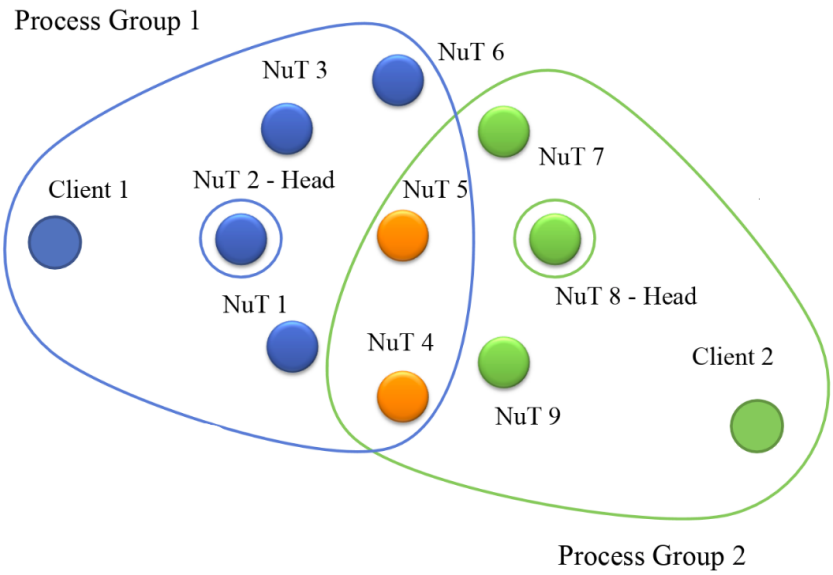



Figure 2.3.: NuT process groups

which considerably improves readability of the code.

In the general case, NuT allows multiple clients to work concurrently with the server. To handle the traffic, the library splits the default MPI communicator at start-up time of the application into appropriate process groups, as it is shown in figure 2.3.

The design of NuT allows sharing of some NuT-MPI processes among different process groups due to performance reasons i.e. finite number of processing units on hardware. To resolve possible deadlocks, each process group has its own representative, called the head. Each client has two views on its respective group which is achieved by means of distinct MPI communicators. The first communicator is responsible for client-head communication whereas the second one allows the client to talk to any NuT process within the group.

A general view of client-server communication looks like a 3-way handshake in the following way: a client sends a request to the head which is a signal to reserve all compute-units of the group for an upcoming task. Having possessed the resources and prepared them for a specific service, the head notifies the client about resource acquisition and the entire process group waits for data. Afterwards, the client sends

data either to a specific NuT-process or to the entire group using the second communicator and waits for a result of the service. In the current implementation of NuT, the communication between client and server is synchronous i.e. the client gets blocked while waiting for a result from the server.

As an example, figure 2.4 represents a general view of ATHLET-NuT coupling where ATHLET is responsible for marching of the numerical integration solver whereas NuT computes solutions of systems 2.9.

ask for an updated image

place for figure 2.4

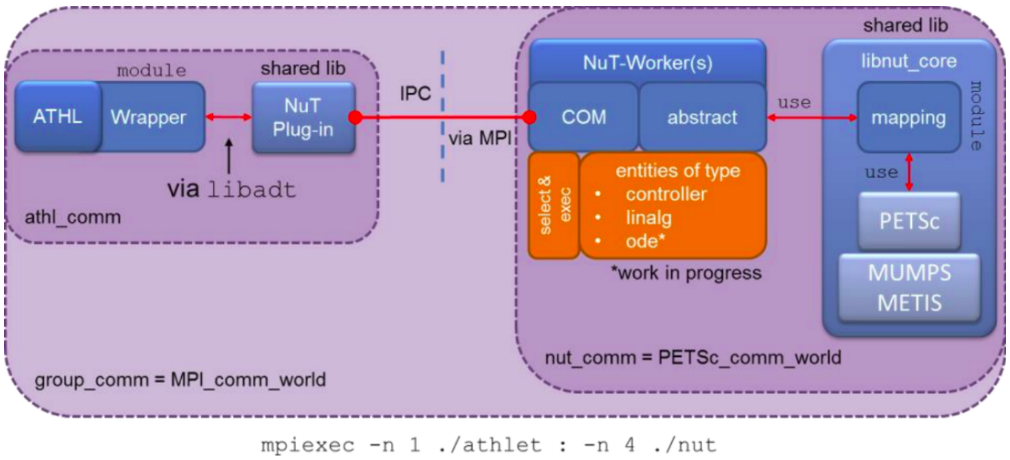



Figure 2.4.: ATHEL-NuT software coupling

Partial and full Jacobian matrix updates derived from finite differences are computed on the client side since only the client has the access to function $f(y)$, equation 2.8. Due to decoupling of the underlying system of PDEs and specifics of finite volume discretization, Jacobian matrix is rather sparse and, therefore, ATHLET uses a Jacobian matrix compression algorithm, described in section 8.1, to reduce the amount of Jacobian column evaluations. Having computed a matrix column, ATHLET immediately broadcasts it to its entire NuT process group by means of 3-way handshake mechanism as described above. It is worth mentioning that this approach allows to circumvent potential memory limits on the client side and thus store the entire sparse Jacobian matrix in a distributed fashion on the server. In other words, ATHLET never holds the entire Jacobian matrix in its memory; conversely, the matrix is distributed across multiple NuT processes according to block-row distribution induced by PETSc. In turn, NuT is waiting for the entire Jacobian matrix information from ATHLET and starts solving systems 2.9 right after the corresponding request from the client.

# 3. Problem Statement

Integration of a system of ODEs by means of W-methods can be considered as a solution of a sequence of linear systems from another point of view. Equations 2.9 can be rewritten in a form 3.1, after grouping both the right- and left-hand sides in a single matrix and vector, respectively.

$$A_i \Delta z_i^l = b_i^l \tag{3.1}$$

where $A = ((h\gamma)^{-1} I - J)$ is a $\mathbb{R}^N \times \mathbb{R}^N$ non-singular sparse matrix, $\Delta z_i^l$ and $b_i^l$ are $\mathbb{R}^N$ vectors.

According to the integration scheme, figure 2.2, and definition of the method, each step of numerical integration requires to solve 6 linear systems with 3 distinct matrices, resulting from the Jacobian matrix by the corresponding shifts of the main diagonal. Therefore, the computational burden of the W-method mainly lies in solving of sparse linear systems.

There exist two families of linear sparse solvers, namely: iterative and direct sparse methods. In the general case, execution time of any algorithm, regardless of the solver family, is bounded by $O(N^2)$ complexity due to matrix sparsity, where $N$ is number of equations in the system. However, the constant in front of the factor $N^2$ can vary significantly across the methods which explains the differences in execution time. Additionally, it is important to mention the families use absolutely different approaches for solution of sparse linear systems and thus posses different numerical properties. Among all the properties, there are some which are particularly important for efficient execution of W-methods, namely:

- robustness (or numerical stability) with respect to ill-conditioner problems
- numerical accuracy
- parallel efficiency, with emphasis on strong scaling

These above mentioned properties can be treated as non-functional requirements for a sparse linear solver for efficient numerical time integration.

Finding solutions of sparse linear systems is a well-known and commonly occurring problem in the field of scientific computing and, therefore, numerous implementations of different kind of linear solvers exist. However, the NuT project imposes some extra constrains due to the design philosophy adopted by GRS:

- open-source license
- direct interface to PETSc

In this study, we are primarily concerned with selection and configuration of a linear sparse solver that can cover all requirements listed above.

This report is organized as follows. Chapter 4 provides full information about the experimental setup and matrix sets used in the study. Chapters BRA and BRA explain theory and some parallelization aspects of iterative and sparse direct methods, respectively, using a well-known representative-algorithm of each type as an example. In chapter BRA, we give a short summary and conclude which type of sparse linear solvers is the best suited for time integration governed by the W-method. In chapter BRA, we represent a list of solvers, according to the chosen type, available at the time of writing, and perform initial tests with the aim of finding the fastest solver of the corresponding type. From chapter BRA onwards, we focus on configuration of a specific solver to reduce its execution time. Chapter BRA sums up overall results of the performed configuration and, in chapter BRA, we give some recommendations to the ATHLET users about which solver parameters are better to use for a specific matrix size: small, medium, large.

An additional topic, considered in this study, is improvement of ATHLET-NuT communication during Jacobian matrix transfer. As it was described in section 2.3, ATHLET, the client, transfers Jacobian matrix in a column-wise fashion. NuT, the server, treats each column transfer as a service and, therefore, each transfer passes through a 3-way handshake. Moreover, it is important to mention one more time, due to the current implementation of ATHLET-NuT coupling, the client-server communication is blocking. In other words, ATHLET gets blocked till completion of a column transfer.

The main goal of Jacobian matrix compression, described in section 8.1, is to minimize the number of perturbations of non-linear function $f(y)$, equation 2.8, derived from

finite differences. Additionally it allows to reduce the amount of column transfers as well. Therefore, it improves overall application performance from both computational and communication point of view. However, there are still some aspects to be considered.

Due to specifics of matrix compression algorithm, described in section 2.3, column length is decreasing between the first and last columns of compressed Jacobian matrix form which, as a result, leads to unequal MPI message sizes.

In this part of the study, we introduce a concept called *accumulator* which allows to transfer a compressed Jacobian matrix in equal chunks. This approach potentially solves three important problems at once. First of all, *accumulator* can help to get rid of small MPI messages and thus improves network bandwidth utilization. Secondly, it helps to reduce the amount of synchronizations between the client and server and, therefore, improves operation of NuT as the server. Lastly, it allows to apply non-blocking MPI communication on the client side and thus overlap Jacobian matrix transfer with its computations.

In section 8.1, we briefly describe the Jacobian matrix compression algorithm and the resulting ATHLET-NuT communication problem. In section 8.2, we present and describe the algorithm which is supposed to resolve the problem. Section 8.3 provides a description of developed benchmarks and test data. Then, we focus and explain obtained results in section 8.4. Finally, in section 8.5, we provide a general conclusion of the performed study and summarize the results one more time.

# 4. Experimental Setup and Matrix Sets

In this study, two matrix sets were used: GRS and SuiteSparse. In our case, the SiteSparse matrix set was, in fact, few matrices downloaded from SuiteSparse Matrix Collection [DH11], [DGL89]. 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.

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.

The main matrix properties as well as matrix sparsity patterns are shown in tables 4.1, 4.2 and appendix A.

| Name | n | nnz | nnz / n | Approximate Condition Number |
|---|---|---|---|---|
| pwr-3d | 6009 | 32537 | 5.4147 | 1.019e+07 |
| cube-5 | 9325 | 117897 | 12.6431 | 1.592e+09 |
| cube-64 | 100657 | 1388993 | 13.7993 | 7.406e+08 |
| cube-645 | 1000045 | 13906057 | 13.9054 | 6.474e+08 |
| k3-2 | 130101 | 787997 | 6.0568 | 1.965e+15 |
| k3-18 | 1155955 | 7204723 | 6.2327 | 1.947e+12 |

Table 4.1.: GRS matrix set

Approximations of condition numbers were computed by means of Rayleigh–Ritz procedure [09]. GMRES solver, with 1000 iteration steps, was applied for un-preconditioned systems to generate a Krylov subspace for each matrix. Then, the resulting Hessenberg matrices were used for approximating eigenspaces and the corresponding eigenvalues. The approximations should be treated as lower bound since the algorithm overestimates the smallest eigenvalue.

The objective of this study is to find and configure a sparse linear solver which can fulfill all requirements listed above for the GRS matrix set. It is worth pointing out, as it

| Name | n | nnz | nnz / n | Approximate Condition Number | Field |
|---|---|---|---|---|---|
| cant | 62451 | 4007383 | 64.1684 | 5.082e+05 | 2D/3D Problem |
| consph | 83334 | 6010480 | 72.1251 | 2.438e+05 | 2D/3D Problem |
| CurlCurl_3 | 1219574 | 13544618 | 11.1060 | 2.105e+05 | Model Reduction Problem |
| Geo_1438 | 1437960 | 63156690 | 43.9210 | 4.677e+05 | 2D/3D Problem |
| memchip | 2707524 | 13343948 | 4.9285 | 1.305e+07 | Circuit Simulation Problem |
| PFlow_742 | 742793 | 37138461 | 49.9984 | 5.553e+06 | 2D/3D Problem |
| pkustk10 | 80676 | 4308984 | 53.4110 | 5.589e+02 | Structural Problem |
| torso3 | 259156 | 4429042 | 7.0903 | 2.456e+03 | 2D/3D Problem |
| x104 | 108384 | 8713602 | 80.3956 | 3.124e+05 | Structural Problem |

Table 4.2.: SuiteSparse matrix set

was mentioned in section 2.1, ATHLET performs many mathematical transformations upon the original system and, finally, generates an approximation of a Jacobian matrix. For that reason, one can assume that GRS matrix set can be structurally different from matrices coming naturally from finite-volume, finite-elements discretization or optimization problems. Therefore, SuiteSparse matrix set was used, from time to time, to examine this statement.

Tow different hardware were available for this study. The first machine was the GRS cluster (HW1) which was the main target. Additionally, LRZ CoolMUC-2 Linux cluster (HW2) was used every time when we got some ambiguous results to check whether a problem was hardware, software or algorithm specific. Table 4.3 shows a single node specification of both machines.

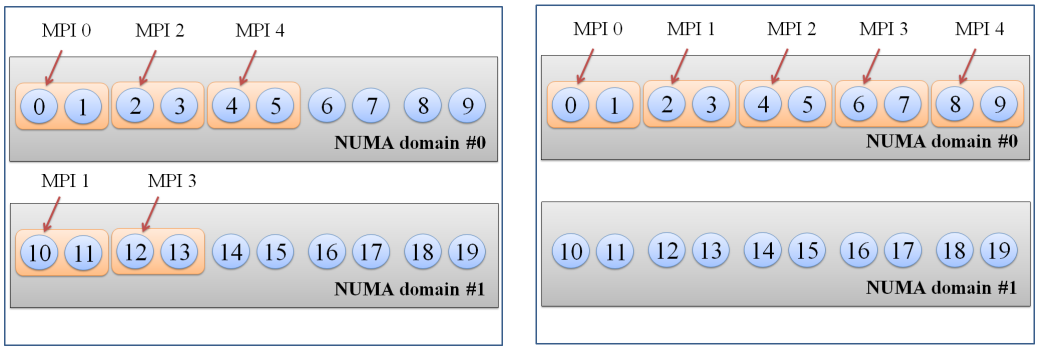
For this study, OpenMPI implementation of the MPI standard was used because of its open-source license and comprehensive documentation. The library has many options for processes pinning which was quite important for of this study.

To make process pinning explicit and deterministic, a python script was developed to generate rank-files automatically 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*

| | 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 4.3.: Hardware specification




(a) *Spread* mode

(b) *Close* mode

Figure 4.1.: An example of process pinning of 5 MPI processes with 2 OpenMP threads per rank in case of HW1 hardware

and *close*. Given a certain number of ranks, *spread* mode tries to distribute them as spread as possible across multiple available NUMA domains in a round-robin fashion.

In contrast to *spread* strategy, *close* one groups ranks as close as possible to keep the maximum number of ranks within a single NUMA domain. Figure 4.1 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).

In this study, PETSc 3.10 and OpenMPI 3.1.1 libraries were chosen and compiled with Intel 18.2 compiler.

# 5. Overview of Sparse Linear Solver Types

## 5.1. Iterative Methods

### 5.1.1. Theory Overview

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 $U_m$.

$$\min_{x \in U_m} ||Ax - b||^2 \tag{5.1}$$

We can consider a solution vector $x$ in the subspace $U_m$ in a form $x = U_m y$. Thus, equation 5.1 can be written as following:

$$\min_{x \in U_m} ||AU_m y - b||^2 \tag{5.2}$$

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 [SS86] used Arnoldi process for constructing an $l_2$-orthogonal basis. As the results equation 5.2 can be written in the following form:

$$\min_{x \in U_m} ||U_{m+1} H_{m+1,m} y - ||b||u_1||^2 = \min_{x \in U_m} ||H_{m+1,m} y - ||b||e_1||^2 \tag{5.3}$$

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,

$$\min_{x \in K_m} ||Ax - b||^2 = \min_{x \in U_m} ||Q^T R y - ||b||e_1||^2 = \min_{x \in U_m} ||\begin{pmatrix} R_m \\ 0 \end{pmatrix} y - \begin{pmatrix} \tilde{b_m} \\ \tilde{b_{n-m}} \end{pmatrix}||^2 \tag{5.4}$$

Given 5.4, we can compute the solution as following:

$$R_m y = \tilde{b_m} \tag{5.5}$$

$$x_m = U_m y \tag{5.6}$$

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.

### 5.1.2. Parallelization Aspects

We can see that some operations, for example 5.6, can be efficiently done in parallel. However, operatiopns like sparse triangular solve 5.5 can introduce some effect on strong scaling behavior. Figure 5.1 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 [Jos+98].

place for figure 5.1

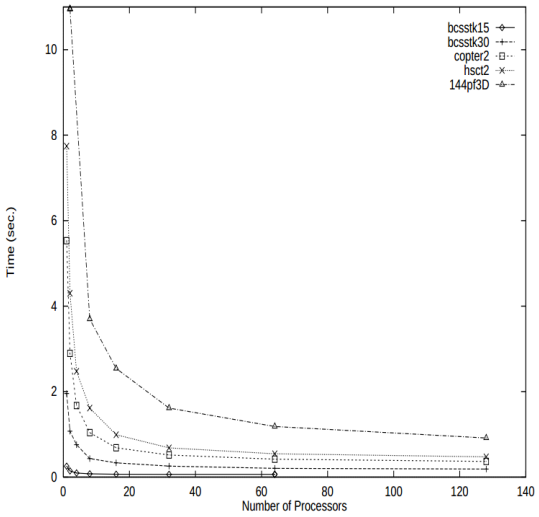



Figure 5.1.: Performance of sparse triangular solve [Jos+98]

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 5.5 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 5.2 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 4, 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

4.1. 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 5.2

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 [CCV18]. 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 [CCV18].

place for figure 5.2

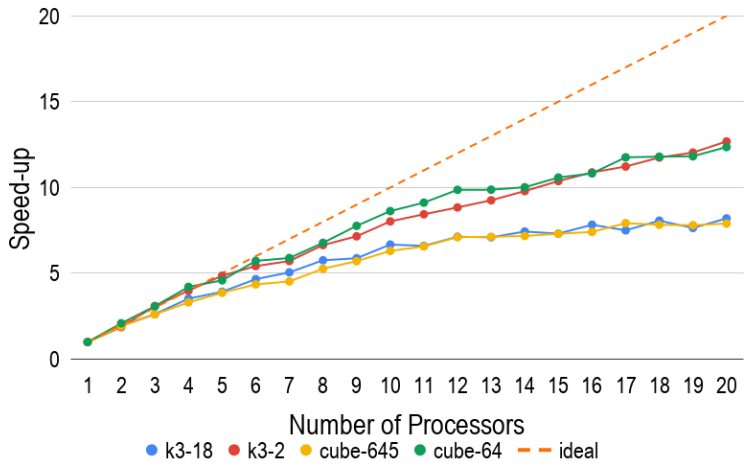



Figure 5.2.: GMRES strong scaling speed-up

### 5.1.3. Preconditioners

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 5.7 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.

$$||e^i||_A \leq 2(\frac{\sqrt{k}-1}{\sqrt{k}+1})^i||e^0||_A \tag{5.7}$$

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 **??**. 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 transformation of **??** can be done from the left side 5.8 or from the right one 5.9.

$$PAx = Pb \tag{5.8}$$

$$AP(P^{-1}x) = b \tag{5.9}$$

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 **??** 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 choose correct parameters for a certain preconditioning algorithm. It can become a challenge to fulfill all requirements 1, 2, 3 mentioned above.

Table

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

Table with comparisons of different preconditioning for our test cases

It interesting to notice that Gupta, Koric, and George came to approximately the same results working on their set of matrices in their work [GKG09]. 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 [GKG09]. 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 **??**.

## 5.2. Direct Sparse Methods

### 5.2.1. Theory Overview

Direct sparse methods combine main advantages of direct dense 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 $O(n^2)$ [Wu12]. 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 Duff and Reid in their work [DR83]. Multifrontal method is an improved extension of a frontal method [IB70] 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 [PT01].

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 5.10. It allows us to focus on the Cholesky factor $L$ and its sparsity pattern only.

$$A = LDL^T \tag{5.10}$$

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 5.3

Figure 5.3 shows an illustrative example of a sparse matrix and its Cholesky factor from [Liu92]. 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 5.11. It is

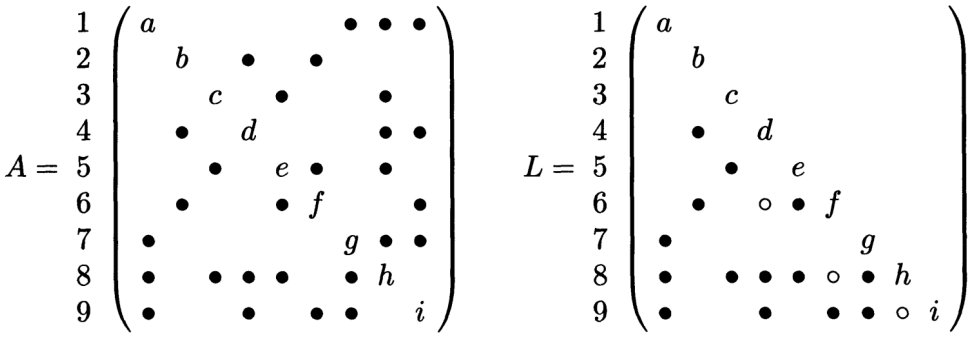


Figure 5.3.: An example of a sparse matrix and its Cholesky factor [Liu92]

worth pointing out the definition 5.11 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 [Liu92] proposed by Liu.

$$p = min(i > j | l_{ij} \neq 0) \tag{5.11}$$

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 5.11 we can build the corresponding elimination tree as it is shown in figure 5.4.

place for figure 5.4

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 5.12

.

$$F_j = Fr_j + \hat{U}_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} & a_{j,i_2} & \dots & a_{j,i_r} \\ a_{i_1,j} & & & & \\ a_{i_1,j} & & & & \\ \vdots & & & 0 & \\ a_{i_r,j} & & & & \end{bmatrix} + \hat{U}_j \tag{5.12}$$

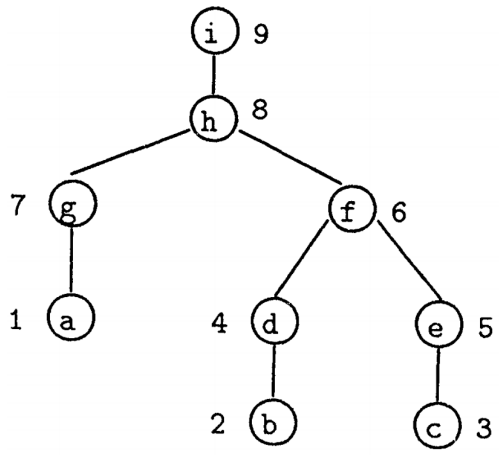



Figure 5.4.: The elimination tree for the matrix example in Figure 5.3 [Liu92]

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:

$$\hat{U}_j = - \sum_{k \in T[j]-j} \begin{bmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_1,k} \end{bmatrix} \begin{bmatrix} l_{j,k} & l_{i_1,k} & \dots & l_{i_1,k} \end{bmatrix} \tag{5.13}$$

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 $\hat{U}_j$.

$$A = \begin{bmatrix} B & V^T \\ V & C \end{bmatrix} = \begin{bmatrix} L_B & 0 \\ VL_B^{-T} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & C - VB^{-1}V^T \end{bmatrix} \begin{bmatrix} L_B^T & L_B^{-1}V^T \\ 0 & I \end{bmatrix} \tag{5.14}$$

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

$$B = L_B L_B^T \tag{5.15}$$

The Schur complement from equation 5.14 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:

$$-VB^{-1}V^T = -(VL_B^{-T})(L_B^{-1}V^T) = -\sum_{k=1}^{j-1} \begin{bmatrix} l_{j,k} \\ \vdots \\ l_{n,k} \end{bmatrix} \begin{bmatrix} l_{j,k} & \dots & l_{n,k} \end{bmatrix} \tag{5.16}$$

As it can be easily seen that equations 5.16 and 5.13 are identical. The difference is that equation 5.13 exploits sparsity of the corresponding row and column of $L$ and thus masks unnecessary information.

We can also notice from equation 5.14 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:

$$\hat{F}_j = \begin{bmatrix} l_{j,j} & \dots & 0 \\ \vdots & I & \\ l_{i_r,j} & & \end{bmatrix} \begin{bmatrix} 1 & \dots & 0 \\ \vdots & U_j & \\ 0 & & \end{bmatrix} \begin{bmatrix} l_{j,j} & \dots & l_{i_r,j} \\ \vdots & I & \\ 0 & & \end{bmatrix} \tag{5.17}$$

where sub-matrix $U_j$ represents the full update from all descendants of node $j$ and node $j$ itself. Equation 5.18 express the sub-matrix $U_j$ in a vector form.

$$\hat{U}_j = -\sum_{k \in T[j]} \begin{bmatrix} l_{i_1,k} \\ \vdots \\ l_{i_1,k} \end{bmatrix} \begin{bmatrix} l_{i_1,k} & \dots & l_{i_1,k} \end{bmatrix} \tag{5.18}$$

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 5.5.

place for figure 5.5

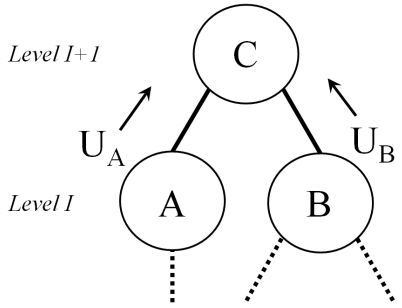



Figure 5.5.: Information flow of the multifrontal method

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 5.18 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 [Liu92].

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 [Liu92] 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.

$$R = \begin{bmatrix} p & q \\ u & v \end{bmatrix}, S = \begin{bmatrix} w & x \\ y & z \end{bmatrix} \tag{5.19}$$

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

$$K = R \boxplus S = \begin{bmatrix} p & q & 0 \\ u & v & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} w & 0 & x \\ 0 & 0 & 0 \\ y & 0 & z \end{bmatrix} = \begin{bmatrix} p+w & q & x \\ u & v & 0 \\ y & 0 & z \end{bmatrix} \tag{5.20}$$

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:

$$F_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} & a_{j,i_2} & \dots & a_{j,i_r} \\ a_{i_1,j} & & & & \\ a_{i_1,j} & & & & \\ \vdots & & & 0 & \\ a_{i_r,j} & & & & \end{bmatrix} \boxplus U_{c_1} \boxplus \dots \boxplus U_{c_s} \tag{5.21}$$

where $c_1$, $c_2$, $\dots$ $c_n$ 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 [Liu92]. 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 5.6 shows an example of posrordering applied to the elimination tree of the matrix from figure 5.3. The results of this can be see in figure 5.7 where consecutive *push* and *pop* operations are efficiently used during factorization and thus simplify the program logic.

place for figure 5.6

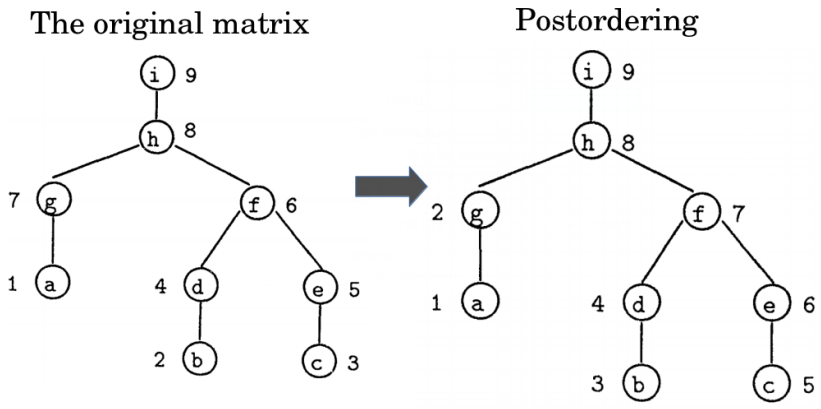



Figure 5.6.: An example of matrix postordering from [Liu92]

place for figure 5.7

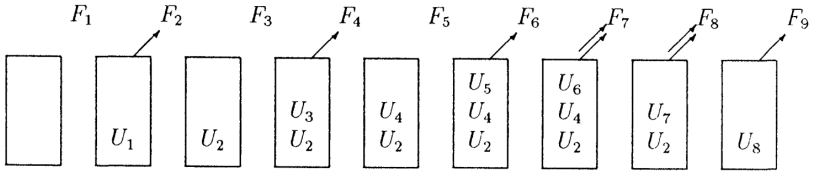



Figure 5.7.: The stack contents for the postordering [Liu92]

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 [Liu86] [Liu88]. Some schemes,

for example elimination tree rotations [Liu88], 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 [Gup00].

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 **??** and **??**).

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 5.8 shows a postordered matrix $A$ and its Cholesky factor $L$ as well as the corresponding supernodal elimination tree.

place for figure 5.8

Equation 5.22 expresses the building process of a frontal matrix of a supernode. In contrast to 5.21, 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.

It should be mentioned there exist more sophisticated variants of supernodes. Most of the time, it intends to improve efficiency of the algorithm. Liu pointed out that supernodes could be defined without using the contiguous constrains [Liu92]. On

check grammar

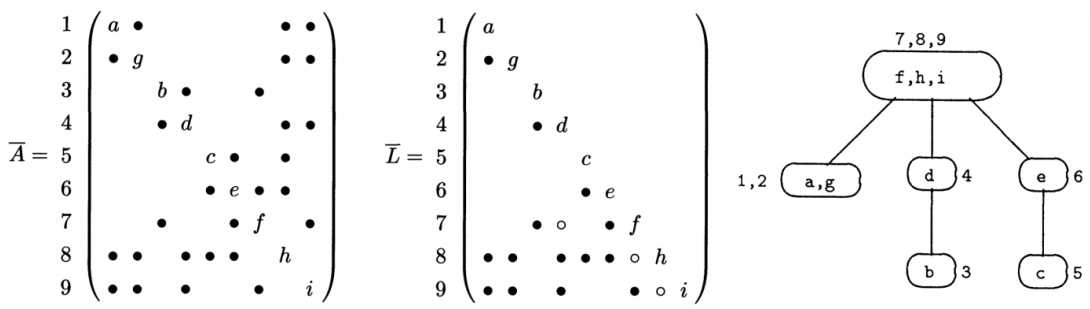



Figure 5.8.: An example of a supernodal elimination tree [Liu92]

another hand, Wu defines supernodes corresponded to separators from the nested dissection step [Wu12] which was used for fill-in reduction.

$$\mathcal{F}_j = \begin{bmatrix} a_{j,j} & a_{j,j+1} & \dots & a_{j,j+t} & a_{j,i_1} & \dots & a_{j,i_r} \\ a_{j+1,j} & a_{j+1,j+1} & \dots & a_{j+1,j+t} & a_{j+1,i_1} & \dots & a_{j+1,i_r} \\ \vdots & \vdots & \dots & \vdots & & & \\ a_{j+t,j} & a_{j+t,j+1} & \dots & a_{j+t,j+t} & a_{j+t,i_1} & \dots & a_{j+t,i_r} \\ a_{i_1,j} & a_{i_1,j+1} & \dots & a_{i_1,j+t} & & & \\ \vdots & \vdots & \dots & \vdots & & 0 & \\ a_{i_r,j} & a_{i_r,j+1} & \dots & a_{i_r,j+t} & & & \end{bmatrix} \; U_{c_1} \; \dots \; U_{c_s} \quad (5.22)$$

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.

### 5.2.2. Parallelization Aspects

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 5.9 shows task parallelism, for the example given in Figure 5.8, where each color represents a set concurrent tasks.

place for figure 5.9

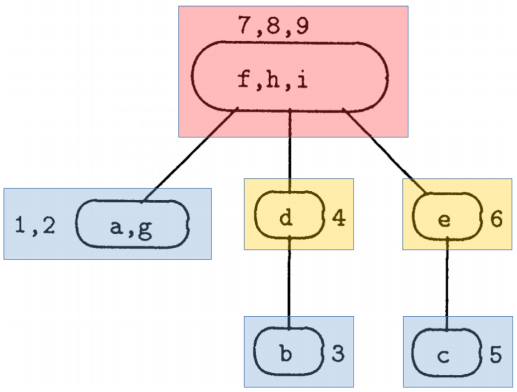



Figure 5.9.: Parallel steps of the multifrontal method based on the example in Figures 5.8

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.

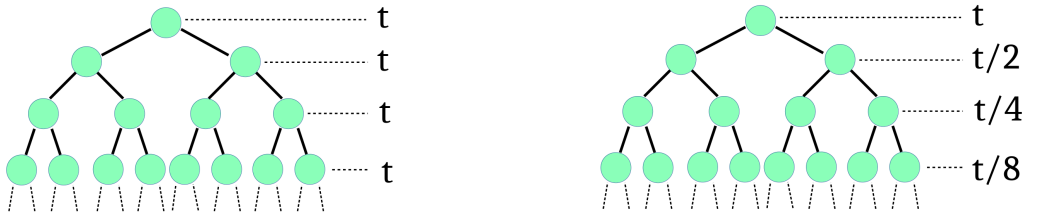
We developed two simple models based on perfectly balanced binary trees to better understand strong scaling of the algorithm. The main concept of the models is so-called cost per level or cost per node. This idea is similar to the recursion trees in [Cor+09] which explains and computes complexity of recurrent algorithms.

Figure 5.10a represents the first model where we keep the same cost per level whereas the second model (Figure 5.10b) 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 5.10

sentence refactor-ing




(a) Model 1: equal cost per level

(b) Model 2: quadratically decreasing cost per level

Figure 5.10.: Simple parallel models of the multifrontal method

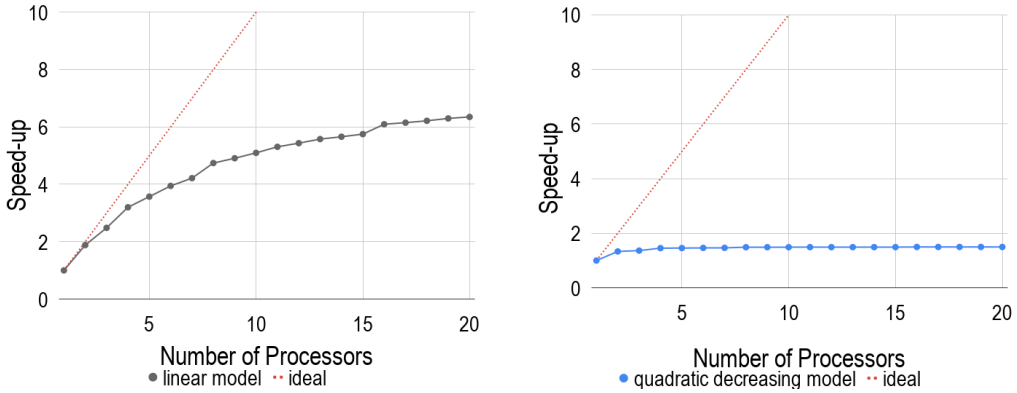
time due to growth of frontal matrices in size, time increase spent on numerical pivoting, increase of MPI communication overheads due to growth of contribution matrices, etc. It should be mentioned that real computer implementations of the multifrontal algorithm (MUMPS, SuperLU, etc.) are quite sophisticated 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 5.9, 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 [Wik18a]. 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 5.10a and 5.10b, respectively. The figures show a rapid drop of performance, especially in case of the quadratic model. Table




(a) Theoretical speed-up of Model 1

(b) Theoretical speed-up of Model 2

Figure 5.11.: Theoretical speed-up

5.1 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 5.11

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

Table 5.1.: Potential speed-up of linear and quadratic models

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 5.2.

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

Table 5.2.: 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 5.10 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. Liu discussed application of the multifrontal method to a $k - by - k$ regular model problem with nine-point difference operator in his paper [Liu92]. 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 5.3 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 5.3.: 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 the 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 5.12 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 [OA07].

place for figure 5.12

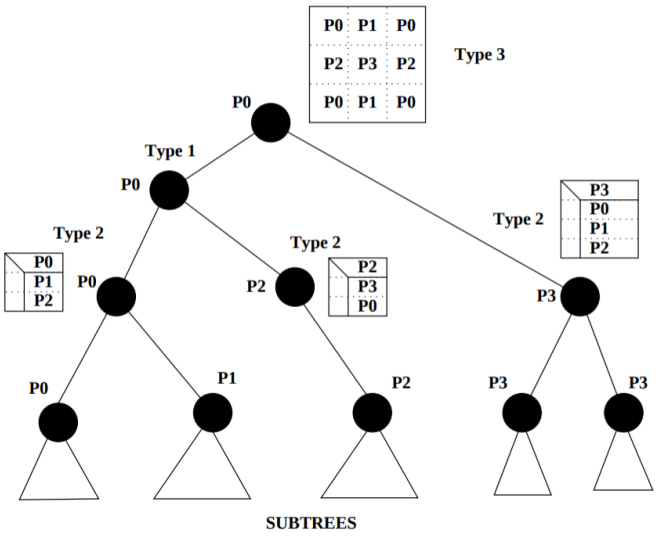



Figure 5.12.: MUMPS parallelism management in case of 4 PEs [OA07]

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 **??**. 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 [17]. 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 that can be deep and unbalanced.

Figure 5.13 shows comparison of strong scaling between model 1 and parallel numerical factorization of the matrix ***memchip*** (Table 4.2) done with using MUMPS library. The sparsity pattern before and after fill-in reduction is shown in figure 5.14.

add some examples to the appendix

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,

place for figure 5.13. Show standard deviation

place for figure 5.14

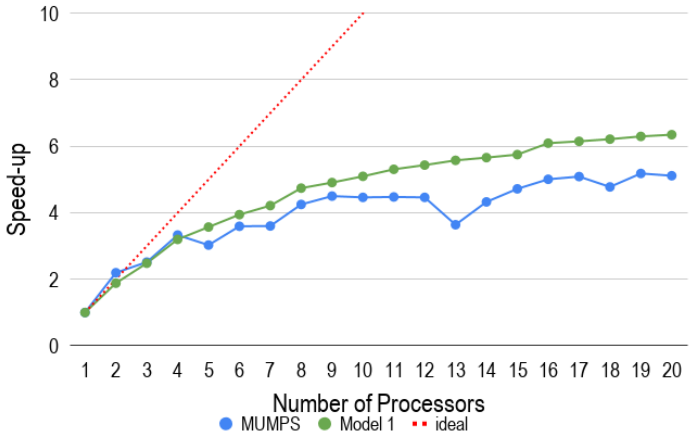



Figure 5.13.: Comparison between model 1 and numerical factorization of the matrix ***memchip*** using MUMPS library

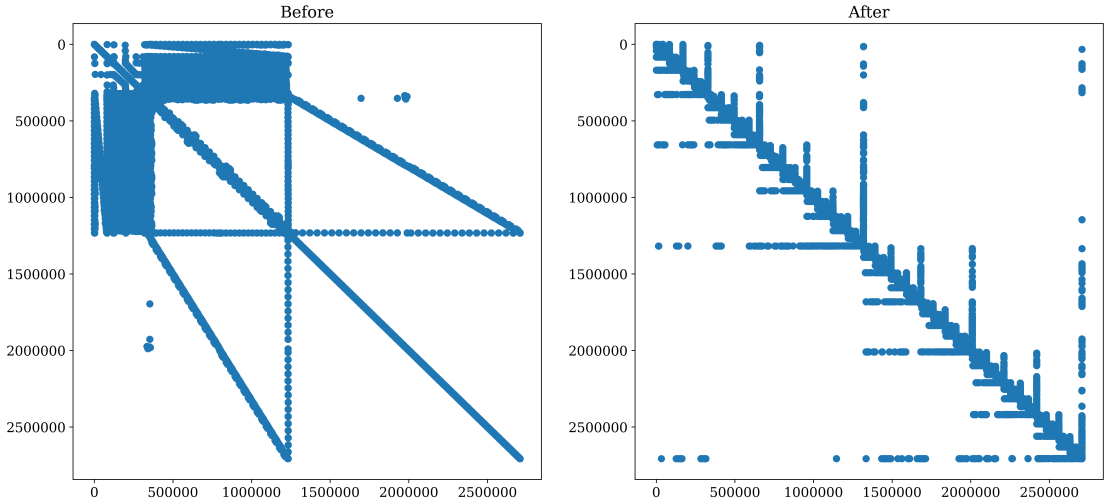



Figure 5.14.: Sparsity structure of the matrix ***memchip*** before and after fill-in reduction

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.

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 5.13. 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 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.

### 5.2.3. Numerical Pivot Handling

## 5.3. Conclusion

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 **??** and **??** 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 **??** 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 [RBH12]. ***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 5.1: A pseudo-code of the Hybrid approach

According to our primary tests, the hybrid approach showed us that it required from 1 to 5 iterations of the GMRES solver on average to converge to a desired residual.

write which solver we used

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 **??**. 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 [Bal+18]. 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.

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

proofreading

# 6. Selection of a Sparse Direct Linear Solver Library

Fair to say, there is no single algorithm or software that is best for all types of linear systems [Li18]. 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 6.1 displays a short summary of almost all available and well-known packages, at the time of writing, in this field based on works [Li18] and [Bal+18].

add footnotes of Open*

| 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 6.1.: List of packages to solve sparse linear systems using direct methods on distributed memory parallel machines [Li18], [Bal+18]

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

We can see, from table 6.1, 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 [11]. 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 7.1). 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 7.1, is claimed to converge to a decent precision within 2 or 3 steps in work [ADD89]. 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 [ADD89] which can be expressed as following:

$$\frac{|b - A\hat{x}|_i}{(|b| + |A||\hat{x}|)_i} \tag{6.1}$$

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 6.1: 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 [17].

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

$$|a_{i,i}| \geq u \times max_{k=i\cdots n}|a_{k,j}| \tag{6.2}$$

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 6.2 cannot be satisfied within a fully-summed block of a frontal matrix (equation 5.22) 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 6.2, 6.3, 6.4 and in appendix B. Numerical values in tables are given in **seconds**.

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, 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 con-

| 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 |

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 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 6.2.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **cube-5** (9352 equations)

ducted 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 6.1.

place for figure 6.1

According to the test results, it is clear that MUMPS significantly outperforms both SuperLU_DIST and PaStiX. A literature review showed that Gupta, Koric, and George, in paper [GKG09], 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 [KBK16] showed an almost opposite outcome. According to Kwack, Bauer, and Koric, 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

| 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 |

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 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 6.3.: 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 |

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 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 6.4.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **k3-18** (1155955 equations)

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

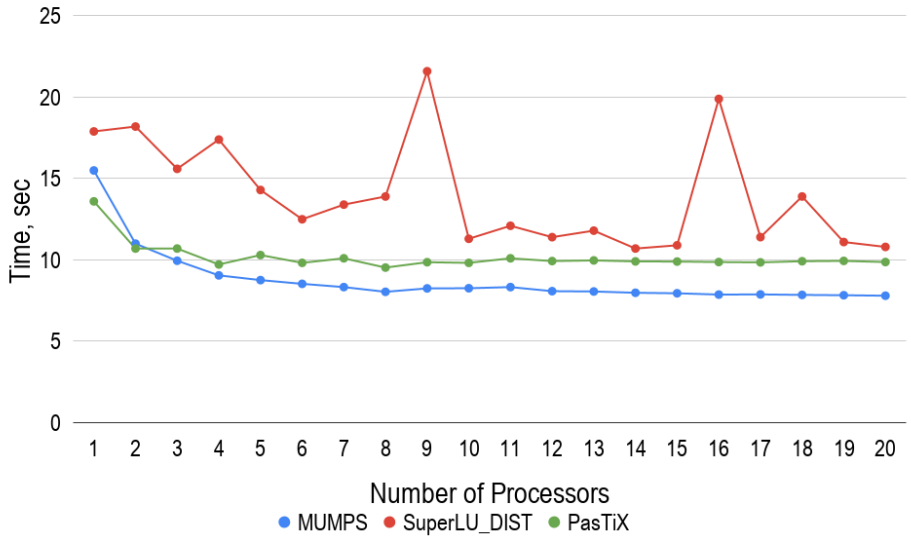



Figure 6.1.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and a 5 point-stencil Poisson matrix (1000000 equations)

# 7. Configuration of MUMPS library

## 7.1. 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 [Ame+98]. 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 [Ame+02]. 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 6 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 **??**. 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) [ADD96], Approximate Minimum Fill (AMF), Approximate Minimum Degree with automatic

quasi-dense row detection (QAMD) [Ame97], Bottom-up and Top-down Sparse Reordering (PORD) [Sch01], Nested Dissection coupled with AMD (Scotch) [Pel08], Multilevel Nested Dissection coupled with Multiple Minimum Degree (METIS) [KK09]. 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 [17].

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 [DK99], [DK01] (for the unsymmetric case) and [DP05] (for the symmetric case). This preprocessing step is supposed to improve numerical accuracy and makes all estimations performed during analysis more reliable [17]. 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 **??**.

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 5.12). 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 [GN89], 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 7.1.

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 6 where different implementations of direct sparse solvers are compared.

place for figure fig:mumps:mapand-scheduling

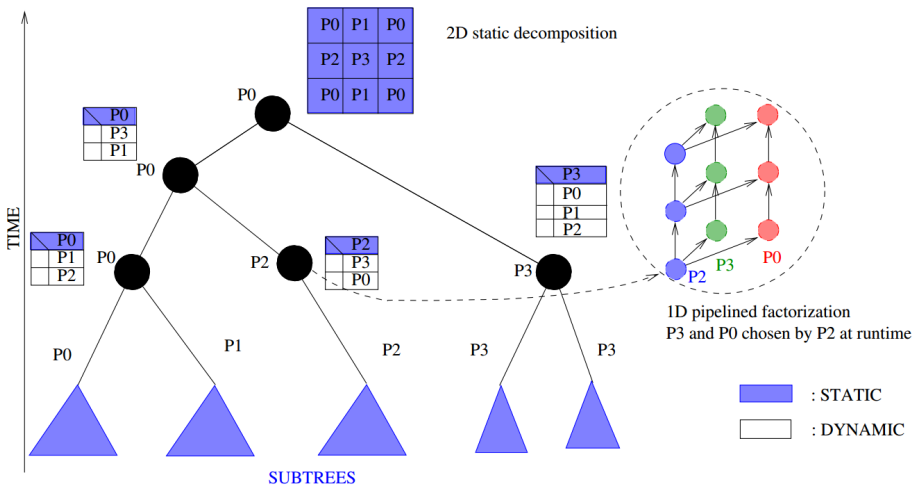



Figure 7.1.: MUMPS: static and dynamic scheduling [LEx12]

## 7.2. 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 7.1. A detailed study and comparison between different methods were done by Guermouche, L'Excellent, and Utard in work [GLU03] for sequential execution of the analysis phase. Guermouche, L'Excellent, and Utard 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 [GLU03].

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 7.1.

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 any explicit process pinning. The results are shown in figure 7.2 as well as in appendix C.

place for figure 7.2

According to the results, parallel performance of MUMPS can vary significantly and very sensitive to a used fill-in reducing reordering algorithm. In average, the difference between application of different algorithms achieves almost **15%**. However, in some particular cases, *cube-5* and *pwr-3d*, the difference varies around **40-55%**.

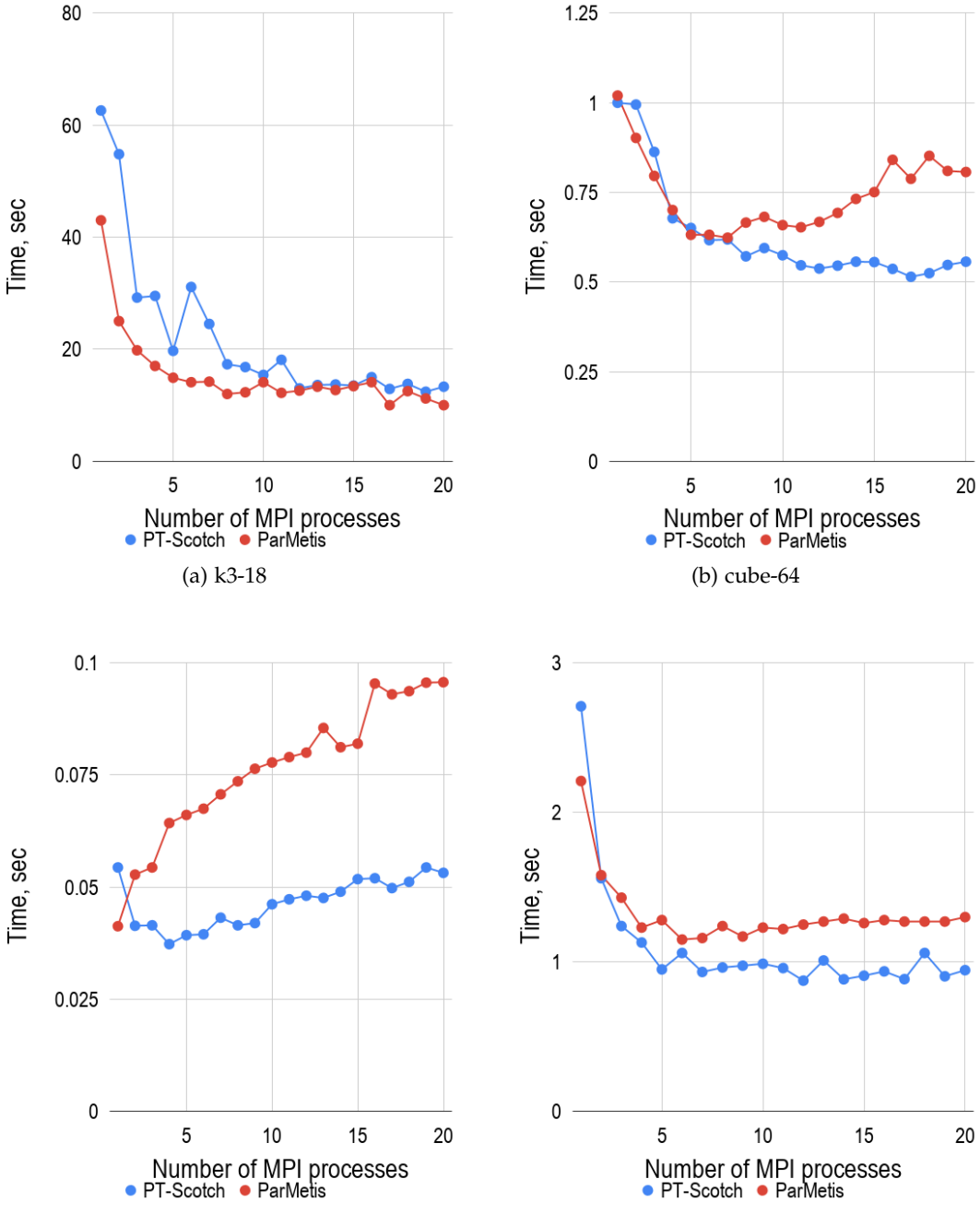
It is important to mention that both packages, PT-Scotch and ParMetis, use heuristic approaches with the main aim to reduce fill-in of the factors and thus do not directly consider quality of the resultant elimination tree. It is relevant to assume that efficiency of a particular heuristic can be very 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*.

During the test, we noticed that application of ParMetis for small systems of equations showed a strong negative effect on parallel performance of MUMPS. The results showed the factorization time of *pwr-3d* and *cube-5* matrices grew with the increase of processing units (figure 7.3).

place for figure 7.3

A simple profiling showed two important things. Firstly, numerical factorization time and time spent on the analysis phase had approximately 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 significantly grew. By and large, we can observe the slow-down of MUMPS mainly comes from only the analysis phase.

A careful investigation revealed the analysis phase contained several peaks at points where the processor count was equal to a power of two. We assumed that cause could be due to either fill reducing or process mapping pre-processing steps. However, a detailed profiling and tracing of the analysis phase, which are out of the scope of this



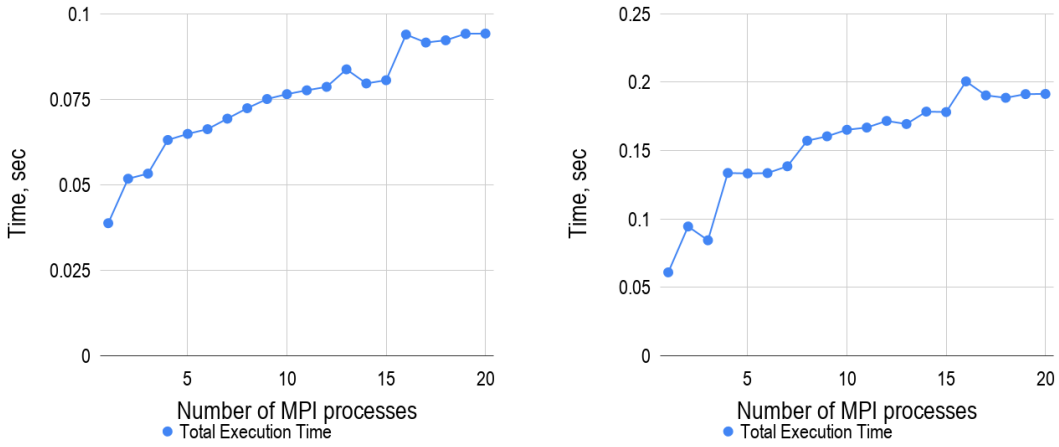

(a) k3-18

(b) cube-64

(c) pwr-3d

(d) k3-2

Figure 7.2.: Comparison of different fill-reducing algorithms




(a) pwr-3d (b) cube-5

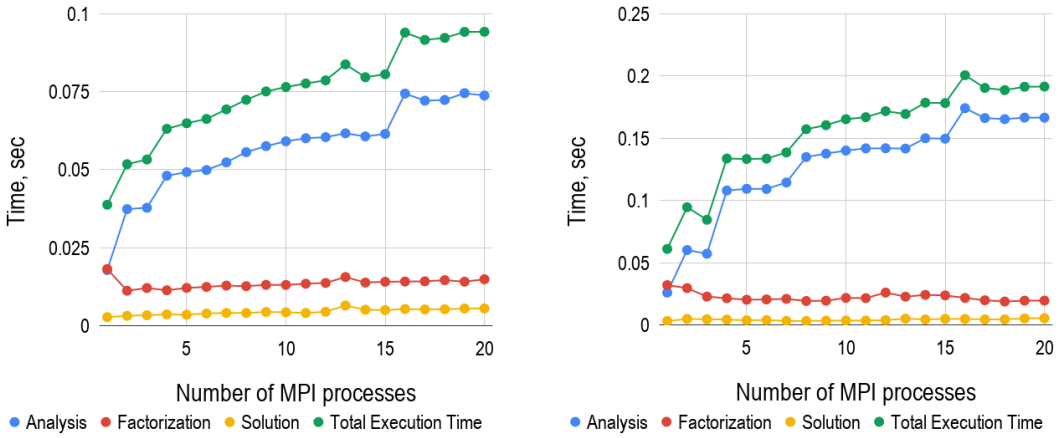
Figure 7.3.: MUMPS-ParMetis parallel performance in case of relatively small matrices

study, are required in order to give the exact answer. The results of profiling are shown in figure 7.4.

place for figure 7.4

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 the 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 as it is in case of factorization of *CurlCurl_3* and *cant* matrices, for example (see appendix C). 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 7.1 and 7.2.

From now onwards, assignments mentioned in tables 7.1, 7.2 will be used without explicitly referring to it.




(a) pwr-3d

(b) cube-5

Figure 7.4.: Profiling of MUMPS library with using ParMetis as a fill-in reducing algorithm in case of factorization of relatively small matrices

| 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 7.1.: GRS matrix set: assignment of matrices to a specific fill-in reducing algorithm based on parallel performance of 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 7.2.: SuiteSparse matrix set: assignment of matrices to a specific fill-in reducing algorithm based on parallel performance of flat-MPI tests

## 7.3. 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 NUMA domains within a compute node. Given the fact that each NUMA domain has its own system bus, this strategy allows to reduce conjunction of memory traffic by balancing data requests equally among the memory channels.

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 4.

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, MUMPS was used with default settings and a specific fill-in reducing algorithm for each test case, mentioned in section 7.2. The tests were performed on both HW1 and HW2 machines using only flat-MPI mode. This comparison allows to investigate influence of number independent system buses of a compute node on MUMPS overall performance since HW1 and HW2 machines have different number of NUMA domains, 2 and 4, respectively. Results are shown in figures 7.5, 7.6, 7.7 and in appendix D. The graphs depict the total time of MUMPS, i.e. time spent on analysis,

factorization and solution phases.

place for figure 7.5

place for figure 7.6

place for figure 7.7

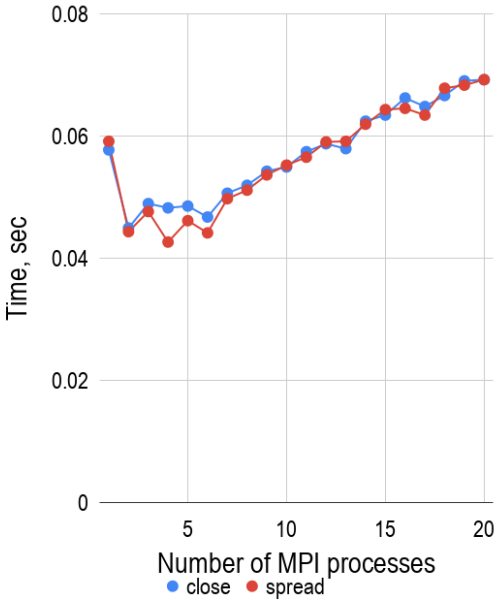
The tests revealed that, in general, *spread*-pinning performed better on both machines. In average, the strategy allows to reduce run-time by approximately **5.5%** and **13.8%** on HW1 and HW2 machines, respectively. The main performance gain can be observed in the middle range of process count i.e. the range from 2 to 12, where the view on the process distribution, between *close* and *spread* strategies, varies considerably. On another hand, the performance gain becomes less prominent while moving towards the tail of process count since difference of process distribution becomes negligible. As expected for HW1, the points where process count is equal to 1 and 20 show the same performance because they basically represent the same process distribution.

It is also important to investigate the gain of performance around the saturation point. It is worth pointing out that from time to time it becomes very difficult to decide where the saturation point locates. For that reason, a careful analysis was performed for each graph based on values of speed-up, efficiency and our subjective opinion. The results are summarized in tables 7.3 and 7.4.

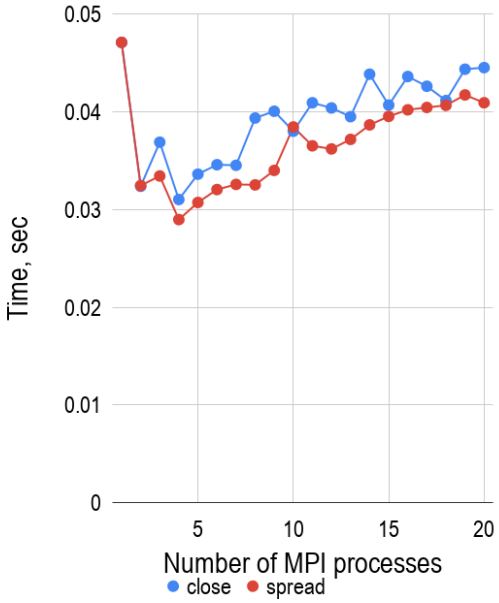
| | HW1 | | | | HW2 | | | |
|---|---|---|---|---|---|---|---|---|
| Matrix Name | MPI | Gain w.r.t "close", % | Speed up | Effi-ciency | MPI | Gain w.r.t "close", % | Speed up | Effi-ciency |
| pwr-3d | 4 | 11.594 | 1.386 | 0.347 | 4 | 6.616 | 1.626 | 0.406 |
| cube-5 | 4 | 8.261 | 1.139 | 0.285 | 4 | 10.640 | 1.156 | 0.289 |
| cube-64 | 8 | 5.645 | 1.812 | 0.226 | 8 | 7.521 | 1.729 | 0.216 |
| cube-645 | 6 | 9.985 | 2.152 | 0.359 | 8 | 9.078 | 2.521 | 0.315 |
| k3-2 | 7 | 7.788 | 2.899 | 0.414 | 8 | 9.947 | 3.298 | 0.412 |
| k3-18 | 8 | 6.716 | 3.472 | 0.434 | 8 | 9.567 | 3.896 | 0.487 |

Table 7.3.: Analysis and comparison of MUMPS performance at the saturation point between HW1 and HW2 for GRS matrix set

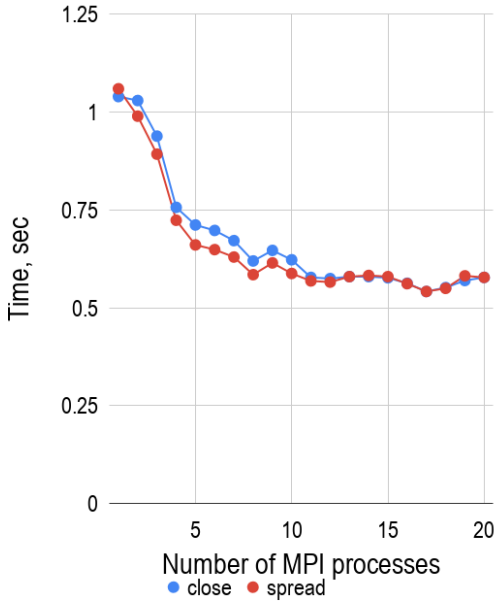
A study of tables 7.3 and 7.3 reveals HW2 machine performs slightly better in contrast to HW1 with respect to performance gain around the saturation points. This result is considerably different from the overall performance gain that was mentioned above where the difference was as twice as much. However, the results also show that increase



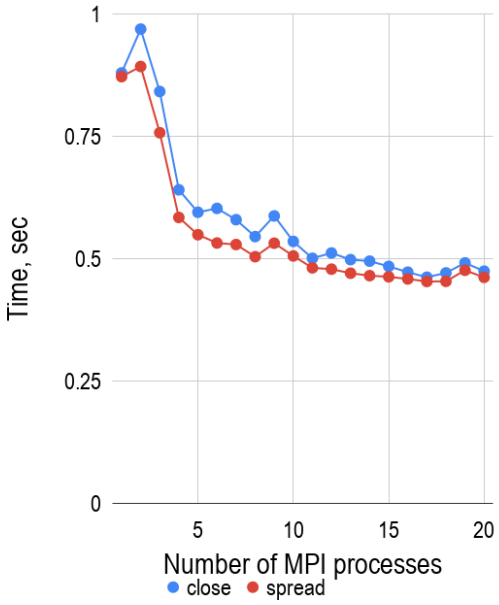

(a) HW1 - pwr-3d




(b) HW2 - pwr-3d

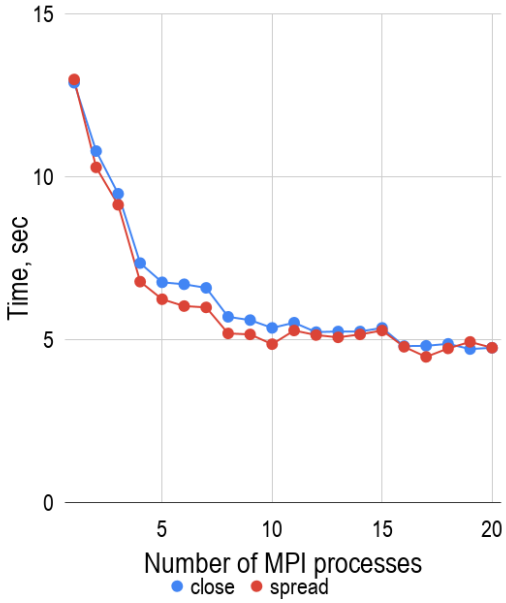



(c) HW1 - cube-64

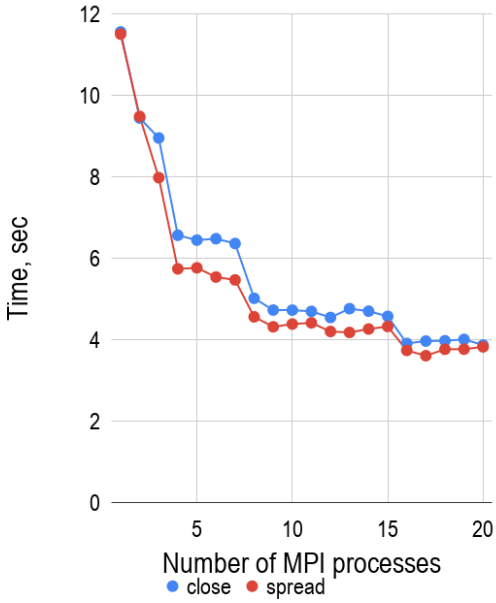



(d) HW2 - cube-64

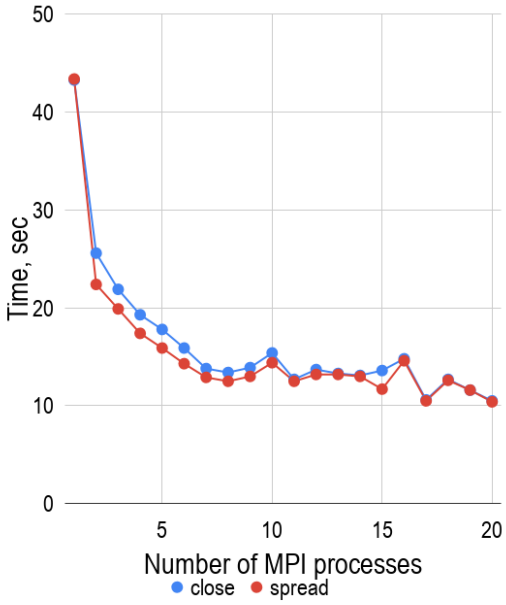
Figure 7.5.: Comparison of *close* and *spread* pinning strategies



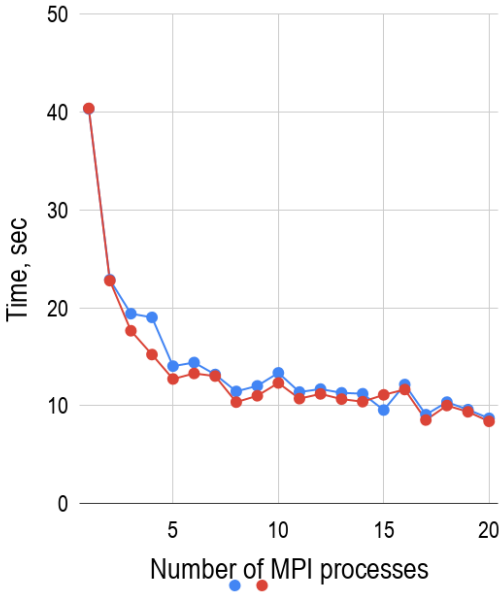

(a) HW1 - cube-645




(b) HW2 - cube-645

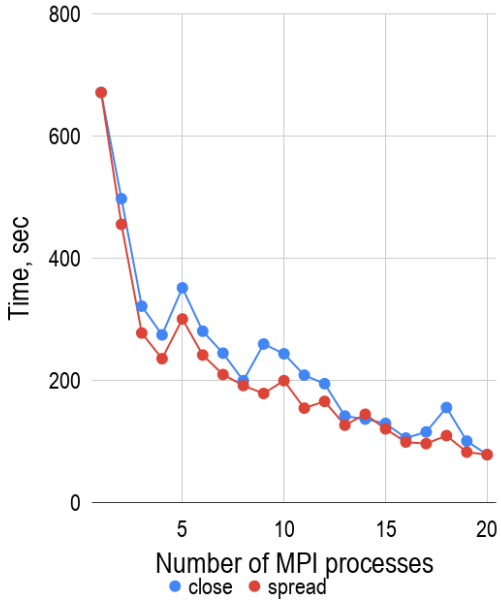



(c) HW1 - k3-18

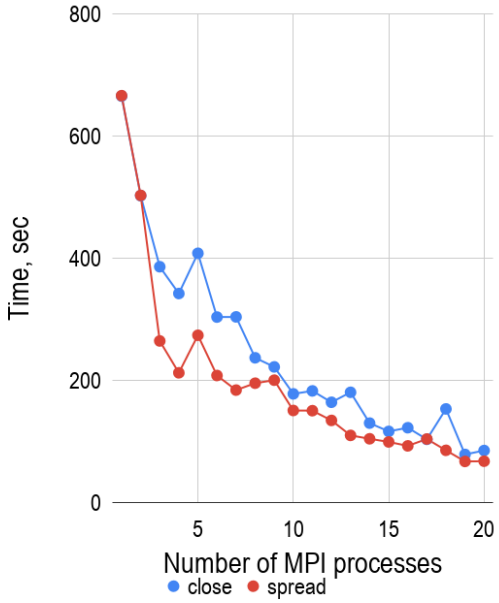



(d) HW2 - k3-18

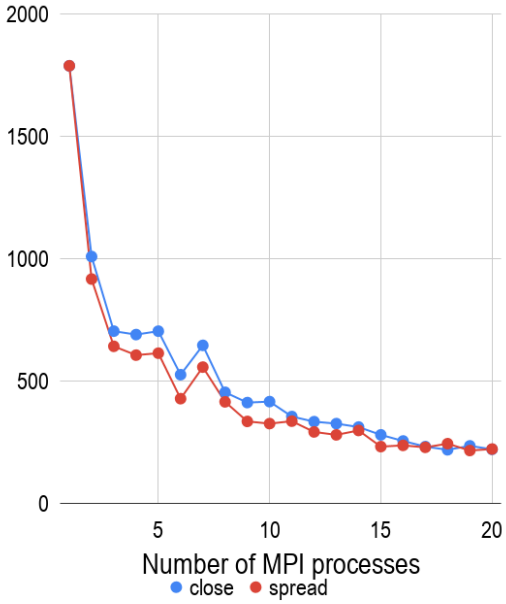
Figure 7.6.: Comparison of *close* and *spread* pinning strategies



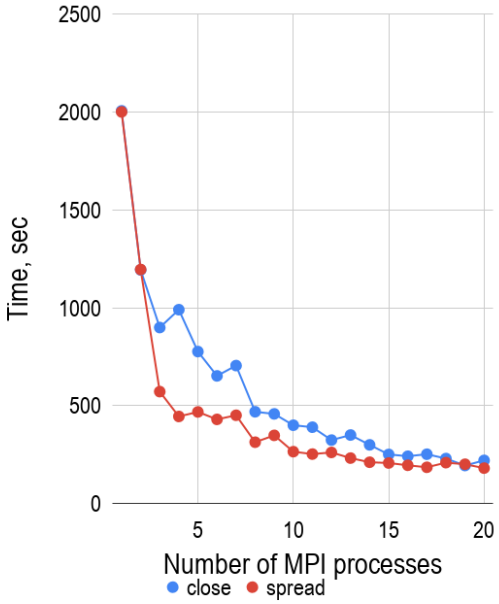

(a) HW1 - PFlow_742




(b) HW2 - PFlow_742




(c) HW1 - CurlCurl_3




(d) HW2 - CurlCurl_3

Figure 7.7.: Comparison of *close* and *spread* pinning strategies

| | HW1 | | | | HW2 | | | |
|---|---|---|---|---|---|---|---|---|
| Matrix Name | MPI | Gain w.r.t "close", % | Speed up | Effi-ciency | MPI | Gain w.r.t "close", % | Speed up | Effi-ciency |
| cant | 8 | 7.914 | 3.297 | 0.412 | 8 | 12.437 | 3.407 | 0.426 |
| consph | 15 | 0.110 | 6.147 | 0.410 | 15 | 2.409 | 6.667 | 0.444 |
| CurlCurl_3 | 19 | 8.051 | 8.249 | 0.434 | 20 | 17.908 | 11.039 | 0.552 |
| Geo_1438 | 13 | 21.609 | 4.548 | 0.350 | ROM | ROM | ROM | ROM |
| memchip | 9 | 11.290 | 4.299 | 0.477 | 9 | 11.102 | 4.213 | 0.468 |
| PFlow_742 | 19 | 17.921 | 8.106 | 0.427 | 20 | 20.469 | 9.798 | 0.490 |
| pkustk10 | 17 | -0.664 | 3.872 | 0.228 | 17 | -1.108 | 4.036 | 0.237 |
| torso3 | 18 | 5.607 | 8.149 | 0.453 | 19 | 6.028 | 9.493 | 0.499 |
| x104 | 6 | 9.537 | 1.789 | 0.298 | 6 | 7.829 | 1.763 | 0.294 |

Table 7.4.: Analysis and comparison of MUMPS performance at the saturation point between HW1 and HW2 for SuiteSparse matrix set.
*ROM - run out of memory

of NUMA domains always help to improve the values of efficiency and speed-up and thus strong scaling behavior of MUMPS.

In this section, we showed influence of process distribution as well as the NUMA domain count on overall MUMPS parallel performance. We show the *spread* process distribution always has some beneficial effect. The increase of NUMA domains also brings some extra performance but not too much as it was expected at the beginning.

This result of this study can be relevant for energy-efficient parallel computing where a strong requirements to program efficiency are applied. This fact usually forces the user to decrease process count and go a slightly away from saturation point in order to keep the values of efficiency around 0.7-0.8. In this case performance of MUMPS can be improved in 15-20% in contrast to the straight forward process pinning.

Taken into account results of the tests, *spread*-pinning has been chosen for the rest of the study. This process distribution can be easily achieved by means of some advanced OpenMPI command line options, for example *–rank-by* and *–bind-to*, as following:.

```
1 mpiexec --rank-by numa --bind-to core -n $num_proc $executable_name
    $parameters
```

Listing 7.1: An example of *spread*-pinning with using advanced OpenMPI command line options

## 7.4. 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 7.9 demonstrates factorization of a type 2 node.

place for figure 7.8

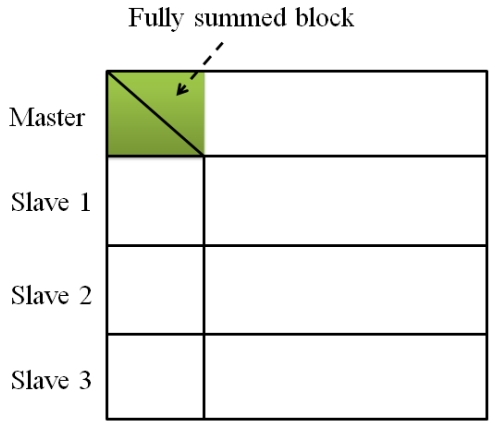



Figure 7.8.: MUMPS: static and dynamic scheduling

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 [06].

The goal of BLAS library is provision of 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.

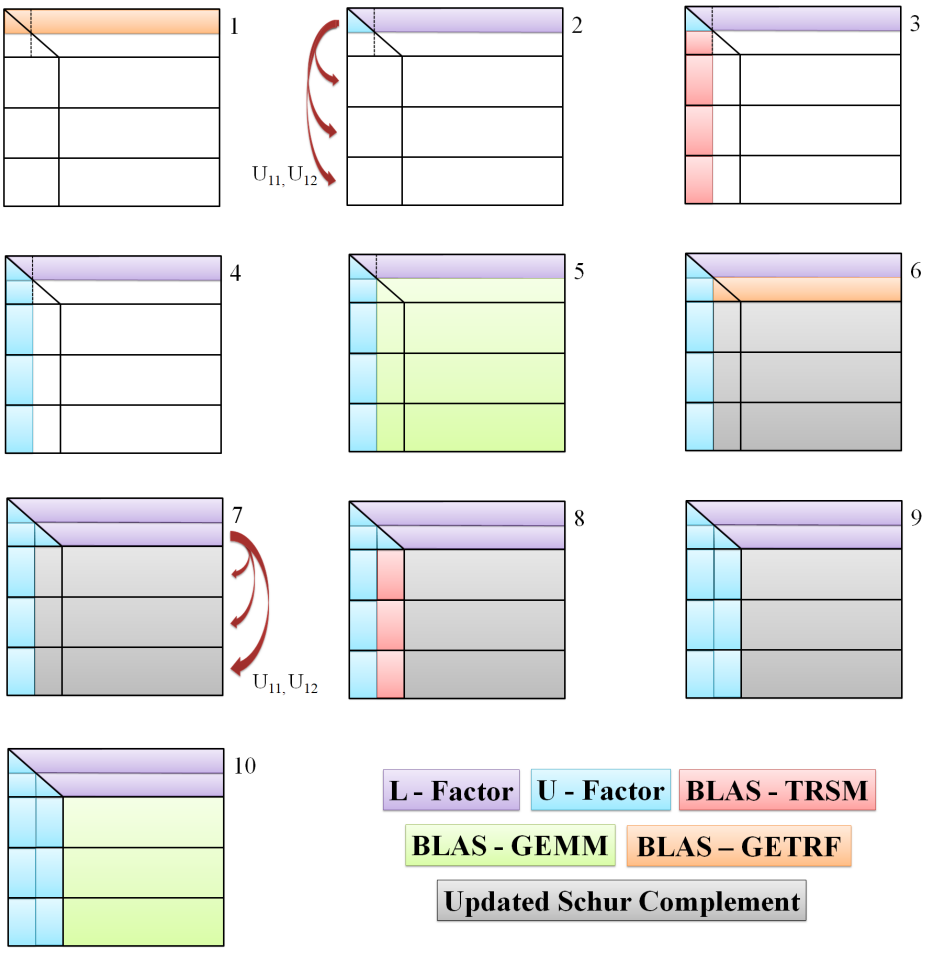



Figure 7.9.: MUMPS: An example of a type 2 node factorization

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 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.

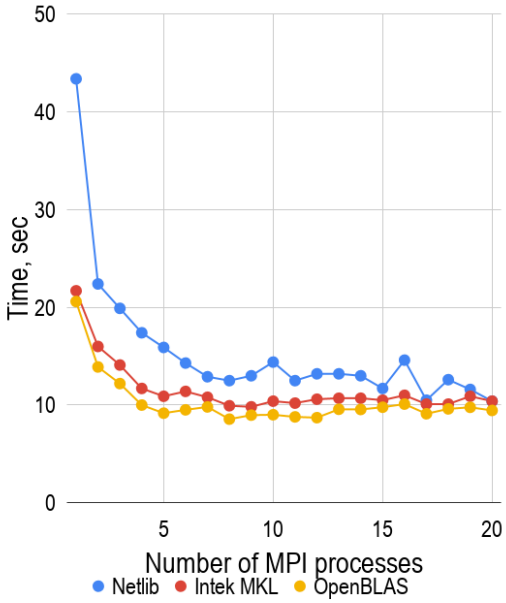
Table 7.5 shows commercial and open-source tunned BLAS implementations available on the market today.

Among all libraries listed in table 7.5 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 routines 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 obtained on HW1 machine only are represented in figures 7.10, 7.11 and appendix E.

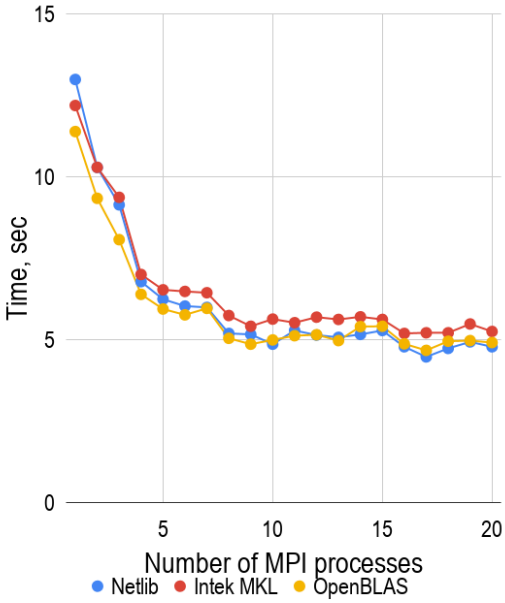
place for figure 7.10

place for figure 7.11

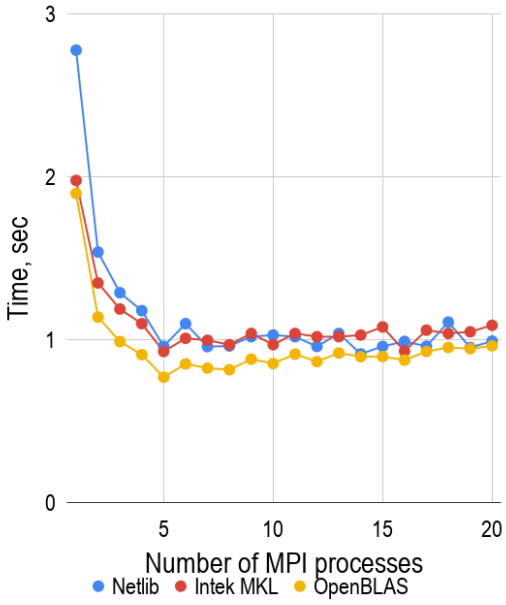
The tests show that OpenBLAS outperforms both Netlib and Intel MKL libraries in case of GRS matrix set. In average, OpenBLAS is about **13%** faster than the default



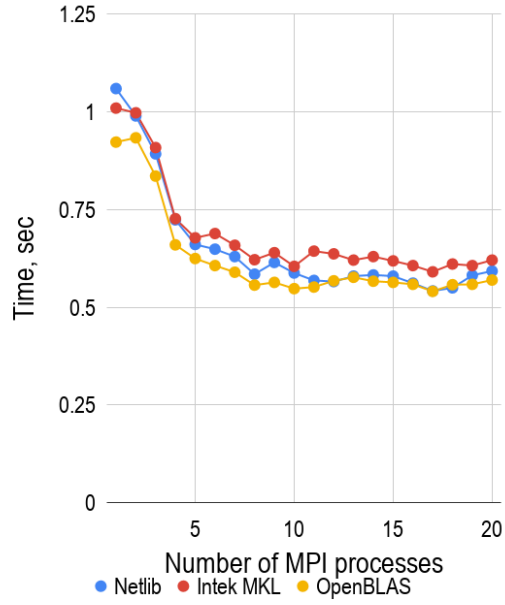

(a) k3-18




(b) cube-645

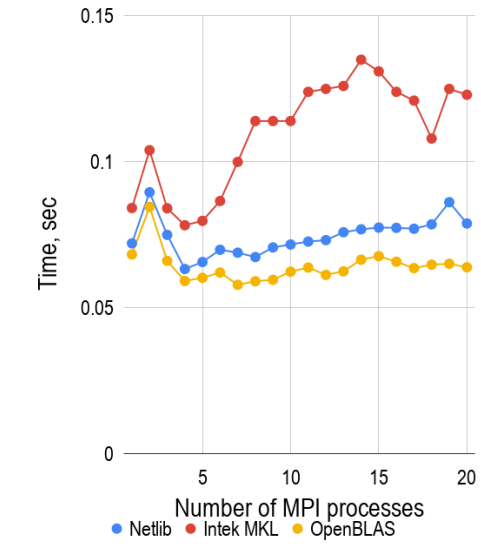



(c) k3-2

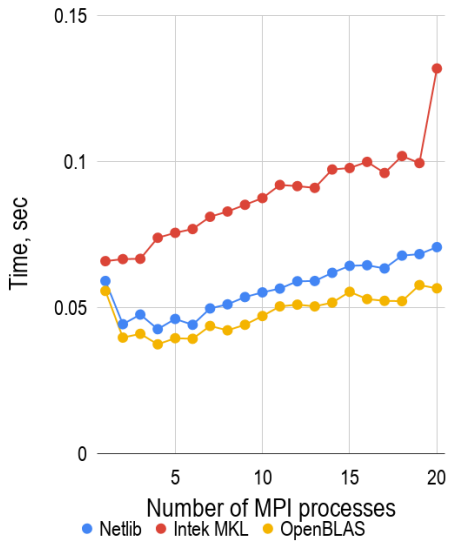



(d) cube-64

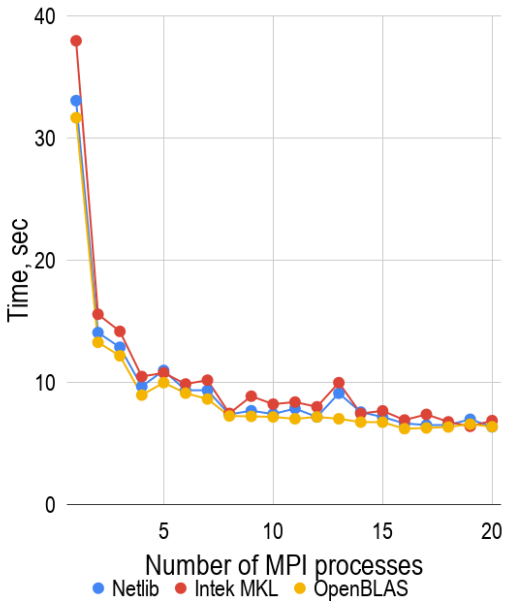
Figure 7.10.: MUMPS: comparison of different BLAS libraries with using GRS matrix set on HW1 machine



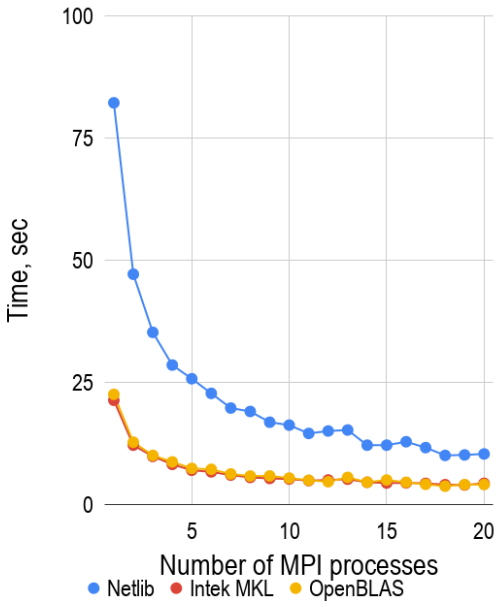

(a) cube-5




(b) pwr-3d




(c) memchip




(d) torso3

Figure 7.11.: MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine

| 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 7.5.: Commercial and open source BLAS implementations [Wik18b]

Netlib implementation and approximately **21%** faster than Intel MKL library. It is interesting to notice Intel MKL turned out to be slower than the default Netlib BLAS implementation for small and medium size GRS matrices in almost **52%** and **2%**, respectively.At the same time, both tuned libraries, OpenBLAS and Intel MKL, showed significant performance gain relatively to the standard Netlib BLAS implementation in case of SuiteSparse matrix set. The libraries reduced the execution time in almost **50%** in average. In opposite to GRS matrix set, it turned out that Intel MKL was often faster than OpenBLAS for almost all test cases from SuiteSparse matrix set. However, the difference between them is negligibly small. The result of comparison are summarized

in tables 7.6 and 7.7.

| Matrix Name | Performance gain of OpenBLAS relatively to Netlib % | Performance gain of IntelMKL relatively to Netlib % | Performance gain of OpenBLAS relatively to Intel MKL % |
|---|---|---|---|
| pwr-3d | 14.607 | -56.249 | 44.695 |
| cube-5 | 13.569 | -47.797 | 39.931 |
| cube-64 | 4.385 | -5.483 | 9.323 |
| cube-645 | 1.897 | -7.474 | 8.702 |
| k3-2 | 13.906 | 0.833 | 13.057 |
| k3-18 | 29.914 | 21.03 | 11.29 |

Table 7.6.: Comparison of different MUMPS-BLAS configurations applied to GRS matrix set

| Matrix Name | Performance gain of OpenBLAS relatively to Netlib % | Performance gain of IntelMKL relatively to Netlib % | Performance gain of OpenBLAS relatively to Intel MKL % |
|---|---|---|---|
| cant | 26.981 | 25.964 | 1.233 |
| consph | 67.617 | 68.252 | -2.327 |
| CurlCurl_3 | 78.804 | 79.37 | -3.371 |
| Geo_1438 | 83.106 | 83.565 | -2.857 |
| memchip | 6.066 | -6.909 | 11.883 |
| PFlow_742 | 75.574 | 74.943 | 1.416 |
| pkustk10 | 35.089 | 34.536 | 0.502 |
| torso3 | 66.185 | 66.988 | -2.837 |
| x104 | 41.82 | 41.936 | -0.445 |

Table 7.7.: Comparison of different MUMPS-BLAS configurations applied to SuiteSparse matrix set

In this section, we have shown how and where MUMPS utilizes common third-party libraries. The design of MUMPS can be considered as a good example of a scientific software architecture which reuses well-known, common blocks which appear quite

often in scientific programs i.e. dense linear algebra kernels. Such design allows to easily configure and adopt software for a specific hardware which we demonstrated in this section.

We have also shown that MUMPS-OpenBLAS configuration is the best choice for GRS matrix set. It has been interesting to observe that the tuned Intel implementation of BLAS compiled with the latest Intel compiler demonstrated slow-down in contrast to the standard not optimized implementation. However, we can assume that amount of GRS test cases is not enough to make such a conclusion and thus the matrix set has to be extended considerably.

## 7.5. MPI-OpenMP Tuning of MUMPS Library

As it was mentioned in section 7.1, 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, Chowdhury and L'Excellent published their first experiments and some issues, in [CL10], 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 L'Excellent and Sid-Lakhdar mentioned, in [LS13], 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 Chowdhury and L'Excellent, 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 [LS13] and [CL10] 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. L'Excellent and Sid-Lakhdar 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, L'Excellent and Sid-Lakhdar did not provide any quantitative measure for that in their work [LS13].

It is also interesting to notice that parallelization of region 1 with using a multi-threaded BLAS library brings most of parallel performance for mixed or pure OpenMP strategies, according to the results from[LS13]. 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, [CL10] and [LS13], 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, [CL10] and [LS13], 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* [17].

As an initial test, we compared 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.

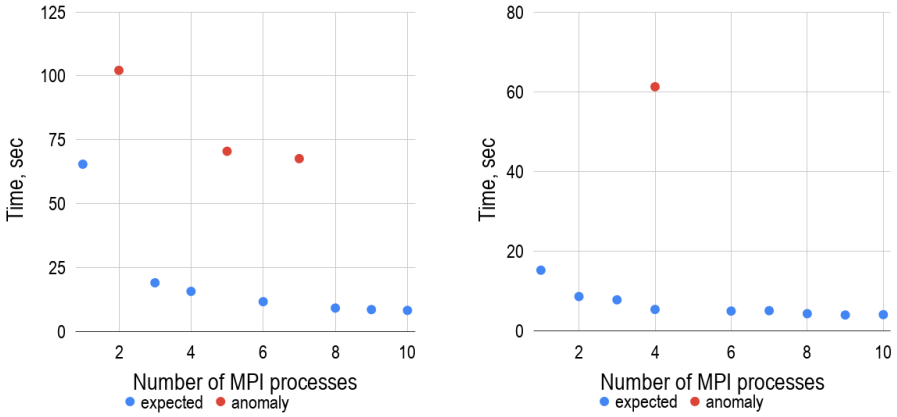
Multiple conflicts between application and system threads were observed using *htop* as an interactive process viewer. Figure 7.13 shows a snapshot taken during factorization of matrix *k3-18* running with 1 MPI process and 20 threads.

place for figure 7.12

It is difficult to say what exactly caused such behavior. However, Chowdhury and L'Excellent 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 [CL10]. For that reason, we decided to stick only to Intel MKL library for the rest of the study of this section because there were no conflicts detected during initial runs.

place for figure 7.13

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




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

Figure 7.12.: Anomalies of MUMPS-OpenBLAS configuration running with 2 OpenMP threads per MPI process

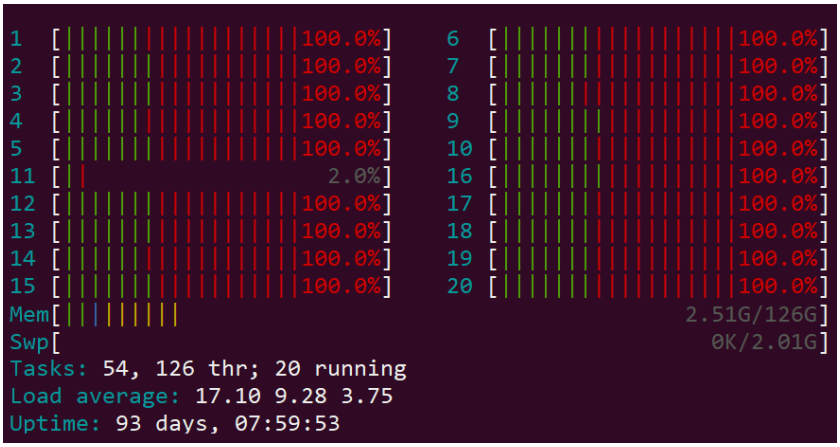



Figure 7.13.: A MUMPS-OpenBLAS thread conflict in case of *k3-18* matrix factorization (green - application threads, red - system threads)

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). The tests were conducted on both HW1 and HW2 machines in order to check whether the results are consistent across different machines with different operating and environment settings. 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 represented in tables 7.8 7.9, 7.10 and 7.11. 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 7.8.: Compassion of different hybrid MPI/OpenMP modes used for GRS matrix set on 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 7.9.: Compassion of different hybrid MPI/OpenMP modes used for GRS matrix set on HW2

According to the results, we have noticed the optimal hybrid MPI/OpenMP mode

| 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 7.10.: Compassion of different hybrid MPI/OpenMP modes used for SuiteSparse matrix set on 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 | ROM | ROM | ROM | ROM | ROM | ROM |

Table 7.11.: Compassion of different hybrid MPI/OpenMP modes used for SuiteSparse matrix set on HW2
*ROM - run out of memory

locates near the saturation point of the corresponding flat-MPI test. Therefore, search of the optimal mode can take considerable amount of time. It is needless to say that the mode varies from matrix to matrix and there is no way to predict the mode in advance. Moreover, the results show that performance gain is around **2.1%** in case of GRS matrix set on HW1 hardware, excluding small test cases such as *cube-5* and *pwr-3d* where runs

with 20 MPI processes were slower in contrast to sequential execution. Much optimistic results were obtained for experiments conducted on HW2 machine were performance gain reached almost **31%** for the same test cases.

SuiteSparse matrix set demonstrates much better performance gain out of hybrid parallel computing on both hardware. In average, run-time is improved in more than **15%** for HW1 and approximately **41%** in case of HW2, excluding *Geo_1438* from statistics. The best result was obtained for *Geo_1438* matrix where execution time dropped in about **3 times** for all hybrid modes relatively to flat-MPI. We suppose it could happen because *Geo_1438* matrix had a very huge ratio of large and small fronts. *Geo_1438* is the best example which shows the benefit of hybrid parallel programing usage.

By an large, we have noticed some negligible improvement of execution time in case of GRS matrix set on HW1 machine with using the optimal hybrid MPI/OpenMP configuration. However, in spite of the improvement, the parallel efficiency drops significantly. As the result, we have come to the conclusion that flat-MPI mode is the best for GRS matrix set on HW1 machine for three reasons. Firstly, this mode is deterministic and requires less amount of testing to find the optimal process count. Secondly, efficiency of hardware usage is better in case of flat-MPI which can be relevant for energy-efficient computing. Thirdly, improvement obtained with using Intel MKL library running with the optimal hybrid MPI/OpenMP mode can deteriorate performance gain of MUMPS-OpenBLAS flat-MPI configuration.

## 7.6. Conclusion

In this chapter, we have examined different types of sparse linear solvers for integration of ODEs systems arisen from thermo-hydraulic simulations. We have come to the conclusion that, in spite of better scalability and parallel efficiency of iterative methods, direct sparse linear solvers are most suitable for this purpose due to its robustness.

In section 6, we have tested different direct sparse solvers, namely: SuperLU_DIST, PasTiX and MUMPS. MUMPS library showed better parallel efficiency among the others according to the test results and thus was chosen for the rest of the study where we focused on performance tuning of the library.

We have shown in subsequent sections there have been four main sources of MUMPS library tuning, namely:

1. careful choice of a fill reducing reordering method
2. distributed MPI process pinning across a compute node
3. configuration of MUMPS with an optimal, tuned BLAS implementaion
4. execution of MUMPS with an optimal hybrid MPI/OpenMP mode

To perform accurate testing and careful analysis, two different matrix sets have been used, namely: GRS and SuiteSparse. The fist one is problem specific for thermo-hydraulic simulations whereas the second one was built from SuiteSparse Matrix Collection [DH11], [DGL89] by downloading a dozen different matrices with respect to the number of equations and the number of non-zeros. Additionally, some tests were performed on two different compute clusters, called HW1 and HW2 (see section 4), to check whether a problem was hardware specific or not.

1. In section 7.2 it has been shown that parallel performance of MUMPS is quite sensitive to a used fill-in reducing reordering algorithm. A right choice of the algorithm can lead to significant improvements in terms of the overall execution time. The average performance gain was almost **15%** and in some particular cases it was possible to reduce execution time in approximately **40-55%** in case of GRS matrix set. During experiments, we came to the conclusion there was no an indirect metric to predict the best algorithm in advance for a specific system of equations.

2. In section 7.3, influence of different process pinning strategies on MUMPS parallel performance has been demonstrated. The tests have shown that equal distribution of MPI process among all available NUMA domains always bring extra performance. In average, *spread*-pinning allows to reduce execution time of MUMPS by almost **5.5%** and **13.8%** on HW1 and HW2 machines, respectively, in case of GRS matrix set.

3. We have discussed and examined in section 7.4 the way how MUMPS library has been implemented to preform partial factorization of type 2 nodes in section. It has turned out the library intensively calls *GEMM*, *TRSM* and *GETRF* BLAS subroutines. Therefore, replacement of standard Netlib BLAS library by tuned BLAS implementations makes it possible to improve overall performance of the solver in case of a sufficient amount of type 2 nodes in a matrix assembly tree.

The results have shown that OpenBLAS outperforms both Netlib and Intel MKL libraries, which were available for the tests, in case of GRS matrix set. The average performance gain turned out to be about **13%** relatively to the default Netlib implementation and approximately **21%** in contrast to Intel MKL library. We have also noticed

that Intel MKL was even slower than Netlib BLAS for small and medium size GRS matrices in almost **52%** and **2%**, respectively. At the same time, both libraries, OpenBLAS and Intel MKL, showed significant performance gain relatively to the standard Netlib BLAS in case of SuiteSparse matrix set and allowed to reduce the execution time by almost **50%** in average.

4. In section 7.5, we have discussed where and how developers of MUMPS library applied shared-memory parallelism based on review of works [CL10] and [LS13]. . We have also found severe slow-down of some hybrid MPI/OpenMP modes due to system and application thread conflicts while MUMPS-OpenMP configuration was running on HW1 cluster. For that reason, the following study was continued with only using Intel MKL library which, in its turn, turned out to be thread-safe.

problems with open blas

intel mkl

We have shown that in some particular cases, factorization of *Geo_1438* matrix for example, hybrid MPI-OpenMP approach can bring significant performance improvement. However, application of hybrid computing to GRS matrix set gives negligible improvement on HW1 machine i.e. around **2.1%**, and significantly deteriorates parallel efficiency. Much optimistic results were obtained for experiments conducted on HW2 machine were performance gain reached almost **31%** for the same matrix set.

During the study, we have also discovered the optimal hybrid MPI/OpenMP mode locates near the saturation point of the corresponding flat-MPI test. This fact can allow to reduce the amount of testing in general. However, we do not recommend to proceed the study in this direction due to above mention parallel efficiency issues detected on HW1 machine and lack of ability to run OpenBLAS library without thread conflicts.

Figures 7.14 and 7.15 show a comparison of MUMPS parallel performance before and after application of above mentioned tuning to GRS matrix set. In case of experiments labeled as *default*, we used ParMetis as a fill reducing reordering algorithm because it had been used by GRS engineers before the current study.

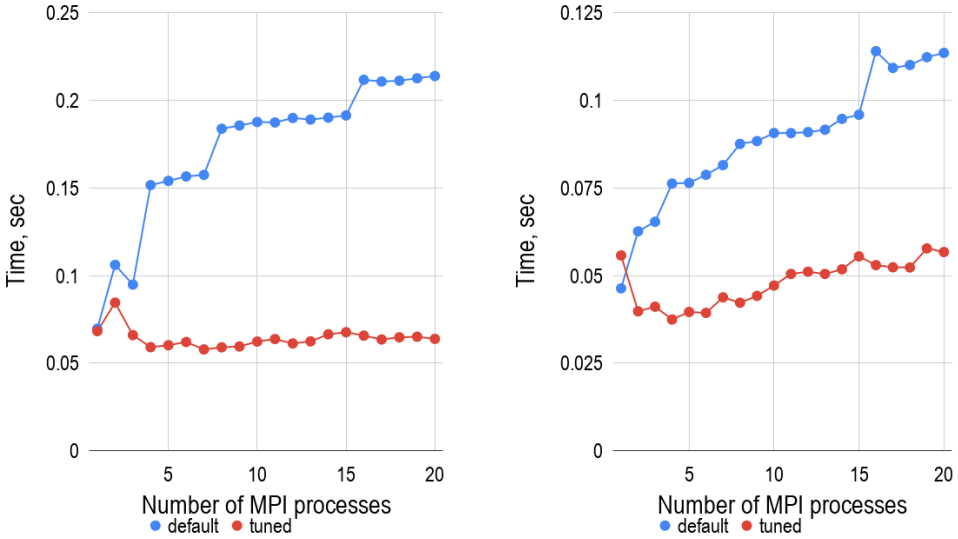
place for figure 7.14

By and large, we have observed significant improvement of applied tuning techniques, mentioned above, for the entire GRS matrix set.

place for figure 7.15

In average, the factorization time has been reduced in **51.4%** for small sized systems. The significant performance gain mainly came from the optimal choice of fill-in reduction algorithm. Moreover, application of PT-Scotch for small sized systems allowed to

include statistics




(a) small system: cube-5

(b) small system: pwr-3d

Figure 7.14.: MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine

drastically change strong scaling behavior and reduce the execution time by approximately **17%** in contrast to sequential execution of the default MUMPS configuration that was a challenge before the study.

The execution time spent on factorization of medium sized systems dropped in **1.4** times. Additionally, we have noticed that scaling of *cube-64* test case has been considerably changed as it happened in case of small sized systems. Application of PT-Scotch, to *cube-64*, allowed to shift the saturation point from 5 to 10 process count without significant drop of parallel efficiency. All in all, tuning of MUMPS made it possible to reduce the execution time around the saturation points in almost **31%** in average for medium sized GRS systems of equations.

The factorization performance gain of large sized systems manly came from configuration of MUMPS with optimized BLAS library, OpenBLAS, and MPI *spread* process distribution because it turned out that ParMetis was the best fill-in reduction algorithm for large systems. A noticeable performance gain difference was observed between *cube-645* and *k3-18* test cases. In average, run-time of MUMPS-OpenBLAS configuration reduced by almost **20%** in contrast to the default installation in case of *k3-18* whereas

factorization time of *cube-645* was improved only by **1.3%**. However, the saturation points of both cases were shifted towards lower values of the process count which allows to considerably improve hardware utilization together with improvement of library performance. For example, a detailed study of *k3-18* performance graph, figure 7.15d, shows that the saturation point has been moved from 17 to 8 process count. At the same time, the execution time drops by almost **19%** and parallel efficiency jumped in about **13%**. The same but less significant results can be observed for *cube-645* test case, figure 7.15c.

## 7.7. Recommendations

According to results and conclusion obtained in this study, we are able to make a general guidance for ATHLET-NuT users and developers.

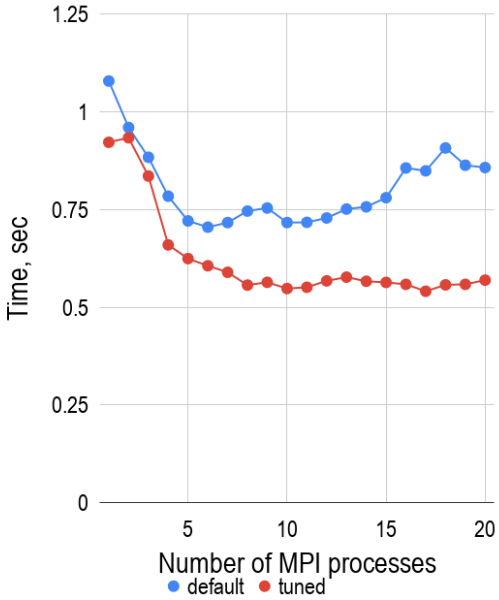
First of all, we insist to configure PETSc-MUMPS module of NuT with OpenBLAS library and use only flat-MPI mode together with *spread* process pinning strategy because, as it was shown in sections 7.3 and 7.4, these settings always have their positive effect on parallel performance regardless of the matrix size and structure.

Secondly, we recommend the user to check table 7.12 before submitting a job on a production compute node in order to define a matrix type of a problem i.e. *small*, *medium* or *large*. Based on the matrix type, the user can select both a fill reducing reordering algorithm and the number of MPI processes which, we believe, can be pretty close to the optimal MUMPS settings.

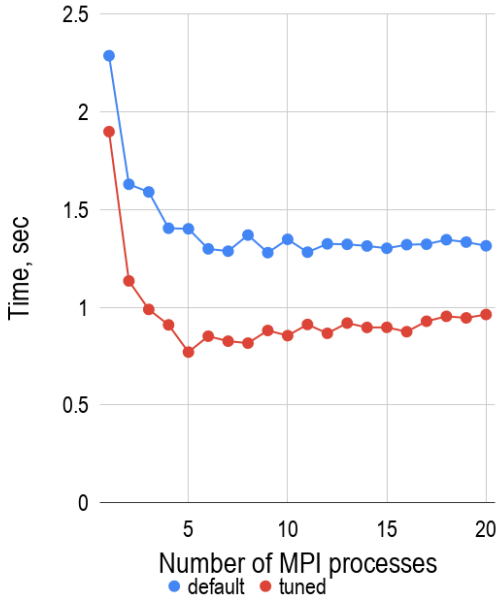
| Matrix type | Number of equations |
|---|---|
| Small | Less than 25000 |
| Medium | form 50000 to 250000 |
| Large | More than 500000 |

Table 7.12.: Matrix type reference table

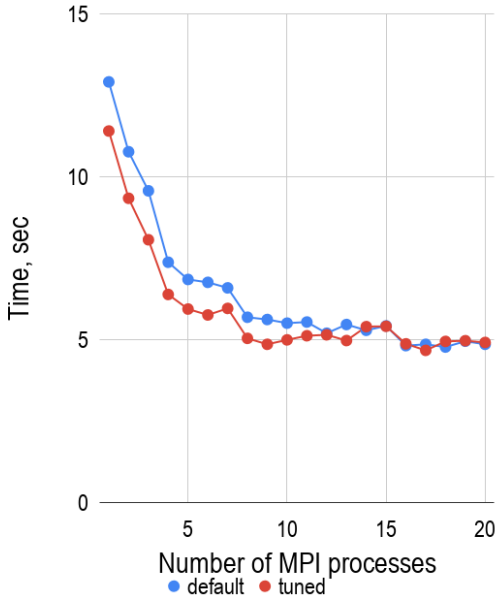
We recommend to use only PT-Scotch and no more than 4 MPI processes for **small** systems. The same reordering algorithm, PT-Scotch, is better to apply to **medium** sized systems of equations as well, however, the optimal number of MPI processes lays in a range from 5 to 10. In case of **large** systems, we insistently recommend to switch to ParMetis package to perform fill-in reduction and start parallel execution from 8 MPI processes.



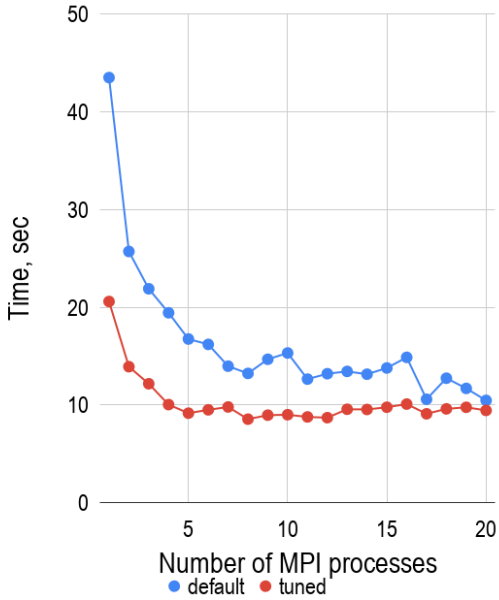

(a) medium system: cube-64




(b) medium system: k3-2




(c) large system: cube-645




(d) large system: k3-18

Figure 7.15.: MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine

# 8. Improvement of ATHLET-NuT Communication

## 8.1. Jacobian Matrix Compression

The main goal of Jacobian matrix compression is minimization of the number of non-linear function evaluations which is quite a computationally intensive operation. Minimization is performed by means of efficient treatment of non-zero entries of a sparse matrix. The problem is also known as matrix partitioning.

In the general case, finite difference method can be used to compute a Jacobian matrix approximation in the following way:

$$\frac{1}{\epsilon}(F(y+\epsilon e_k) - F(y)) \approx J(y)e_k, \quad 1 \leq k \leq N \tag{8.1}$$

where $F : \mathbb{R}^N \rightarrow \mathbb{R}^N$ is a non-linear function, $e_k \in \mathbb{R}^N$ is the k*th* coordinate unit vector, $\epsilon$ is a small step size.

Equation 8.1 does not exploit Jacobian matrix sparsity and, therefore, estimation of a Jacobian matrix requires $N$ function evaluations.

place for figure 8.1

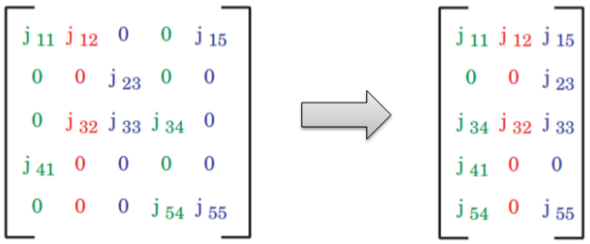



Figure 8.1.: An example of matrix coloring and compression [GMP05]

The compression algorithm is based on the notion of *structurally orthogonal* columns i.e. columns which do not share any non-zero entry in a common row. Figure 8.1 shows an example of a matrix compression where each color denotes independent *structurally orthogonal* columns.

Having obtained a compressed form of Jacobian, another set of vectors $d \in \mathbb{R}^N$, also known as seed vectors, can be used to perform function perturbation instead of unit vectors $e_k$. A seed vector $d$ has 1's in components corresponding to the indices of columns in a structurally orthogonal group of columns, and zeros in all other components [GMP05]. By differencing the function $F$ along the vector $d$, one can simultaneously determine the nonzero elements in all of these columns through one additional function evaluation at $F(y + d)$ [GMP05].

It is obvious the algorithm requires to partition a matrix into the fewest amount of groups, colors, in order to achieve the most of efficiency. There exist various methods and huristics dedicated to that particular problem. Gebremedhin, Manne, and Pothen, in work [GMP05], conducted one of the most recent comprehensive studies in this field and summarized different matrix partitioning algorithms proposed over the last 20 years. Currently, Jacobian matrix compression has been successfully implemented in ATHLET by means of the corresponding built-in PETSc subroutines through NuT interface.

Figure 8.2 shows an illustrative example of an efficient matrix partitioning where an initial 100 by 100 Jacobian matrix is transformed into its 100 by 28 compressed form using 28 distinct colors. It can be clearly observed from the figure the column vector length of the compressed Jacobian form is gradually decreasing. Figure 8.3 provides a detailed and clear view on the problem, using data from figure 8.2 as an example, where a bar represents the corresponding column length.

place for figure 8.2

According to the ATHLET-NuT coupling design, each column is transfered to NuT by means of the synchronous 3-way handshake procedure, described in section 2.3, immediately after its evaluation. Thus, the figure 8.3 determines the communication pattern during the Jacobian matrix transfer.

Code listings 8.1 and 8.2 represent the default compressed Jacobian matrix transfer between ATHLET and NuT. This code was used as a baseline for the remaining part of the study.

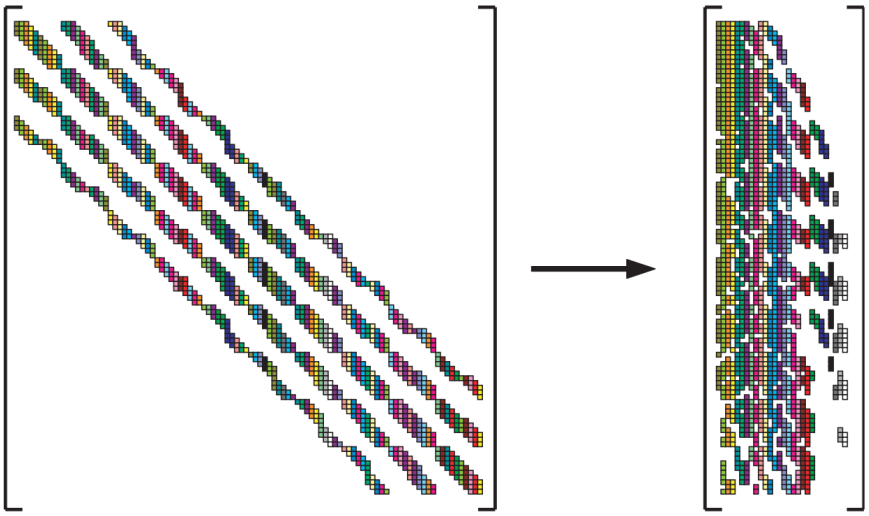


Figure 8.2.: An example of an efficient Jacobian matrix partitioning [GMP05]

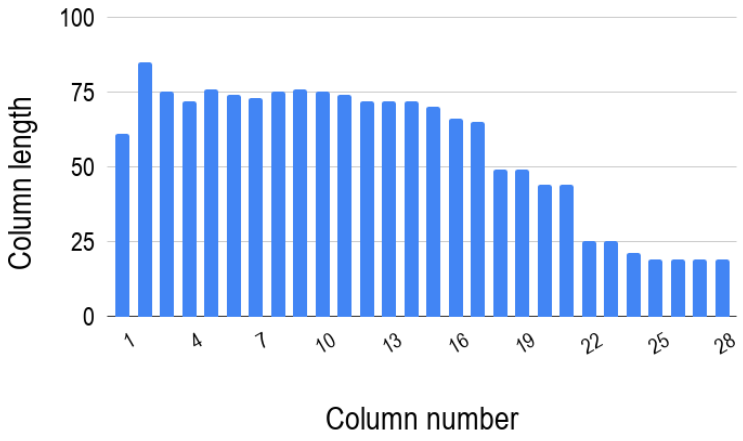



Figure 8.3.: Column length distribution of the example of figure 8.2

All **code listings**, presented in this part of the study, are written in **pseudo-code** and intended for for convenience of reading. The aim is to show and display the main ideas skipping non-relevant parts of the source code. The **pseudo-code** is a mixture of several programming languages, namely: **Python**, **C/C++**, **Fortran**, **MPI**.

```
# GIVEN PARAMETERS:
# acomm - the athlet communicator
# acomm_id - athlet identification number
# y - known vector
# N - problem size
# COO - compressed matrix coordinate format

eps = 1e-4
center = f(y)
column = zeros(N)

# compute Jacobian and send it to NuT column-by-column
for seed_vector in seed_vectors:

  # compute the next column
  vector = evaluate_jacobian(f, seed_vector, center, eps)

  length = perturbed_vector.length
  signal = [encode("add_to_jacobian"), acomm_id]

  # perform 3-way handshake
  MPI_Send(signal, 2, int, acomm.head, acomm)

  # broadcast jacobian column length
  MPI_Bcast(length, 1, int, acomm.head, acomm)

  # broadcast jacobian column
  MPI_Bcast(vector.data, length, COO, acomm.all, acomm)
```

Listing 8.1: Pseudocode of the default ATHLET-NuT coupling: ATHLET part

```
# N - problem size
# J - allocated distributed jacobian matrix
# COO - compressed matrix coordinate format
nut_running = True

while nut_running:
  if rank in heads:

    # receive request
    MPI_Recv(signal, 2, int, NUT_WORLD.any_client, NUT_WORLD)

    comm = my_comm_list[signal[1]]
    if (comm not None):
      # posses resources
      MPI_Bcast(signal, 2, int, comm.all, comm)
    else:
      continue

  else:
    MPI_Recv(signal, 2, int, NUT_WORLD.any_head, NUT_WORLD)

  # decode request
  comm = my_comm_list[signal[1]]
  if (comm not None):
    request = decode(signal[0])

    case(request):
      ...
      if (request == "exit"):
        # beak while loop
        nut_running = False

      if (request == "add_to_jacobian"):
        # receive jacobian column length
        MPI_Recv(length, 1, int, comm.client, comm)

        # receive row jacobian column
        MPI_Recv(elements, length, COO, comm.client, comm)


        for i in range(0, length):
          if (local_min < elements[i].row < local_max):
            J.insert(elements[i])
    ...
```

Listing 8.2: Pseudocode of the default ATHLET-NuT coupling: NuT part

## 8.2. Accumulator Concept

A simple concept, named *accumulator*, has been proposed to improve MPI communication during a Jacobian matrix transfer preserving the current AHLET-NuT architecture and coupling.

The concept represents two arrays of length $2L$ where the first one, called *accumulator*, is used for accumulation of Jacobian matrix elements, stored in compressed sparse matrix format, till the critical array length equaled to $L = F \cdot N$, where $N$ is a size of the underlying Jacobian matrix and $F$ is a so-called capacity factor. Once the current array length of *accumulator* exceeds its critical length, the accumulated data are moved to *send buffer* by means of a simple swap of pointers, *ACC_PTR* and *SEND_BUFF_PTR*, see figure 8.4. Having swapped the pointers and reset control variables, accumulation procedure can be immediately continued together with an immediate instantiation of the corresponding non-blocking broadcast operation with respect to *send buffer* content.

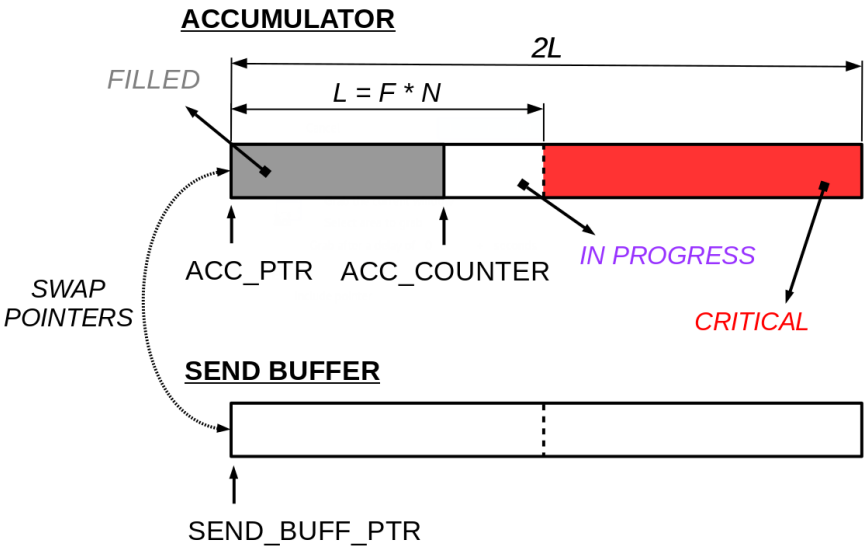



Figure 8.4.: Accumulator concept

The second array part of *accumulator*, also called the critical part, is used for safe placement of data surplus without any extra program checks and manipulations, using an assumption that a column vector cannot exceed the Jacobian matrix size. The assumption is based on long-term and first-hand experience of ATHLET-NuT users. On another hand, the event, described above, triggers a signal for a regular pointer

swap and, therefore, the subsequent non-blocking data transfer.

The factor $F$, depicted in figure 8.4, can be used by the user for two purposes. Mainly, it allows the user to adjust the *send buffer* length $L$ till the point of saturation of physical interconnection bandwidth, see figure 8.5 as an example, and, thus, achieve efficient resource utilization. Additionally, it reduces the amount of handshakes, the amount of resource acquisition requests, between NuT and clients. The default value of the factor is equal to 1, however, we insistently recommend to increase the value via the corresponding environment variable for small-sized problems and operation of NuT in a multi-client mode.

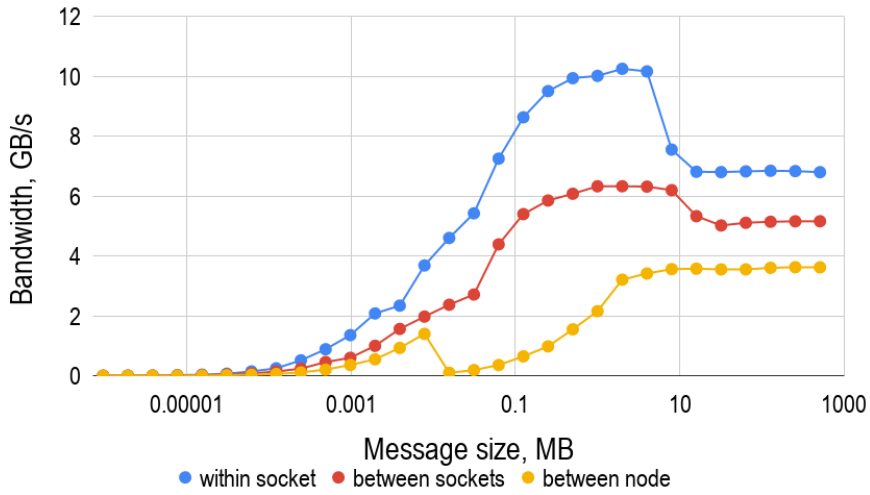
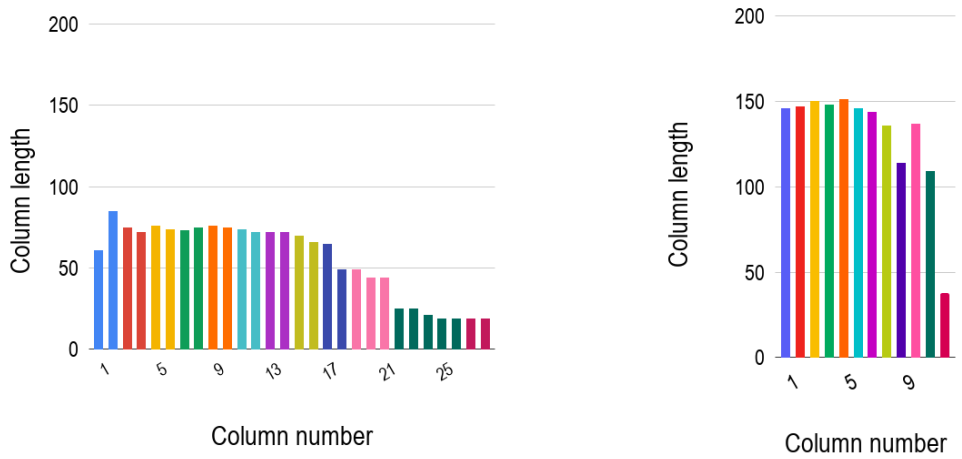



Figure 8.5.: Technical characteristics of HW1 interconnection

Figure 8.6 depicts an application of the *accumulator* algorithm to the example represented in figure 8.3 with the following parameters: $N = 100$ and $F = 1$. It can be clearly observed the algorithm reduces the number of transfers from 28 to 12. Additionally, the average column length, excluding the last one, jumps from 56 to 131. By and large, the algorithm allows to transform the original distribution shape to a more or less rectangular one which, in turn, allows to transfer the matrix in approximately equal chunks.

place for figure 8.6

Before ATHLET can send a request to NuT to start solving systems 3.1 it has to be certain that the entire Jacobian matrix has been transfered to the NuT side. For this reason, the last column transfer is done by means of the corresponding blocking




(a) Before

(b) After

Figure 8.6.: Application of *accumulator* concept to the example of figure 8.3, with $N = 100$ and $F = 1$

MPI operations. It means ATHLET gets blocked only during the last column transfer and MPI gives the execution control back only when the last piece of data has been successfully distributed among NuT processes.

## 8.3. Benchmark and Test Data

ATHLET is a dedicated industrial Computational Fluid Dynamic (CFD) package meant for simulation of thermal-hydraulic circuits in various nuclear power plant facilities. Besides the main part, the solver, it includes some pre-processing steps that allow the user to conveniently set up different simulation parameters, computational mesh, output data, etc.

Testing of new concepts and ideas directly in ATHLET can be quite cumbersome, computationally expensive and inconvenient. Therefore, a dedicated benchmark has been developed to test the *accumulator* concept.

The benchmark fully replicates all basic ideas of the default ATHLET implementation and the *accumulator* concept. It focuses only on the compressed Jacobian matrix transfer and, therefore, does not include any expensive compute-operations such as

function perturbations with seed vectors. The approach allowed to sufficiently speed up time of development, comparison and testing which, in turn, helped to design to the final concept, described in section 8.2, excluding mistakes made at earlier steps of development after several iterations of testing.

In order to mimic the real run-time ATHLET-NuT behavior during Jacobian matrix updates, a few communication patterns were recorded in ATHLET and played in the benchmark. The recordings helped to generate column vectors with the length corresponding to that in the recordings, filled with random numbers, for each data transfer. Figure 8.7 shows an example of a part of the **cube-64** communication pattern used in the study, where *COO* stands for compressed coordinate format. As it can be observed, the pattern includes both full and partial Jacobian updates.

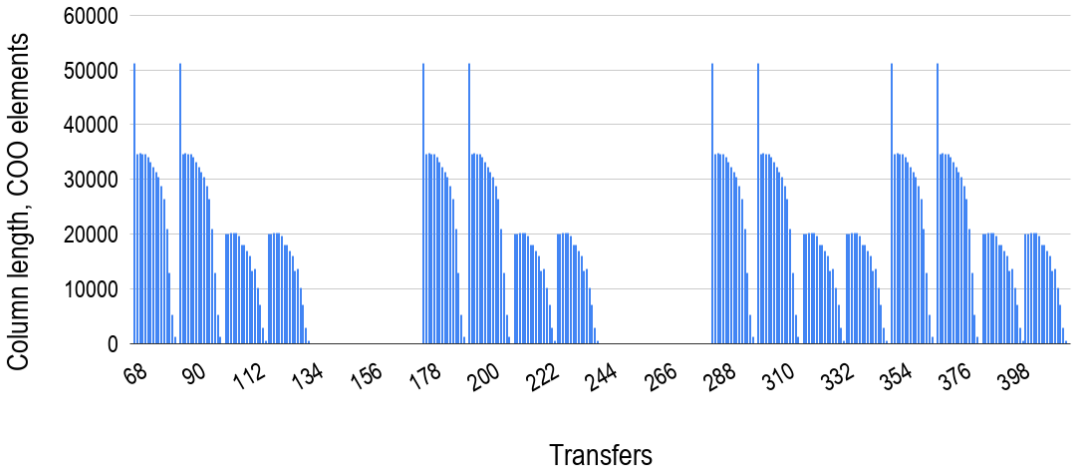



Figure 8.7.: A part of **cube-64** communication pattern

According to the *accumulator* concept, described in section 8.2, the main changes take place only on the client side and, hence, the server side remains unchanged which follows the original idea of the least code modifications. Code listing 8.3 represents an additional auxiliary class used for data accumulation. The pseudo-code of the benchmark client side is in listing 8.4.

```
# problem_size - given Jacobian matrix size
# COO - compressed matrix coordinate format
class Accumulator:
  constructor(problem_size, acomm, acomm_id):
    private:
      N = problem_size; comm = acomm; id = acomm_id
      signal = [encode("add_to_jacobian"), id]
      is_allocated = false; is_non_blocking_op_called = false
      send_buffer = []
      factor = int(read_enviroment_variable("CNUT_ACC_SIZE"))
      if factor == None:
        factor = 1
      permissible_size = factor * N
    public:
      accumulator = []

  def allocate_accumulator():
    if is_allocated == false:
      accumulator = allocate(2 * permissible_size, type(COO))
      send_buffer = allocate(2 * permissible_size, type(COO))
      is_allocated = true

  def deallocate_accumulator():
    if is_allocated == true:
      deallocate(accumulator); deallocate(send_buffer)
      is_allocated = false

  def commit():
    if accumulator.size > permissible_size:
      swap(accumulator.pointer, send_buffer.pointer)
      if is_non_blocking_op_called == true:
        MPI_Wait()
      # perform 3-way handshake
      MPI_Send(signal, 2, int, comm.head, comm)
      # send data
      MPI_Ibcast(send_buffer.size, 1, int, comm.head, comm)
      MPI_Ibcast(send_buffer.data, send_buffer.size, COO, comm.all, comm)
      is_non_blocking_op_called = true
      accumulator.content.reset("to_beginning")

  def finalize():
    if is_non_blocking_op_called == true:
      MPI_Wait()
    MPI_Send(signal, 2, int, comm.head, comm)
    MPI_Bcast(accumulator.size, 1, int, comm.head, comm)
    MPI_Bcast(accumulator.data, accumulator.size, COO, comm.all, comm)
    is_non_blocking_op_called = false
    accumulator.content.reset("to_beginning")
```

Listing 8.3: Pseudocode of the auxiliary class

```
# GIVEN PARAMETERS:
# acomm - the athlet communicator
# acomm_id - athlet identification number
# N - problem size
# recording - data structure that holds a recorded communication pattern
# COO - compressed matrix coordinate format

if global_counter != 0:
  container = Accumulator.constructor(N, acomm, acomm_id)
  container.allocate_accumulator()
  ++global_counter
  file = open("benchmark_results.txt", "w")

for column in recording:

  time_start = MPI_Wtime()

  # charge accumulator
  for i in range(column.length):
    element = generate_random_coo_element()
    container.accumulator.add(element)

  # instantiate non-blocking data broadcast

    container.commit()
  time_end = MPI_Wtime() - time_start
  file.write(column.length, time_end)

# transfer the remainder and synchronize
time_start = MPI_Wtime()
  container.finalize()
time_end = MPI_Wtime() - time_start
file.write(column.length, time_end)
```

Listing 8.4: A pseudocode of the benchmark: modified ATHLET part

## 8.4. Results

The benchmark was ran on HW1 machine where the client and server parts were distributed in three different ways, namely: within a socket, in two separate sockets of a node and in two separate nodes. Nodes of HW1 machine are connected via *infiniband* interconnect with the characteristics shown in figure 8.5. In order to estimate the affect of pure data accumulation, the benchmark, listing 8.3, was modified to use only blocking MPI operations i.e. MPI_Bcast. We denote the main benchmark as **BM1** and the modified one as **BM2** to distinguish and separately explain effects of pure data

accumulation and non-blocking data transfer.

Figure 8.8 represents results of the benchmarks obtained using **cube-64** communication pattern. The client and server parts of the code were distributed in different sockets within the same node. The factor value was chosen to be equal to 1.

Figure 8.8a shows that *accumulator* approach resulted in more that **6** times drop, from **344** to **51**, of the total number of data transfers and resulting resource acquisitions, within the range depicted on the graphs. According to **BM2** benchmark, the accumulation effect allowed to reduce the run-time by almost **9%** by means of more efficient utilization of intra-node interconnection. The obtained results also demonstrated that overall effect of both accumulation and non-blocking data transfer decreased the run-time of **BM1** benchmark in more than **26%**. Table 8.1 summarizes results obtained for all three client-server distributions within the same range of the recorded communication pattern displayed in figure 8.1.

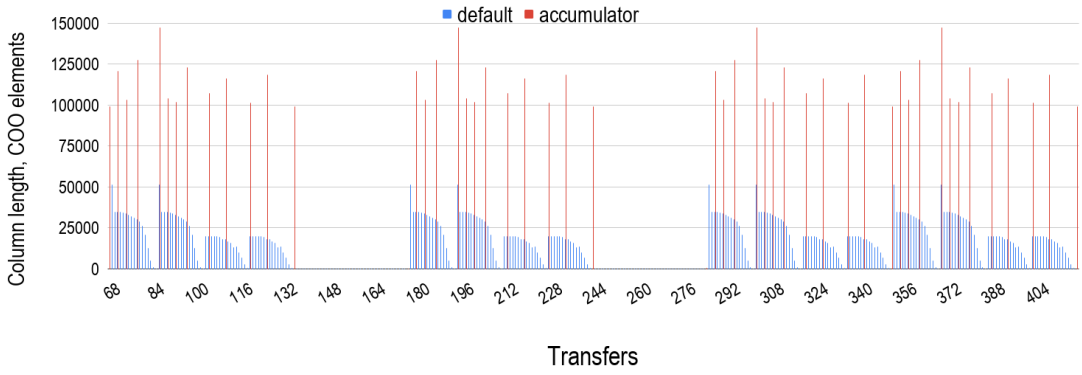
| Benchmark name | BM2, % | BM1, % |
|---|---|---|
| within a socket | 7.61 | 13.84 |
| between sockets | 9.04 | 26.26 |
| between nodes | -2.06 | 3.20 |

Table 8.1.: Time reduction in contrast to the default approach in case of **cube-64** communication pattern

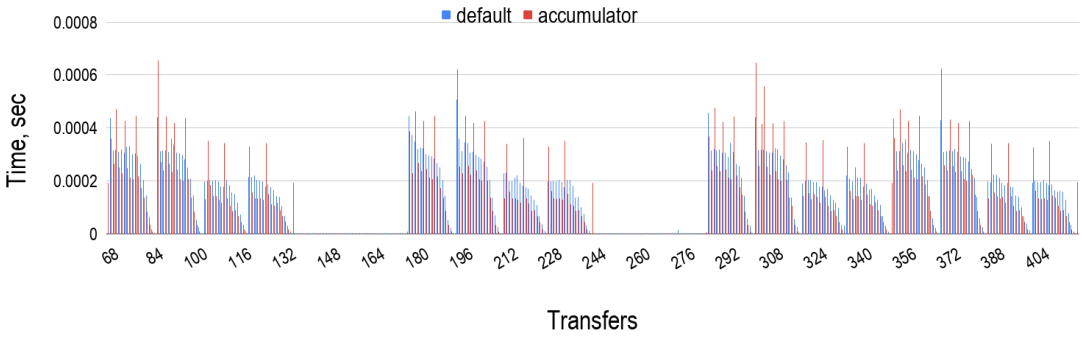
It turned out that **BM2** benchmark was slower than the default ATHLET approach in approximately **2%** in case of the inter-node communication. However, non-blocking data broadcast, according to **BM1** benchmark, helped to alleviate the slow-down and achieve almost **3%** of improvement.

Unimpressive results of non-blocking the inter-node communication can be explained by specifics of the benchmark design. In particular, time spent on generation of random matrix elements was not enough to overlap time spent on non-blocking data transfer in case of **cube-64** test case, see figure 8.9. Hence, the execution control was probably suspended by MPI library at the subsequent call of *MPI_Wait()* function.

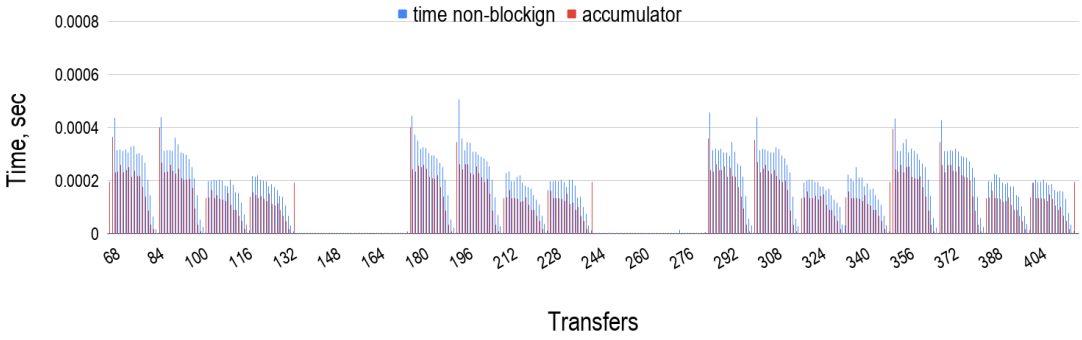
The slow-down resulting from pure data accumulation could be explained by automatic MPI protocol switching, namely: *Eager* and *Rendezvous* [Lab14]. The protocols




(a) communication pattern of **cube-64**




(b) BM2: comparison of data accumulation and blocking communication with the default approach




(c) BM1: comparison of data accumulation and non-blocking communication with the default approach

Figure 8.8.: Comparison of benchmarks running recorded **cube-64** communication pattern between two sockets within a node

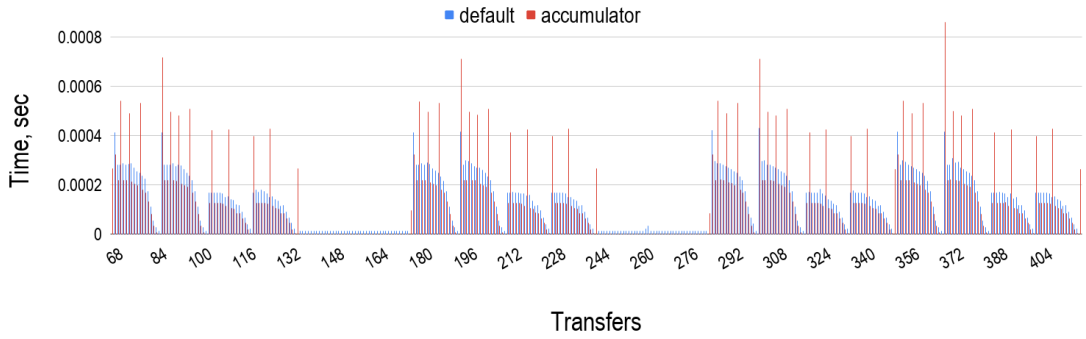



Figure 8.9.: Comparison of **BM1** benchmark with the default approach running recorded **cube-64** communication pattern between two nodes

are dedicated to small and large message transfers, respectively, where the quantitative measure of the message size is defined by a concrete implementation of the MPI standard and can be controlled through dedicated environment variables.

Similar results were observed for **cube-645** test case where the number of equations was approximately $10^6$ and the average compressed Jacobian column length reached around $1.7 \cdot 10^5$ elements. In case of the inter-node communication, **BM1** benchmark again showed performance degradation by **6.35%** whereas non-blocking data broadcast improved run-time by **23.21%**. Such performance jump, from **-6.35%** to **23.21%**, is explained by the fact that time spent on generation of random elements was enough to hide the corresponding data transfer and overheads.

Ideas, expressed in **BM1** benchmark, listings 8.3 and 8.4, were successfully implemented in NuT: the client side of NuT located in ATHLET. Several simulation scenarios were taken for the final verification and performance testing, namely: **cube-64**, **k3-2** and **pwr3d**. Verification of the modified code did not detect any deviations of numerical results from the original implementation. Additionally, all tests showed considerable improvement with respect to the communication time. As an example, time spent on communication between ATHLET and NuT during compressed Jacobian transfers decreased by **66.17%**, **76.03%** and **42.55%** for intra-socket, intra-node and inter-node client-server process allocation respectively, for **pwr3d** scenario, taking it as the most representative simulation case known in GRS.

However, the overall improvement of applied changes achieved only **0.14%** in average, regardless of client-server allocation. Profiling showed the communication part of

the original implementation took around **0.24%** of the total time spent on the matrix evaluations and transfers. This fact explains that negligible overall performance gain of the modified code that was observed in all conducted tests.

## 8.5. Conclusion

In this part of the study, we have designed and implemented the *accumulator* concept for efficient sparse compressed Jacobian matrix transfer between ATHLET and NuT. The concept is rather simple and did not require drastic changes of the existing software design and architecture. In spite of simplicity, the concept allows to significantly reduce the communication time i.e. by almost **60%**. The overall performance gain comes from three different sources:

1. efficient utilization of interconnection
2. reduced number of handshakes and, as a result, the amount of NuT process synchronizations
3. overlap of communications with computations

The study has shown non-blocking data transfer is the main source of the performance gain. Efficient bandwidth utilization can additionally give **7-9%** of improvement when applications work within the same compute-node.

One can experience a slight slow-down from pure data accumulation in case of the inter-node communication due to probable MPI protocol switching. However, as it has been shown, it is always compensated by means of communication/computation overlap.

The final tests have shown the concept does not give a considerable overall improvement because the computation part takes almost **99.8%** of the total execution time of the corresponding part of the source code. However, results can be much better in case of multi-client operation of NuT; especially when clients share common NuT processes. Reduced number data transfers results in reduced amount of handshakes which is always accompanied by the resource acquisition mechanism, described in section 2.3. Unfortunately, it is difficult to design and prepare valid tests to verify this statement.

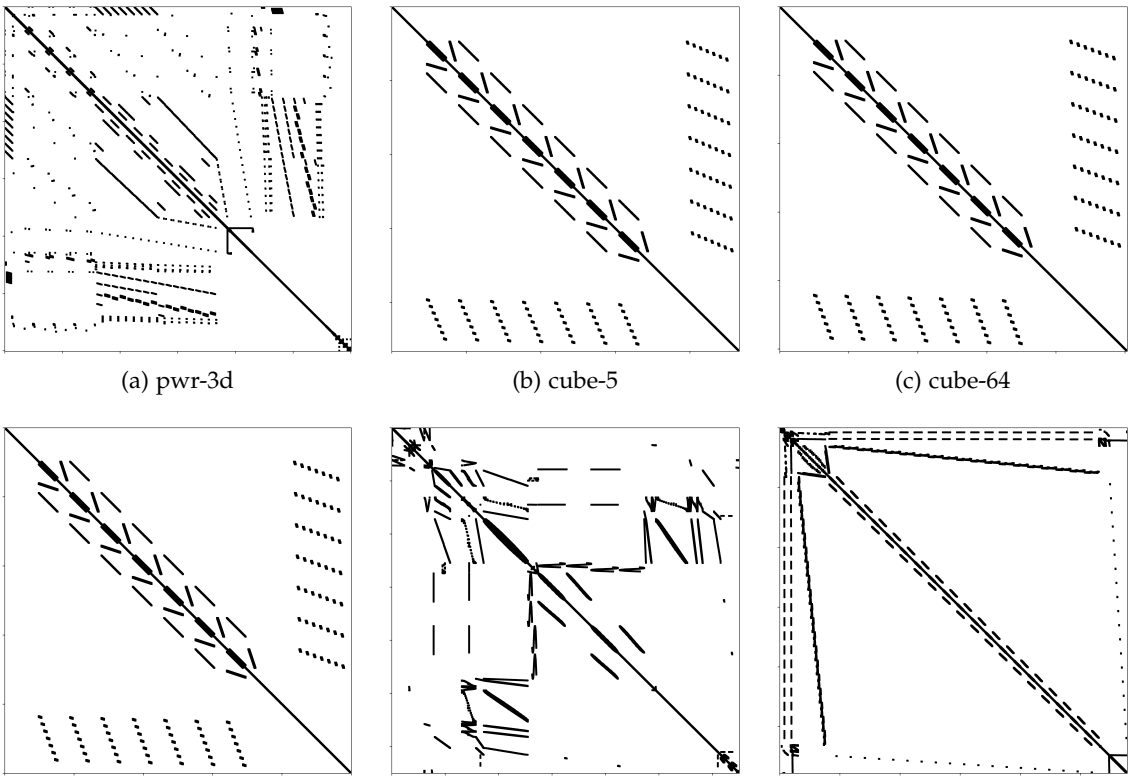
By and large, verification of the modified code has not detected any deviations in results. The new concept has always resulted in a slight overall performance gain. The

study has also shown the main bottleneck is, indeed, the non-linear function evaluation.

It is worth mentioning that only a sequential ATHLET code, running in a single core, has been available for this study. However, there exists a parallel version of ATHLET multi-threaded with OpenMP. Therefore, the results can be even better because of the reduced fraction of function evaluation time. This fact also shows that development and performance tuning of ATHLET is constantly in progress and is being done in parallel among several departments of GRS, covering different areas of the program source code.

# Appendices

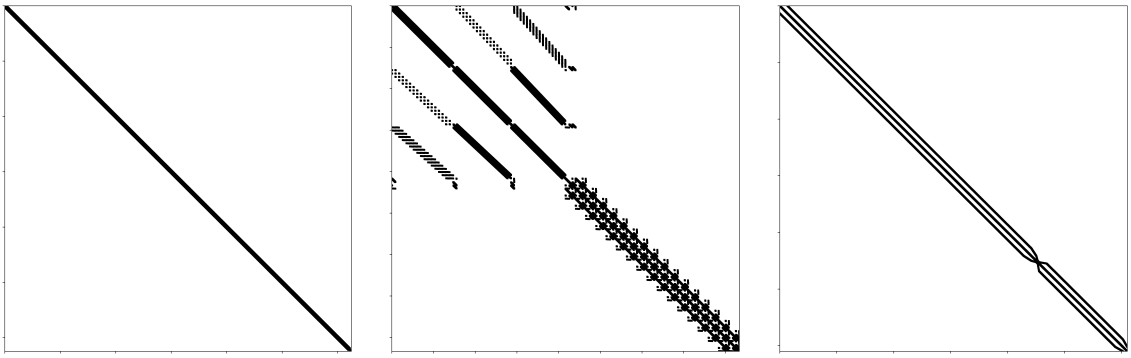
# A. Sparsity Patterns of Matrix Sets



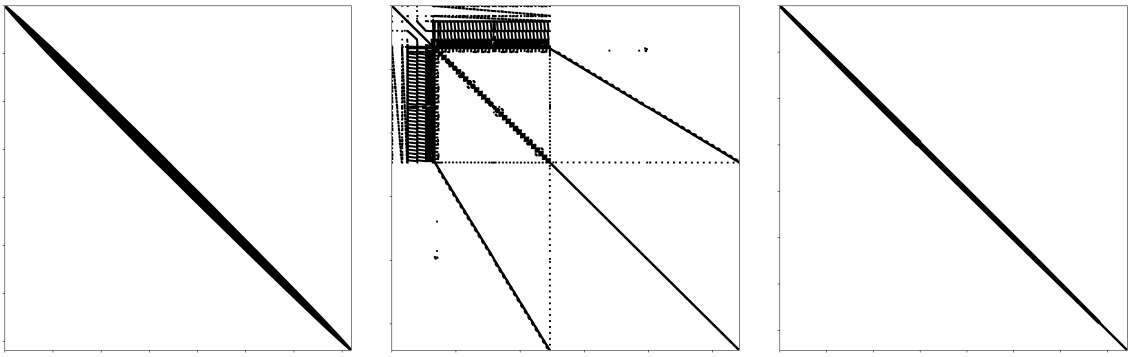
(a) pwr-3d (b) cube-5 (c) cube-64

(d) cube-645 (e) k3-2 (f) k3-18

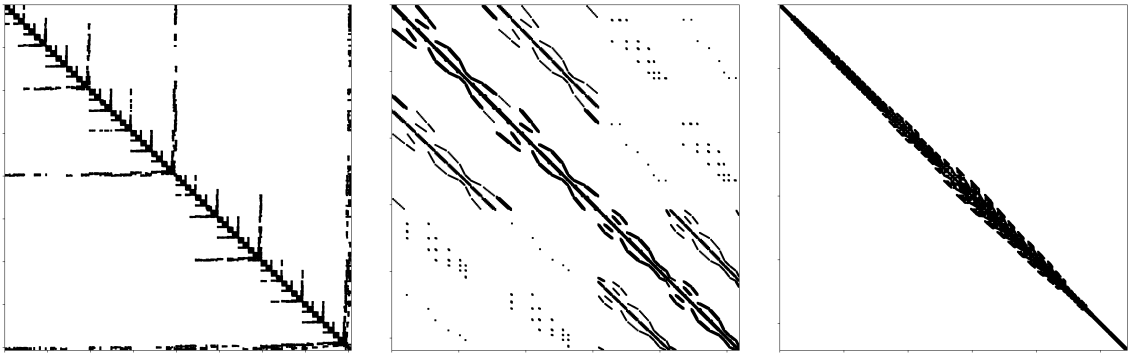
Figure A.1.: Sparsity patterns of GRS matrix set



(a) cant (b) consph (c) CurlCurl_3



(d) Geo_1438 (e) memchip (f) PFlow_742



(g) pkustk10 (h) torso3 (i) x104

Figure A.2.: Sparsity patterns of SuiteSparse matrix set

# B. 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 |

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 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 B.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 |

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 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 |

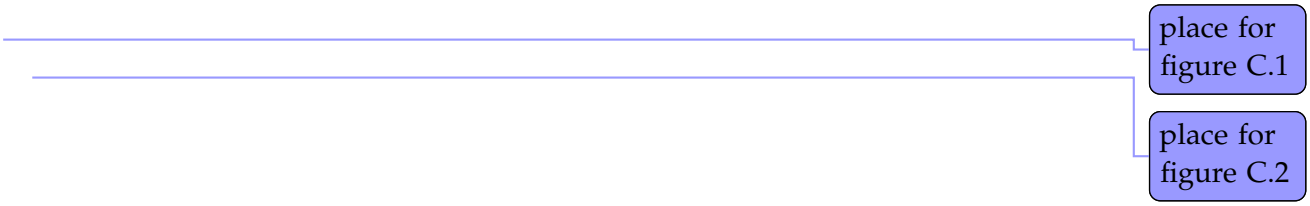
Table B.2.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **k3-2** (130101 equations)

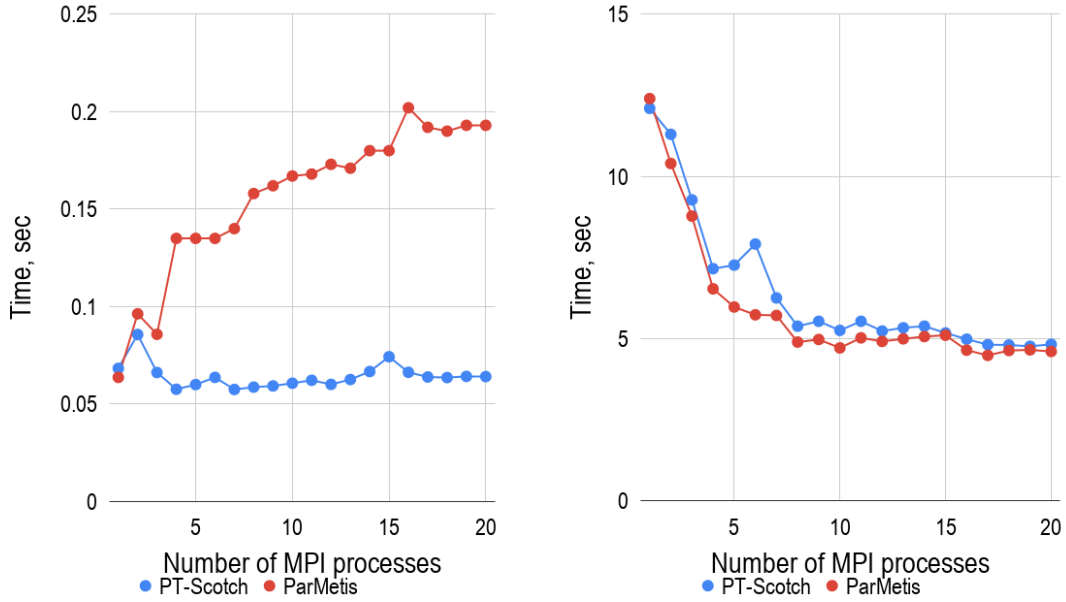
| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 1 | 1.52E+01 | 1.61E+01 | crashed |
| 2 | 1.13E+01 | 1.13E+01 | crashed |
| 3 | 1.00E+01 | 1.03E+01 | crashed |
| 4 | 9.29E+00 | 1.05E+01 | crashed |
| 5 | 8.85E+00 | 9.84E+00 | crashed |
| 6 | 8.43E+00 | 8.99E+00 | crashed |
| 7 | 8.64E+00 | 9.69E+00 | 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 |

| MPI | MUMPS | PaStiX | SuperLU |
|---|---|---|---|
| 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 B.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)

# C. Choice of Fill Reducing Reordering

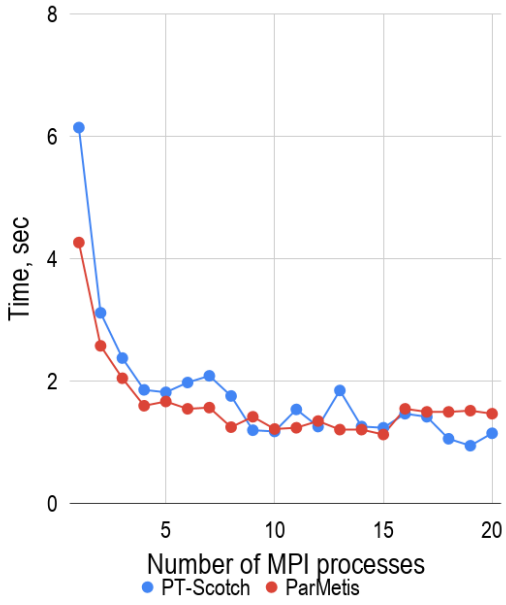




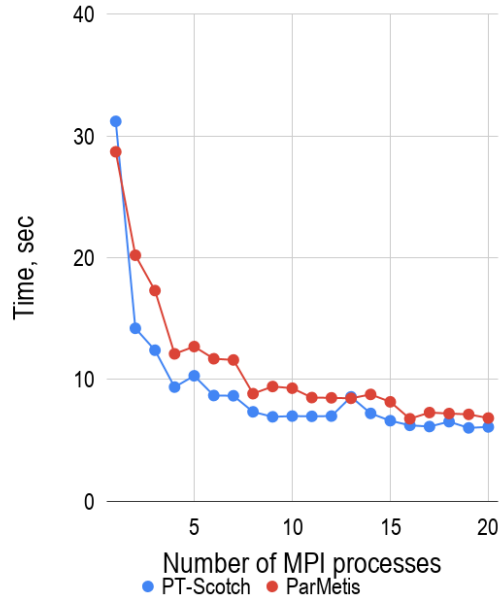

(a) cube-5




(b) cube-645

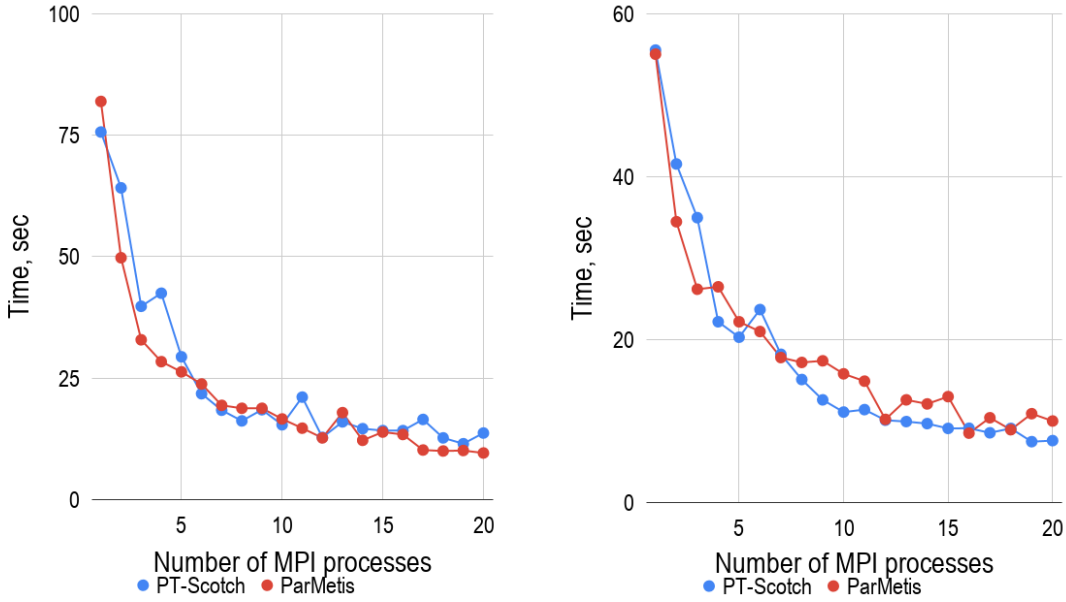



(c) cant




(d) memchip

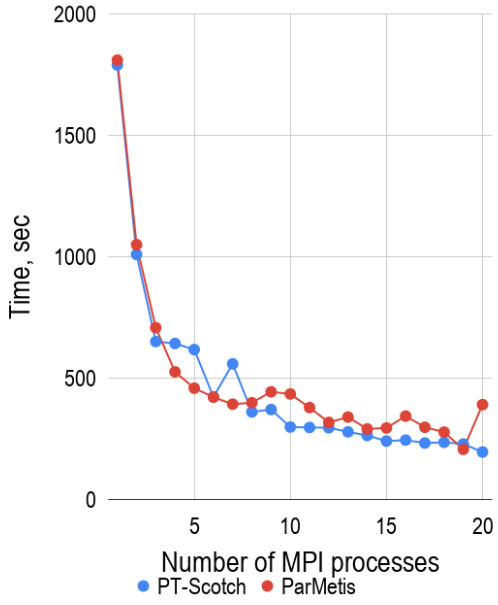
Figure C.1.: Comparison of different fill-reducing algorithms



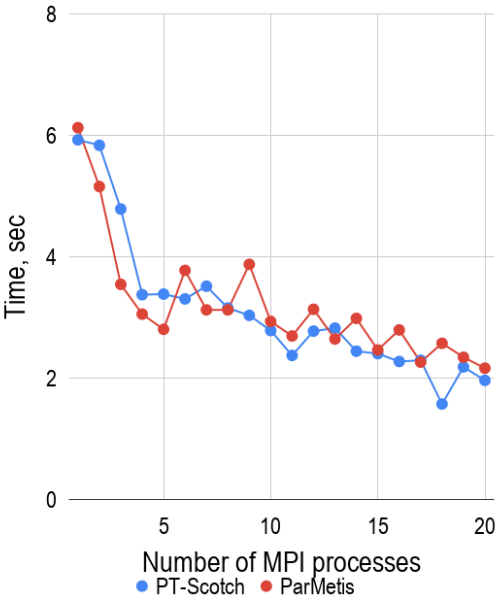

(a) torso3




(b) consph



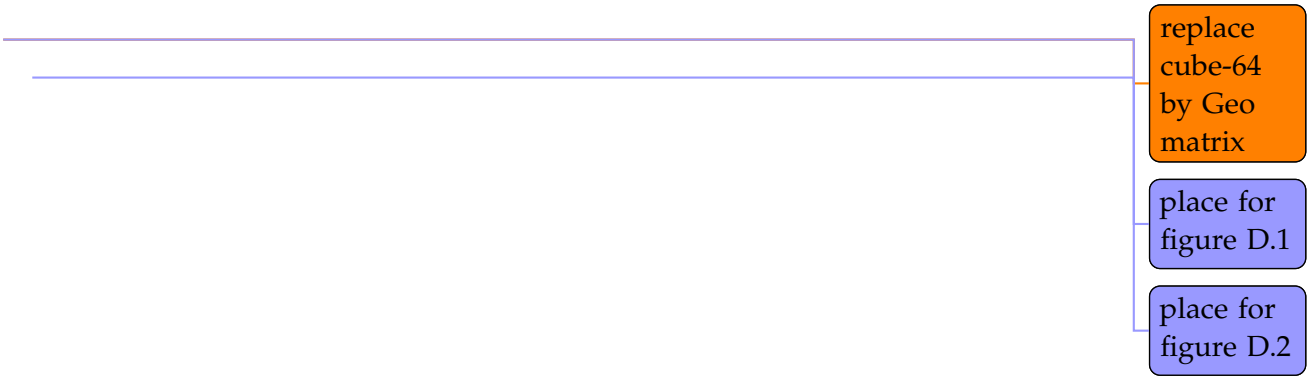

(c) CurlCurl_3

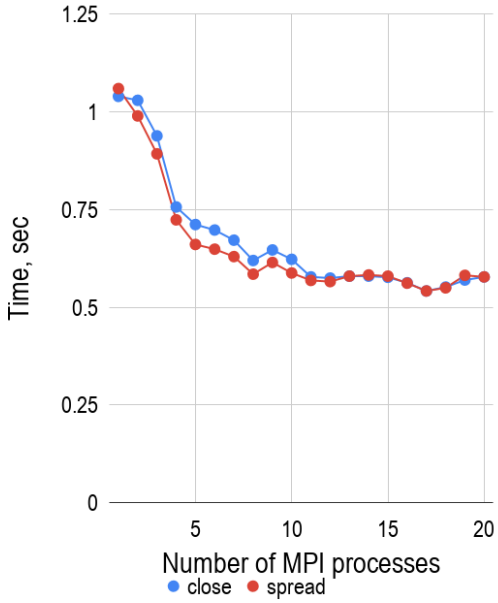



(d) x104

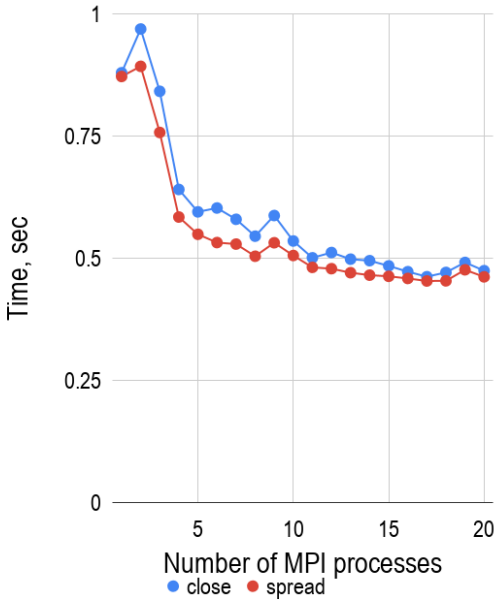
Figure C.2.: Comparison of different fill-reducing algorithms

# D. MUMPS: Process Pinning

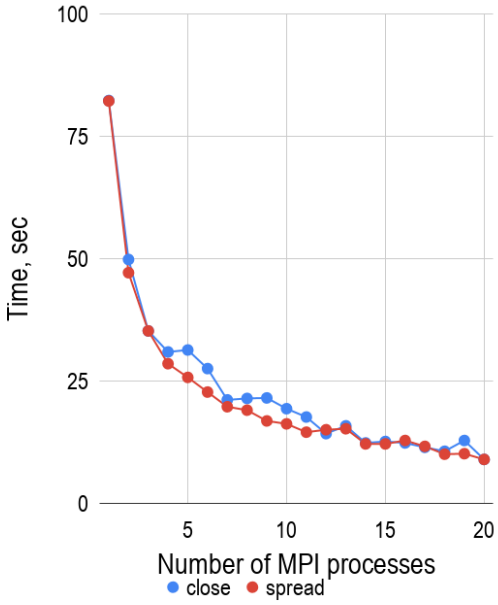




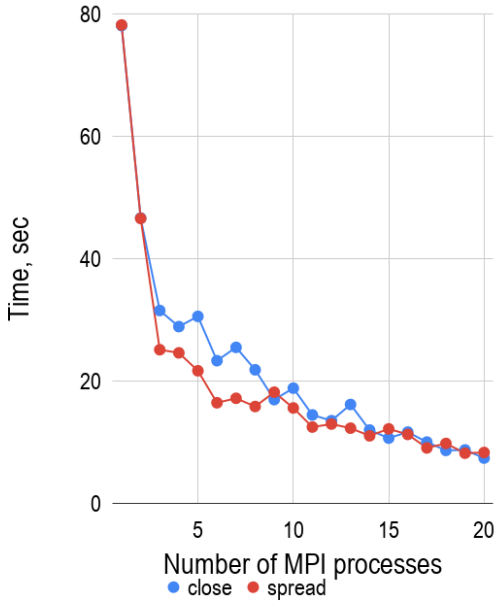

(a) HW1 - cube-64




(b) HW2 - cube-64

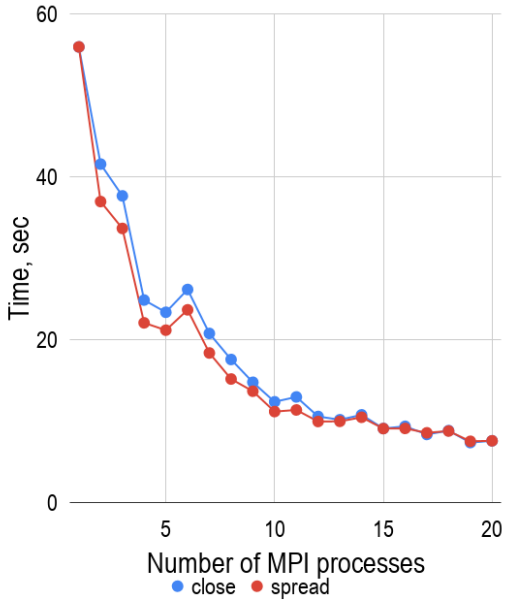



(c) HW1 - torso3

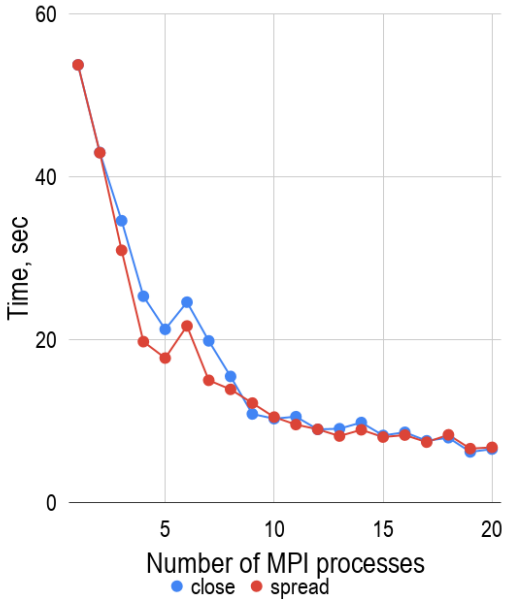



(d) HW2 - torso3

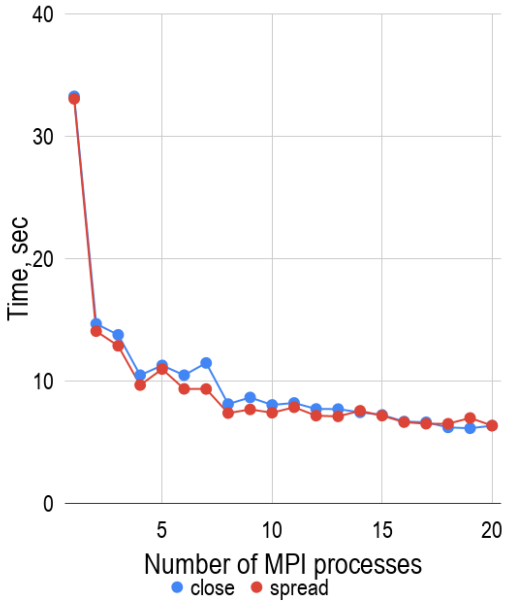
Figure D.1.: Comparison of *close* and *spread* pinning strategies



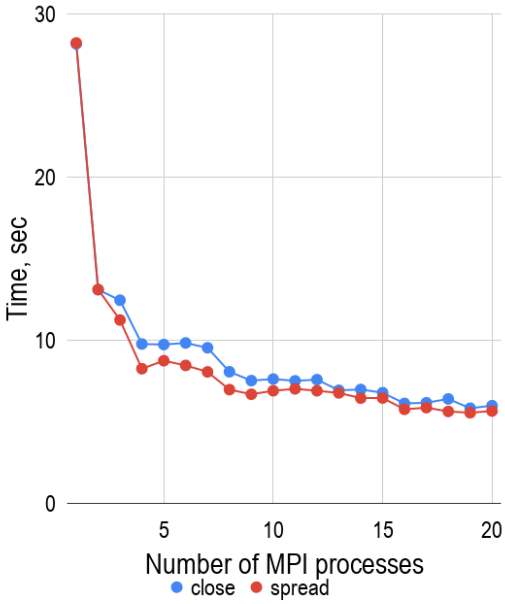

(a) HW1 - consph




(b) HW2 - consph



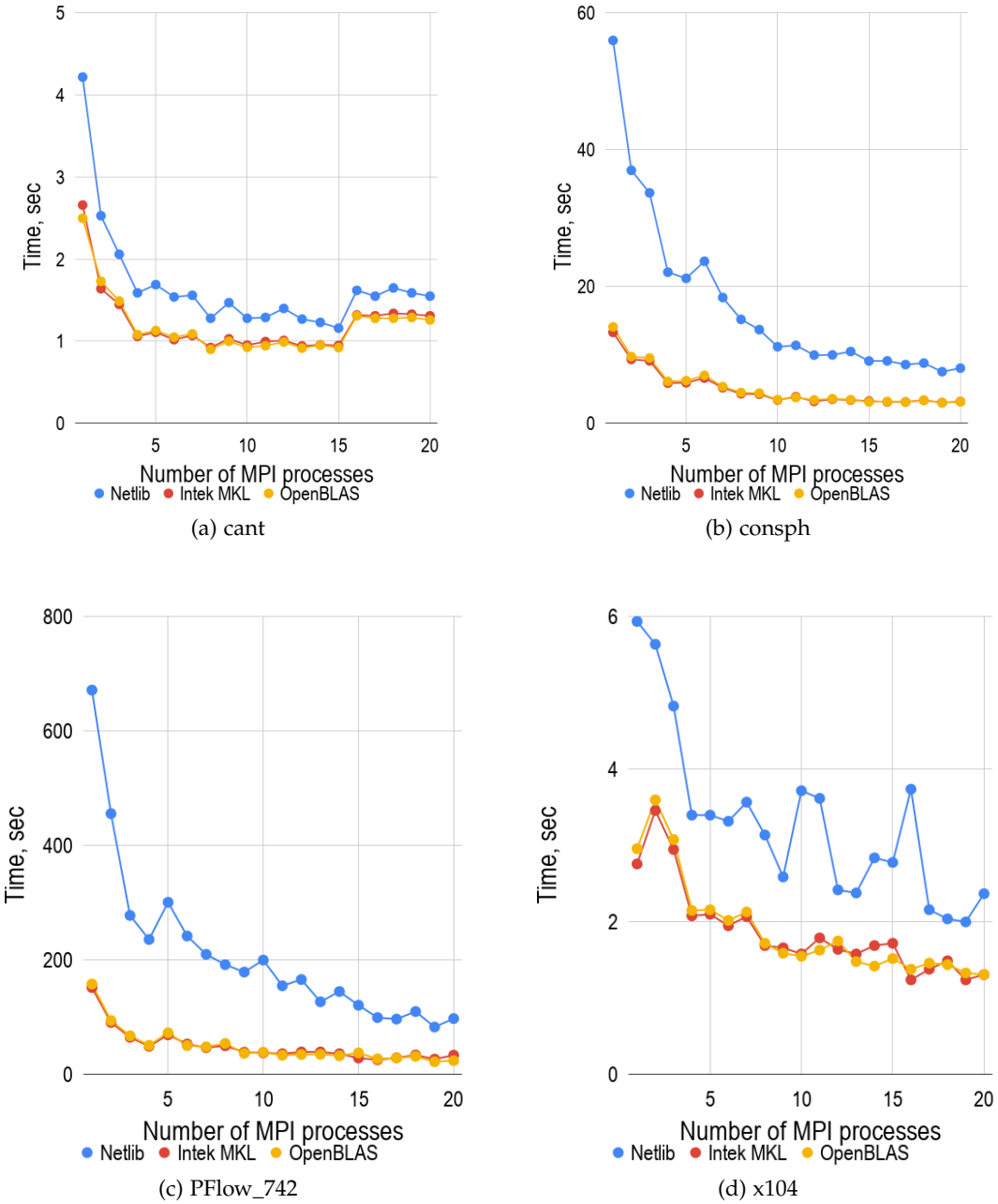

(c) HW1 - memchip_3




(d) HW2 - memchip

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

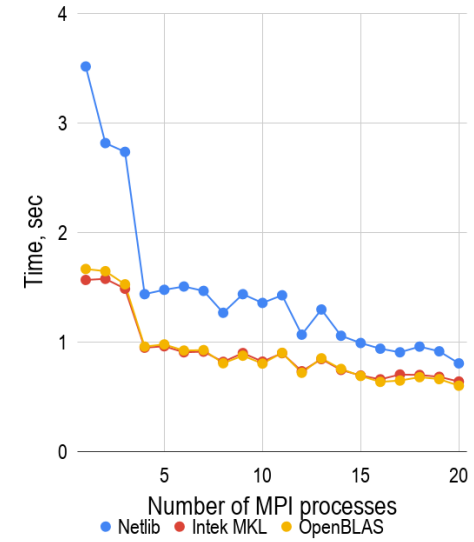
# E. Choice of BLAS Library



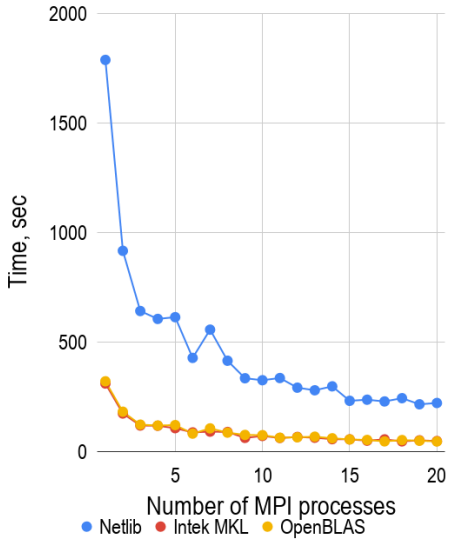

(a) cant (b) consph

(c) PFlow_742 (d) x104

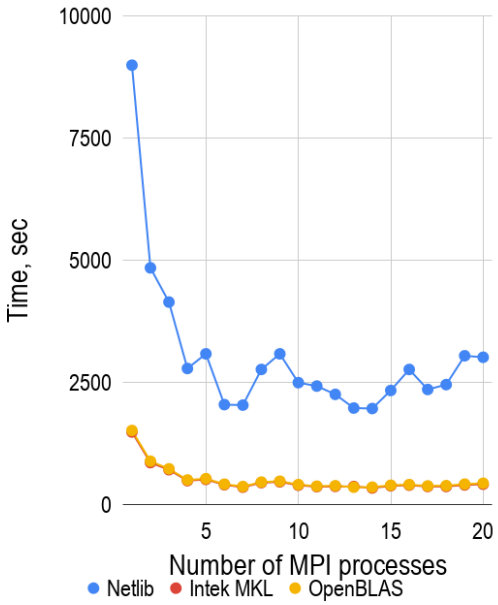
Figure E.1.: MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine




(a) pkustk10




(b) CurlCurl_3




(c) Geo_1438

Figure E.2.: MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine

# List of Figures


2.1. ATHLET: one dimensional finite volume formulation of the problem . . 4
2.2. A general view on the W-method solver implemented in ATHLET . . . 5
2.3. NuT process groups . . . . . 7
2.4. ATHEL-NuT software coupling . . . . . 8

4.1. An example of process pinning of 5 MPI processes with 2 OpenMP threads per rank in case of HW1 hardware . . . . . 14

5.1. Performance of sparse triangular solve [Jos+98] . . . . . 18
5.2. GMRES strong scaling speed-up . . . . . 19
5.3. An example of a sparse matrix and its Cholesky factor [Liu92] . . . . . 23
5.4. The elimination tree for the matrix example in Figure 5.3 [Liu92] . . . . 24
5.5. Information flow of the multifrontal method . . . . . 26
5.6. An example of matrix postordering from [Liu92] . . . . . 28
5.7. The stack contents for the postordering [Liu92] . . . . . 28
5.8. An example of a supernodal elimination tree [Liu92] . . . . . 30
5.9. Parallel steps of the multifrontal method based on the example in Figures 5.8 . . . . . 31
5.10. Simple parallel models of the multifrontal method . . . . . 32
5.11. Theoretical speed-up . . . . . 33
5.12. MUMPS parallelism management in case of 4 PEs [OA07] . . . . . 35
5.13. Comparison between model 1 and numerical factorization of the matrix ***memchip*** using MUMPS library . . . . . 36
5.14. Sparsity structure of the matrix ***memchip*** before and after fill-in reduction 36

6.1. 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) . . . . . 46

7.1. MUMPS: static and dynamic scheduling [LEx12] . . . . . 50
7.2. Comparison of different fill-reducing algorithms . . . . . 52
7.3. MUMPS-ParMetis parallel performance in case of relatively small matrices 53

7.4. Profiling of MUMPS library with using ParMetis as a fill-in reducing algorithm in case of factorization of relatively small matrices . . . . . 54
7.5. Comparison of *close* and *spread* pinning strategies . . . . . 57
7.6. Comparison of *close* and *spread* pinning strategies . . . . . 58
7.7. Comparison of *close* and *spread* pinning strategies . . . . . 59
7.8. MUMPS: static and dynamic scheduling . . . . . 61
7.9. MUMPS: An example of a type 2 node factorization . . . . . 62
7.10. MUMPS: comparison of different BLAS libraries with using GRS matrix set on HW1 machine . . . . . 64
7.11. MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine . . . . . 65
7.12. Anomalies of MUMPS-OpenBLAS configuration running with 2 OpenMP threads per MPI process . . . . . 71
7.13. A MUMPS-OpenBLAS thread conflict in case of *k3-18* matrix factorization (green - application threads, red - system threads) . . . . . 71
7.14. MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine . . . . . 77
7.15. MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine . . . . . 80

8.1. An example of matrix coloring and compression [GMP05] . . . . . 81
8.2. An example of an efficient Jacobian matrix partitioning [GMP05] . . . . 83
8.3. Column length distribution of the example of figure 8.2 . . . . . 83
8.4. Accumulator concept . . . . . 86
8.5. Technical characteristics of HW1 interconnection . . . . . 87
8.6. Application of *accumulator* concept to the example of figure 8.3, with $N = 100$ and $F = 1$ . . . . . 88
8.7. A part of **cube-64** communication pattern . . . . . 89
8.8. Comparison of benchmarks running recorded **cube-64** communication pattern between two sockets within a node . . . . . 93
8.9. Comparison of **BM1** benchmark with the default approach running recorded **cube-64** communication pattern between two nodes . . . . . 94

A.1. Sparsity patterns of GRS matrix set . . . . . 98
A.2. Sparsity patterns of SuiteSparse matrix set . . . . . 99

C.1. Comparison of different fill-reducing algorithms . . . . . 103
C.2. Comparison of different fill-reducing algorithms . . . . . 104

D.1. Comparison of *close* and *spread* pinning strategies . . . . . 106

D.2. Comparison of *close* and *spread* pinning strategies . . . . . 107

E.1. MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine . . . . . 109
E.2. MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine . . . . . 110

# List of Tables


1.1. A general overview of software developed by GRS [Ges] . . . . . 2

4.1. GRS matrix set . . . . . 12
4.2. SuiteSparse matrix set . . . . . 13
4.3. Hardware specification . . . . . 14

5.1. Potential speed-up of linear and quadratic models . . . . . 33
5.2. Efficiency of linear and quadratic models using 20 PEs . . . . . 33
5.3. Distribution workload per level in case model 1 and 2 . . . . . 34

6.1. List of packages to solve sparse linear systems using direct methods on distributed memory parallel machines [Li18], [Bal+18] . . . . . 40
6.2. Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **cube-5** (9352 equations) . . . 44
6.3. Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **cube-64** (100657 equations) . 45
6.4. Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix **k3-18** (1155955 equations) . . 45

7.1. GRS matrix set: assignment of matrices to a specific fill-in reducing algorithm based on parallel performance of flat-MPI tests . . . . . 54
7.2. SuiteSparse matrix set: assignment of matrices to a specific fill-in reducing algorithm based on parallel performance of flat-MPI tests . . . . . 55
7.3. Analysis and comparison of MUMPS performance at the saturation point between HW1 and HW2 for GRS matrix set . . . . . 56
7.4. Analysis and comparison of MUMPS performance at the saturation point between HW1 and HW2 for SuiteSparse matrix set. *ROM - run out of memory . . . . . 60
7.5. Commercial and open source BLAS implementations [Wik18b] . . . . . 66
7.6. Comparison of different MUMPS-BLAS configurations applied to GRS matrix set . . . . . 67
7.7. Comparison of different MUMPS-BLAS configurations applied to SuiteSparse matrix set . . . . . 67

7.8. Compassion of different hybrid MPI/OpenMP modes used for GRS matrix set on HW1 . . . . . 72
7.9. Compassion of different hybrid MPI/OpenMP modes used for GRS matrix set on HW2 . . . . . 72
7.10. Compassion of different hybrid MPI/OpenMP modes used for SuiteSparse matrix set on HW1 . . . . . 73
7.11. Compassion of different hybrid MPI/OpenMP modes used for SuiteSparse matrix set on HW2 *ROM - run out of memory . . . . . 73
7.12. Matrix type reference table . . . . . 79

8.1. Time reduction in contrast to the default approach in case of **cube-64** communication pattern . . . . . 92

B.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) . . . 100
B.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) . . . . 100
B.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) 101

# Bibliography


[06] *Netlib Frequently Asked Questions*. 2006. URL: http://www.netlib.org/misc/faq.html#2.1.

[09] *Krylov subspace projection methods: Rayleigh-Ritz procedure*. 2009. URL: http://web.cs.ucdavis.edu/~bai/Winter09/krylov.pdf.

[11] *SuperLU Users' Guide*. 2011.

[17] *MUltifrontal Massively Parallel Solver, Users' guide*. 2017.

[ADD89] M. Arioli, J. Demmel, and I. S. Duff. "Solving sparse linear systems with sparse backward error". In: *SIAM Journal on Matrix Analysis and Applications* (1989).

[ADD96] P. R. Amestoy, T. A. Davis, and I. S. Duff. "An Approximate Minimum Degree Ordering Algorithm". In: *SIAM Journal on Matrix Analysis and Applications* (1996).

[Ame+02] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. *MUMPS: A Multifrontal Massively Parallel Solver*. 2002. URL: https://www.ercim.eu/publication/Ercim_News/enw50/amestoy.html.

[Ame+98] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and P. Plecháč. "PARASOL: An Integrated Programming environment for Parallel Sparse Matrix Solvers". In: *PINEAPL Workshop, A Workshop on the Use of Parallel Numerical Libraries in Industrial End-user Applications*. CERFACS, Toulouse, France, 1998.

[Ame97] P. Amestoy. "Recent progress in parallel multifrontal solvers for unsymmetric sparse matrices". In: *Proceedings of the 15th World Congress on Scientific Computation, Modelling and Applied Mathematics, IMACS*. Vol. 97. 1997.

[Bal+18] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, D. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, and H. Zhang. *PETSc Web page*. http://www.mcs.anl.gov/petsc. 2018.

[Blo+13] D. Blom, H. Bijl, P. Birken, A. Meister, and A. Van Zuijlen. "Rosenbrock time integration for unsteady flow simulations". In: *Coupled Problems* (2013).

[CCV18] J. Cornelis, S. Cools, and W. Vanroose. "The Communication-hiding Conjugate Gradient Method with Deep Pipelines". In: *arXiv:1801.04728 [cs.DC]* (2018).

[CL10] I. Chowdhury and J.-Y. L'Excellent. "Some experiments and issues to exploit multicore parallelism in a distributed-memory parallel sparse direct solver". PhD thesis. INRIA, 2010.

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. "The recursion-tree method for solving recurrences". In: *Introduction to Algorithms*. 2009.

[DGL89] I. S. Duff, R. G. Grimes, and J. G. Lewis. "Sparse Matrix Test Problems". In: *ACM Trans. Math. Softw.* 15.1 (Mar. 1989), pp. 1–14. ISSN: 0098-3500. DOI: 10.1145/62038.62043.

[DH11] T. A. Davis and Y. Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

[DK01] I. S. Duff and J. Koster. "On algorithms for permuting large entries to the diagonal of a sparse matrix". In: *SIAM Journal on Matrix Analysis and Applications* 22.4 (2001), pp. 973–996.

[DK99] I. S. Duff and J. Koster. "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices". In: *SIAM Journal on Matrix Analysis and Applications* 20.4 (1999), pp. 889–901.

[DP05] I. S. Duff and S. Pralet. "Strategies for scaling and pivoting for sparse symmetric indefinite problems". In: *SIAM Journal on Matrix Analysis and Applications* 27.2 (2005), pp. 313–340.

[DR83] I. S. Duff and J. K. Reid. "The Multifrontal Solution of Indefinite Sparse Symmetric Linear". In: *ACM Transactions on Mathematical Software, (TOMS)* (1983).

[Eur18] European Commission. *Nuclear Energy: Safe nuclear power*. 2018. URL: https://ec.europa.eu/energy/en/topics/nuclear-energy.

[Ges] Gesellschaft für Anlagenund Reaktorsicherheit (GRS) gGmbH. *GRS official website*. URL: https://www.grs.de.

[Ges16] Gesellschaft für Anlagenund Reaktorsicherheit (GRS) gGmbH. *ATHLET 3.1A: Program Overview*. 2016. URL: https://software.intel.com/en-us/mkl-developer-reference-c-overview-of-scalapack-routines.

[GKG09] A. Gupta, S. Koric, and T. George. "Sparse matrix factorization on massively parallel computers". In: *IEEE Xplore* (2009).

[GLU03] A. Guermouche, J.-Y. L'Excellent, and G. Utard. "On the memory usage of a parallel multifrontal solver". In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE. 2003, 8–pp.

[GMP05] A. H. Gebremedhin, F. Manne, and A. Pothen. "What color is your Jacobian? Graph coloring for computing derivatives". In: *SIAM review* 47.4 (2005), pp. 629–705.

[GN89] G. Geist and E. Ng. "Task scheduling for parallel sparse Cholesky factorization". In: *International Journal of Parallel Programming* 18.4 (1989), pp. 291–314.

[Gup00] A. Gupta. *WSMP: Watson Sparse Matrix Package, Part II - direct solution of general systems*. Research Report. IBM T. J. Watson Research Center, 2000.

[IB70] Irons and Bruce. "A frontal solution program for finite element analysis". In: *International Journal for Numerical Methods in Engineering* (1970).

[Jor19] Jordan Hanania and Braden Heffernan and Jenden, James and Lefsrud, Nathan and Lloyd, Ellen and Stenhouse, Kailyn and Toor, Jasdeep and Donev, Jason. *Nuclear power plant - Energy Education*. 2019. URL: %%20https://energyeducation.ca/encyclopedia/Nuclear_power_plant.

[Jos+98] M. V. Joshi, A. Guptam, G. Karypis, and V. Kumar. "A High Performance Two Dimensional Scalable Parallel Algorithm for Solving Sparse Triangular Systems". In: *IEEE Xplore* (1998).

[KBK16] J. H. Kwack, G. Bauer, and S. Koric. "Performance Test of Parallel Linear Equation Solvers on Blue Waters – Cray XE6/XK7 system". In: May 2016.

[KK09] G. Karypis and V. Kumar. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0*. http://www.cs.umn.edu/~metis. University of Minnesota, Minneapolis, MN, 2009.

[Lab14] L. L. N. Laboratory. *MPI Performance Topics: MPI Message Passing Protocols*. 2014. URL: https://computing.llnl.gov/tutorials/mpi_performance/.

[LEx12] J.-Y. L'Excellent. "Multifrontal methods: parallelism, memory usage and numerical aspects". PhD thesis. Ecole normale supérieure de lyon-ENS LYON, 2012.

[Li18] X. Li. *Direct Solvers for Sparse Matrices*. 2018. URL: http://crd-legacy.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf.

[Liu86] J. W. H. Liu. "On the storage requirement in the out-of-core multifrontal method for sparse factorization". In: *ACM Trans. Math. Software* (1986).

[Liu88] J. W. H. Liu. "Equivalent Sparse Matrix Reordering by Elimination Tree Rotations". In: *SIAM J. Sci. Statist. Comput.* (1988).

[Liu92] J. W. H. Liu. "The Multifrontal Method for Sparse Matrix Solution: Theory and Practice". In: *SIAM* (1992).

[LS13] J.-Y. L'Excellent and M. W. Sid-Lakhdar. "Introduction of shared-memory parallelism in a distributed-memory multifrontal solver". PhD thesis. INRIA, 2013.

[OA07] B. Olivier and G. Abdou. "Task Scheduling for Parallel Multifrontal Methods". In: *Proceedings of the 13th International Euro-Par Conference on Parallel Processing* (2007).

[Pel08] F. Pellegrini. "Scotch and libScotch 5.1 user's guide". In: (2008).

[PT01] A. Pothen and S. Toledo. "Elimination Structures in Scientific Computing". In: *CRC Press, LLC* (2001).

[RBH12] S. Rajamanickam, E. G. Boman, and M. A. Heroux. "ShyLU: A Hybrid-Hybrid Solver for Multicore Platforms". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (2012).

[Sch01] J. Schulze. "Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods". In: *BIT Numerical Mathematics* 41.4 (2001), pp. 800–841.

[SS86] Y. Saad and M. H. Schultz. "GMRES: A Generalized Minimal Residual Algorithm For Solving Nonsymmetric Linear Systems". In: *SIAM Journal on Scientific and Statistical Computing* (1986).

[Tim07] Time for change. *Pros and cons of nuclear power*. 2007. URL: %%20https://timeforchange.org/pros-and-cons-of-nuclear-power-and-sustainability.

[Wik18a] Wikipedia contributors. *Amdahl's law — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-November-2018]. 2018.

[Wik18b] Wikipedia contributors. *Basic Linear Algebra Subprograms — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-December-2018]. 2018.

[Wik18c] Wikipedia contributors. *Portable, Extensible Toolkit for Scientific Computation — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-January-2019]. 2018.

[Wu12] C. J. Wu. "Partial Left-Looking Structured Multifrontal Factorization and Algorithms for Compressed Sensing". PhD thesis. University of California, 2012.