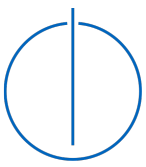
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computational Science and Engineering

Configuration of a linear solver for linearly implicit time integration and efficient data transfer in parallel thermo-hydraulic computations

Ravil Dorozhinskii



DEPARTMENT OF INFORMATICS

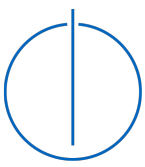
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computational Science and Engineering

Configuration of a linear solver for linearly implicit time integration and efficient data transfer in parallel thermo-hydraulic computations

Konfiguration eines Lösers für linear-implizite Zeitintegration und effizienter Datentransfer in parallelen thermohydraulischen Berechnungen

Author:	Ravil Dorozhinskii
Supervisor:	Prof. Dr. Thomas Huckle
Advisors:	Dr. rer. nat. Tim Steinhoff, Dr. rer. nat. Tobias Neckel
Submission Date:	07 March 2019



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 07 March 2019 Ravil Dorozhinskii

Abstract

Key words. *numerical time integration, direct sparse linear methods, multifrontal methods, MUMPS, BLAS, parallel performance, distributed-memory computations, multi-threading, MPI, OpenMP, non-blocking communication*

An application of linearly implicit methods for time integration of stiff systems of ODEs results in solving sparse systems of linear equations. An optimal selection and configuration of a parallel linear solver can considerably accelerate the time integration process. A comparison of iterative and direct sparse linear solvers has shown that direct ones are the most suitable for this purpose because of their natural robustness to ill-conditioned linear systems that can occur during numerical time integration. Testing of different direct sparse solvers applied to systems generated by ATHLET software has revealed that MUMPS, an implementation of the multifrontal method, performs better than the others in terms of the overall parallel execution time.

In this study, we have mainly focused on configuring MUMPS with the aim of improving parallel performance of the solver for thermo-hydraulic computations within a single node of GRS compute-cluster. However, the overall approach, proposed in the study, may be considered as a general framework for a selection and adaptation of a linear sparse solver for solving problem-specific systems of linear equations on distributed-memory machines.

Additionally, we have shown that an intelligent application of non-blocking MPI communication in some parts of the existing thermo-hydraulic simulation code, ATHLET, can additionally solve issues of inefficient data transfer preserving the current software design and implementation without drastic changes of the source code.

Acronyms

ATHLET Analysis of THermal-hydraulics of LEaks and Transients.

BLAS Basic Linear Algebra Subprograms.

GEMM GEneral Matrix-matrix Multiplication (BLAS subroutine).

GETRF GEneral TRiangular Factorization (LAPACK subroutine).

GMRES General Minimum RESidual method.

GRS Gesellschaft für anlagen- und ReaktorSicherheit gGmbH.

LAPACK Linear Algebra PACKage.

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

MPI Message Passing Interface.

MUMPS MUltifrontal Massively Parallel sparse direct Solver.

NUMA Non-Uniform Memory Access.

NUT NUmerical Toolkit.

ODE Ordinary Differential Equation.

OpenMP Open Multi-Processing library.

PDE Partial Differential Equation.

PETSc Portable, Extensible Toolkit for Scientific Computation.

ScaLAPACK Scalable Linear Algebra PACKage.

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

Glossary

HW1 Hardware installed on GRS cluster.

HW2 Hardware installed on a LRZ CoolMUC-2 Linux cluster.

List of Figures

List of Tables

Listings

Contents

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, produce 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 [28]. In spite of considerable initial investments, nuclear power plants have low operating costs and long service life which make them particularly cost effective.

In recent years, nuclear power plants have become an attractive means of power generation because of relatively low emission of carbon dioxide. As a result, a level of green house gase emissions to the atmosphere and thus the contribution of nuclear power plants to the global warming are relatively small [45].

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 [17].

The main problem assosiated with nuclear power is radioactive waste which is extremely dangerous for people and the 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 the 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 an 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 since 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 [17].

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 [23]. 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 [23]. 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
NUT	Container of various numerical tools and algorithms

Table 1.1.: A list of software developed by GRS, [22], where LWR stands for a Light Water Reactor

The main focus of this thesis is dedicated to ATHLET and NUT software packages. The goal of the study is to identify the most compute-intensive parts of the ATHLET-NUT code and possibly accelerate its execution time.

The next chapter continues the introduction and gives a general overview of ATHLET-NUT purpose, design, architecture and coupling. The introduction ends with a clear exposition of the problem statement presented in Chapter 3 where the order of the remaining thesis is described in detail.

2. Overview of ATHLET and NUT software

2.1. ATHLET

Analysis of THermal-hydraulics of LEaks and Transients (ATHLET) software is developed by GRS for an 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 [21]. The code provides specific models and methods to simulate many types of nuclear power plants, comprising current light water reactors (PWR[1], BWR[2], WWER[3], HPCR[4]), advanced Generation III+ and IV reactors as well as SMRs[5] [21].

Physical processes inside of hydraulic circuits of light water reactors can be naturally described by a two-phase thermo-fluiddynamic model based on equations of conservation of mass, momentum and energy for liquid and vapor phases i.e. Equations 2.1 - 2.7, [7].

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
$$\frac{\partial(\alpha\rho_v\vec{w}_v)}{\partial t} + \nabla(\alpha\rho_v\vec{w}_v\vec{w}_v) + \nabla(\alpha p) = \vec{F}_v \tag{2.4}$$

[1]Pressurized Water Reactor
[2]Boiling Water Reactor
[3]Water-Water Energetic Reactor
[4]High Power Channel-type Reactor
[5]Small Modular Reactor

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

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

7. Volume vapor fraction

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

The notation is as follows: p - pressure of a liquid-vapor mixture, ψ - a mass source term, $\vec{F}$ - an external composite force acting on a CV[6], E - an external composite energy source term within a CV, subscripts l and v denote liquid and vapor phases, respectively.

Spacial integration of the conservation equations, System 2.1 - 2.7, is performed on basis of the finite volume method using one-dimensional problem formulation, Figure 2.1. Then, the system is transformed to an initial value problem of a system of non-autonomous Ordinary Differential Equations (ODEs) by means of certain additional mathematical transformations, see [7].

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

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

According to *ATHLET Mod 3.1A – Models and Methods* [7], System 2.8 is stiff and thus must to be solved with an implicit solver. To avoid a high computational cost of a fully implicit method, ATHLET makes use of an extrapolation ansatz based on the linearly implicit Euler method. This may be interpreted as a six stage W-method where the exact Jacobian information is not required. However, fresh information of the Jacobian matrix during a numerical simulation greatly improves stability and robustness of the method. Therefore, several mechanisms have been implemented in ATHLET to closely monitor and, if it is required, to update a Jacobian matrix approximation using the finite difference method.

[6]CV - Control Volume

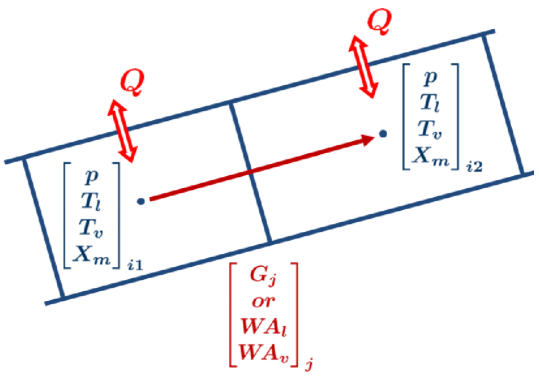


Figure 2.1.: One-dimensional finite volume formulation of a thermo-hydraulic problem in ATHLET, [44], where T_l - a temperature of liquid inside of a CV; T_v - a temperature of vapor inside a CV; X_m - mass quality; G - mass flow of a liquid-vapor mixture; W - a velocity component of a liquid-vapor mixture perpendicular to a CV boundary; A_l - an area occupied by the liquid fraction on a CV boundary, A_l - an area occupied by the vapor fraction on a CV boundary; Q - an external heat transfer

In the general case, a step of the W-method method, implemented in ATHLET, can be viewed as a sequence of six stages in the following way. Each stage uses the implicit Euler method and exactly one Newton's iteration to evaluate values of vector y at the next integration step h with different accuracy. Then, the obtained values are extrapolated, in the order shown in Figure 2.2, to achieve the third order of numerical

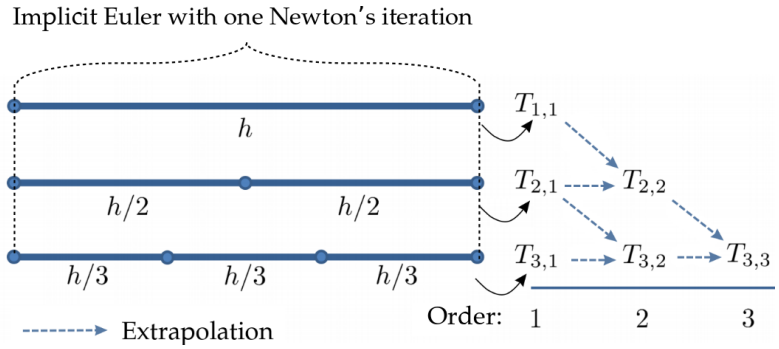


Figure 2.2.: A general view on the 6-stage W-method implemented in ATHLET (based on [44]), where $T_{1,1} = y_0 + y_{1,0}$; $T_{2,1} = y_0 + y_{2,0} + y_{2,1}$; $T_{3,1} = y_0 + y_{3,0} + y_{3,1} + y_{3,2}$

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

$$(I - h_i J)\delta y_{ij} = h_i f(t_0 + j \cdot h_i, y_0 + \sum_{l=0}^{j-1} \delta y_{il}) + h^2 \frac{\partial f_0}{\partial t} \tag{2.9}$$

where $J \approx \frac{\partial f}{\partial y}$ - an approximation of Jacobian matrix; $i = 1, 2, 3$; $j = 0, \ldots, i-1$; $h_i = h/i$.

2.2. NUT

NUmerical Toolkit (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 a paradigm of the *Adapter/Wrapper* pattern which provides a uniform common interface for its services to any application that follows a certain communication protocol. Currently, only ATHLET can communicate with NUT. However, more GRS applications are going to adopt the protocol in the future and will be able to operate with NUT as well. The approach, implemented in NUT, helps to achieve re-usability, flexibility and extensibility properties of the code.

Currently, NUT is based heavily on Portable, Extensible Toolkit for Scientific Computation (PETSc). It is one of the most widely used parallel numerical software libraries [47]. It includes a large suite of parallel linear and nonlinear equation solvers as well as a software-infrastructure to handle computations on distributed-memory machines by means of Message Passing Interface (MPI) and specific data structures. Fortunately, through 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

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 by analogy with

the Transmission Control Protocol (TCP). The plug-in hides all MPI calls to the sever which considerably improves readability of the code.

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

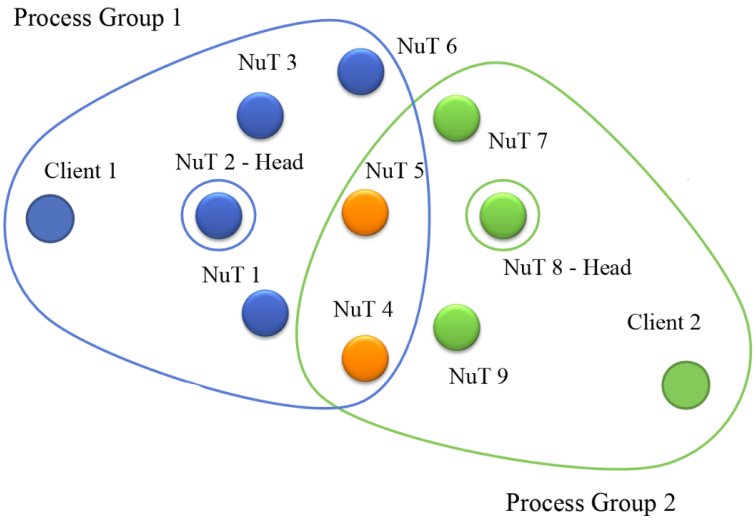


Figure 2.3.: An example of NUT process groups

The design of NUT allows sharing of some NUT-MPI processes among different process groups due to performance reasons i.e. a 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 a 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. According to the current implementation of NUT, the communication between a client and server is synchronous i.e. a client gets blocked while waiting for a result from the server.

A general view on ATHLET-NUT software coupling is given in Figure 2.4 where ATHLET is responsible for marching the numerical time-integration process whereas NUT computes solutions of linear systems derived from Equation 2.9.

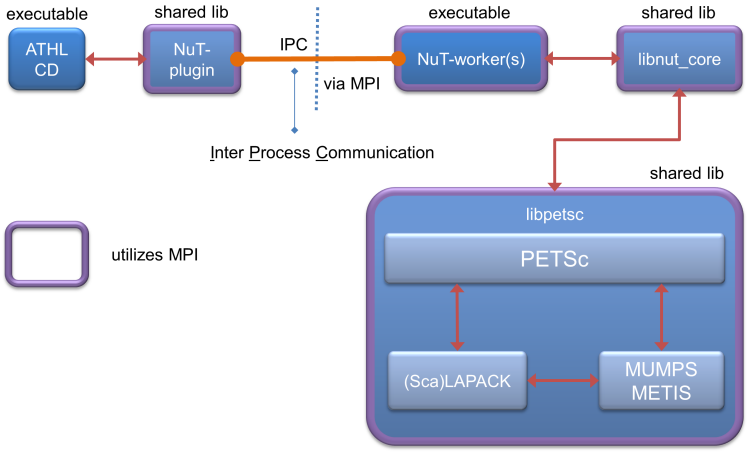


Figure 2.4.: ATHLET-NUT software coupling

Partial and full Jacobian matrix updates derived from finite differences are computed on the client side since only a client has access to the function f of Equation 2.8. Due to transformations of the underlying system of Partial Differential Equations (PDEs) and specifics of finite volume discretization, the Jacobian matrix is sparse and, therefore, ATHLET uses a matrix compression algorithm described in Section 6.1 to reduce an amount of Jacobian column evaluations. Having computed a matrix column, ATHLET immediately broadcasts it to its entire NUT process group by means of the 3-way handshake mechanism 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 the block-row distribution scheme induced by PETSc. In turn, NUT is waiting for the entire Jacobian matrix information from ATHLET and starts solving Systems 2.9 right after receiving the corresponding request from the client.

3. Problem Statement

Integration of a system of ODEs by means of W-methods involves solving several systems of linear equations. 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.

$$A_i \delta y_{ij} = b_{ij} \tag{3.1}$$

where $A_i = (I - h_i J)$ is a $n \times n$ nonsingular sparse matrix; δy_{ij} and b_{ij} 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 both solving sparse linear systems and evaluations of non-linear function f, Equation 2.8. However, because of the time limit of the thesis, the main focus of this study is on solving Systems 3.1 efficiently on GRS computational cluster.

There exist two families of linear sparse solvers, namely: iterative and direct sparse methods. In the general case, execution time of any method, regardless of a solver family, is bounded by $O(n^2)$ complexity due to matrix sparsity, where n is the number of equations in a system. However, constants in front of the factor n^2 can vary significantly between the methods which explains differences in execution time. Additionally, it is important to mention that the families follow different approaches for solving sparse linear systems and are greatly different in details. Therefore, they possess different numerical properties. Among all properties, there are some which are particularly important for efficient execution of W-methods, namely:

- robustness of a method to treat, in particular, ill-conditioned systems
- parallel efficiency

These, above mentioned properties, can be treated as non-functional requirements to

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 kinds 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
- technical support and maintainability of a solver/package

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

This report is organized as follows. Chapter 4 provides information about methodology, data, software and hardware used in this study. Subsections 5.1.1 and 5.1.2 give an overview of the theory and parallelization aspects of iterative and sparse direct methods as well as discussions of some issues related to numerical solution accuracy. Then, in Subsection 5.1.3, we make a conclusion about which type of sparse linear solvers is well suited for numerical time integration governed by W-methods. In Section 5.2, a concrete implementation of a specific method is selected by means of testing. From Section 5.4 onwards, we perform configuration and adaptation of a solver for distributed-memory computations. At the end, Section 5.6 summarizes obtained results and makes a general conclusion with respect to data and compute environment provided by GRS.

An additional topic, considered in this study, is an improvement of ATHLET-NUT communication during Jacobian matrix transfers. As it was described in Section 2.3, ATHLET, the client, transfers a Jacobian matrix in a column-wise fashion. NUT, the server, treats each column transfer as a service and, therefore, each transfer passes through the 3-way handshake described in Section 2.3. Moreover, it is important to mention one more time, due to the current implementation of ATHLET-NUT coupling, 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 6.1, is to minimize the number of perturbations of non-linear function f of Equation 2.8. Additionally it allows to reduce an amount of column transfers as well. Therefore, it improves the

overall application performance from both computational and communication points of view. However, there are still some aspects to be considered.

Due to specifics of a matrix compression algorithm, described in Section 6.1, column lengths are decreasing between the first and last columns of a compressed Jacobian matrix form which, as a result, leads to unequal MPI message sizes.

In the last 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 which improves utilization of network bandwidth. Secondly, it helps to reduce an amount of synchronizations between a client and the 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 transfers with computations.

In Section 6.1, we briefly describe the Jacobian matrix compression algorithm and the resulting ATHLET-NUT communication problem. In Section 6.2, we present and describe an algorithm which is supposed to resolve the problem. Section 6.3 provides a description of developed benchmarks and test data. Then, we discuss obtained results in Section 6.4. Finally, in Section 6.5, we provide a general conclusion of the performed part of the study and summarize the results.

4. Methodology and Experimental Setup

ATHLET is a CFD[1] tool designed for computer-based simulations of transient thermo-hydraulic problems where topology of hydraulic circuits can be changed during a numerical simulation. As a direct consequence, the Jacobian matrix can frequently change with respect to both numerical values and the matrix sparsity structure between time integration steps. Since ATHLET usually generates hundreds of matrices during a simulation, configuration of a linear solver in run-time becomes a time consuming and compute-expensive problem. Moreover, results of such dynamic solver configuration may be difficult to analyze and interpret.

In this study, a static solver configuration approach is used, instead. In other words, a solver is configured with only a small set of matrices, i.e. GRS matrix set, randomly saved during simulations of the most common GRS test scenarios. In the general case, it may lead to inaccurate conclusions, however, it is only one technically feasible approach.

Besides GRS matrix set, a second set was used for verification of testing results. The set, called SuiteSparse matrix set, was generated by donwloading a dozen of matrices from SuiteSparse Matrix Collection [11], [12] where we tried to pick out different matrices with respect to both the number of equations n and matrix density i.e. ratio between the number of non-zero elements nnz and the number of equations in a system.

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

Approximations of condition numbers, shown in Tables 4.1 and 4.2, were computed using the Rayleigh–Ritz procedure. The reader can become familiar with the procedure in [30]. GMRES solver, configured with 1000 iteration steps before the restart, was applied to 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 bounds since the algorithm overestimates the smallest eigenvalues.

[1] Computational Fluid Dynamics

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

Table 4.1.: GRS matrix set, where SYMM - symmetric; NON-SYMM - non-symmetric; SYMM-PTRN- non-symmetric but with symmetric sparsity pattern

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

Table 4.2.: SuiteSparse matrix set, where SYMM - symmetric; NON-SYMM - non-symmetric; SYMM-PTRN- non-symmetric but with symmetric sparsity pattern

Two different hardware were available for this study. The first machine was a compute-cluster installed in GRS (HW1) which was the main target. Additionally, LRZ CoolMUC-2 Linux cluster (HW2) was used every time when some ambiguous results were obtained in order to check whether a problem was hardware, software or algorithmic specific. Table 4.3 shows compute-node specifications of both compute-clusters.

For this study, OpenMPI implementation of the MPI standard was used because of its open-source license and comprehensive documentation. The library has many

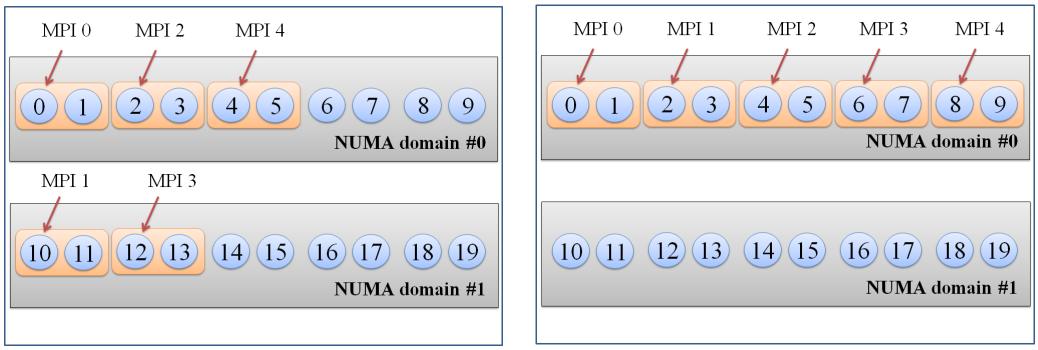
	HW1 (GRS)	HW2 (LRZ Linux)
Architecture	x86_64	x86_64
CPU(s)	20	28
On-line CPU(s) list	0-19	0-27
Thread(s) per core	1	1
Core(s) per socket	10	14
Socket(s)	2	2
NUMA node(s)	2	4
Model	62	63
Model name	E5-2680 v2	E5-2697 v3
Stepping	4	2
CPU MHz	1200.0	2036.707
Virtualization	VT-x	VT-x
L1d cache	32K	32K
L1i 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
RAM per node, GB	128	64

Table 4.3.: Hardware specifications

options for processes pinning which was intensively used during the study.

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

A rank-file specifies explicit mapping between MPI processes, ranks, and actual processing elements, cores, of a compute-cluster. The script has two modes, namely: *spread* 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 mapping 5 MPI ranks and 2 OpenMP threads per rank onto a compute



(a) *Spread* mode (b) *Close* mode

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

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. Configuration of a sparse linear solver

5.1. Overview of Sparse Linear Solver Types

Next two subsections on iterative and direct sparse methods, respectively, are organized as follows. The first part of each subsection contains a concise theory overview of a linear solver type. The second parts provide discussions of the main sources and means of parallelization of both iterative and direct sparse methods. The last part of each subsection focuses on issues related to numerical solution accuracy of a particular method type.

Then, the section continuous with a discussion and conclusion of which type of linear solvers is well suited for solving systems derived from ATHLET thermo-hydraulic simulations.

5.1.1. Iterative Methods

5.1.1.1. Theory Overview

Given an initial guess, an iterative method generates a sequence of approximate solutions by means of a specific rule. Depending on the method and the given problem, there may exist certain conditions such that the aforementioned sequence eventually converges to the exact solution. Iterative methods are preferred for their relatively low computational cost per iteration and storage requirements $O(nnz)$. In essence, the methods make use of simple linear algebra kernels at each iteration and thus can handle matrix sparsity efficiently.

The family of iterative methods consists of two distinct classes, namely: stationary and Krylov methods. Nowadays, Krylov methods dominate in the field of scientific computing because of their rather fast convergence in case of solving well conditioned systems or/and a "good" initial guess.

The most well-known methods among the Krylov family are Conjugate Gradient (CG) for symmetric positive definite matrices, MINimal RESidual method (MINRES)

for symmetric indefinite systems and General Minimum RESidual method (GMRES) for non-symmetric systems of linear equations. There also exist different variants of CG such as BiConjugate Gradient Method (BiCG), BiConjugate Gradient STABilized Method (BiCGSTAB), etc.

The key idea is a construction of an approximate solution of a system of linear equations as a linear combination of vectors b, Ab, A^2b, $A^3b, \dots A^{n-1}b$ where, without of lost of generality, the initial guess x_0 is equal to zero. The combination defines a subspace, also known as Krylov subspace $\mathcal{K}_n$. At each iteration, the subspace is expanded by adding and evaluating the next vector in the sequence. The methods usually define and expand another subspace $\mathcal{L}_n$ of the same size as $\mathcal{K}_n$ such that $r_n = b - Ax_n \perp \mathcal{L}_n$ which is known as the Petrov-Galerkin condition. A construction of subspace $\mathcal{L}_n$ is defined by the methods and based on matrix properties.

Some Krylov methods also have interpretations as minimization problems. For example, GMRES aims to minimize the Euclidean norm of the residual r_m of a solution vector x_m in the mth Krylov subspace $\mathcal{K}_m$. However, the basis vectors (b, Ab, A^2b, $A^3b, \dots A^{m-1}b$) of the mth Krylov subspace are usually close to linearly dependent and, therefore, a solution vector x_m is constructed in an orthonormal basis U_m which forms the same subspace $\mathcal{K}_m$.

$$r_m = \min_{x_m \in \mathcal{K}_m} ||Ax_m - b||^2 = \min_{y_m \in \mathbb{R}^m} ||AU_m y_m - b||^2 \tag{5.1}$$

where a vector x_m can be written in the basis U_m as:

$$x_m = U_m y \tag{5.2}$$

The orthogonalization of the basis can be performed in different ways. Saad and Schultz, in [42], proposed to use the Arnoldi algorithm, see [5] for details, for constructing an l_2-orthogonal basis. As a result, Equation 5.1 can be written as follows:

$$r_m = \min_{y_m \in \mathbb{R}^m} ||U_{m+1} H_{m+1,m} y_m - ||b|| u_1||^2 = \min_{y_m \in \mathbb{R}^m} ||H_{m+1,m} y_m - ||b|| e_1||^2 \tag{5.3}$$

where $H_{m+1,m}$ is a $(m+1) \times m$ Hessenberg matrix with non-zero entries defined by the Arnoldi algorithm.

An application of the Givens rotation algorithm results in computing the corresponding QR decomposition of matrix $H_{m+1,m}$. After substitution of the matrix with the corresponding QR decomposition and some mathematical transformations, Equation 5.3 can be written as follows:

$$r_m = \min_{y_m \in \mathbb{R}^m} ||Q^T R y_m - ||b|| e_1||^2 = \min_{y_m \in \mathbb{R}^m} || \begin{pmatrix} R_m \\ 0 \end{pmatrix} y_m - \begin{pmatrix} \tilde{b_m} \\ \tilde{b_{n-m}} \end{pmatrix} ||^2 \tag{5.4}$$

Now, the minimization problem 5.4 can be solved as:

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

Given the decomposition of a vector in the orthonormal basis U_m, Equation 5.2, a solution vector x_m can be written as:

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

A solution significantly improves with growth of subspace $\mathcal{K}_m$. This, as a direct consequence, leads to a considerable increase of a computational cost and storage space. Therefore, the algorithm usually runs till 20 - 50 column vector evaluations of the corresponding Krylov subspace and restarts using a computed approximate solution as an initial guess for the next iteration.

5.1.1.2. Parallelization Aspects

In the general case, iterative methods usually make use of dot and matrix-vector products for solving systems of linear equations. Applications of these linear algebra kernels allow to efficiently handle sparsity of linear systems and thus reduce computational complexity of the methods. Additionally, it allows to distribute vectors and matrices across multiple compute-units and solve systems of equations in parallel, efficiently exploiting data-based parallelism. Hence, the main drop of parallel performance mainly comes from process-communication overheads.

However, some methods can have sequential parts that may affect on parallel performance as well. For instance, a triangular solve operation, Equation 5.5, of the GMRES method is usually computed in a single processor because of its small size which depends on the number of iterations before the restart. Hence, if the underlying system

of equations is relatively small then such sequential operations can become a bottleneck in solving the corresponding system.

5.1.1.3. Preconditioners

The most important criterion of Krylov methods is convergence. In case of a "bad" initial guess x_0, the convergence of iterative methods strongly depends on an involved matrix and, in particular, on its condition number. For instance, Equation 5.7 shows dependence of an error reduction in the solution on the corresponding matrix condition number in case of the CG method. It can be clearly observed that a big condition number may results in a very slow error reduction and, therefore, in a slow convergence rate.

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

where $k = \frac{\lambda_{max}}{\lambda_{min}}$ - a matrix condition number

In practice, a linear transformation, known as preconditioning, is applied to the original system of equations in order to reduce its condition number. As a result, the solution process of the modified system can be accelerated. The transformation can be applied in different ways. For instance, Equations 5.8 and 5.9 show applications of a preconditioning matrix P from left and right sides, respectively.

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

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

In the general case, a good preconditioning algorithm should result in *a low computational cost, low storage space and a low condition number of the transformed system*. Additionally, computations of large linear systems require an algorithm to be adapted for parallel executions as well.

There exist numerous techniques to compute a preconditioner P for 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. Experience has shown that some techniques can work particularly well for matrices derived from a certain

Package name	Origin	Method	Tuning parameters	Comments
block Jacobi	PETSc	block Jacobi	-pc_bjacobi_blocks -sub_pc_type	-
additive Schwarz	PETSc	additive Schwarz	-pc_asm_blocks -pc_asm_overlap -pc_asm_type -pc_asm_local_type -sub_pc_type	-
euclid	hypre	ILU(k)	-nlevel -thresh -filter	deprecated form PETSc
pilut	hypre	ILU(t)	-pc_hypre_pilut_tol -pc_hypre_pilut_maxiter -pc_hypre_pilut_factorrowsize	-
parasail	hypre	SPAI	-pc_hypre_parasails_nlevels -pc_hypre_parasails_thresh -pc_hypre_parasails_filter	-
SPAI	Grote, Barnard	SPAI	-pc_spai_epsilon -pc_spai_nbstep -pc_spai_max -pc_spai_max_new -pc_spai_block_size -pc_spai_cache_size	-
BoomerAMG	hypre	algebraic multigrid	-pc_hypre_boomeramg_cycle_type -pc_hypre_boomeramg_max_levels -pc_hypre_boomeramg_max_iter -pc_hypre_boomeramg_tol etc.	39 tuning parameters in total

Table 5.1.: A list of parallel preconditioning algorithms available in PETSc

PDE or a system of PDEs e.g. Poisson, NavierStokes, etc., and discretized in a certain way. However, *sometimes it can take a quite considerable amount of time to tune a particular preconditioning algorithm in order fulfill all above listed requirements*.

As it can be clearly observed from Table 4.1, all matrices contained in GRS matrix set are very ill-conditioned and, as a result, require suitable linear transformations. PETSc provides various preconditioning methods as well as access to some external preconditioning libraries. Table 5.1 contains a list of some widely-used preconditioning algorithms available in PETSc, capable to run in parallel on distributed-memory machines, as well as their short descriptions and tuning parameters.

Detailed descriptions of all tuning parameters listed in Table 5.1 can be found in either PETSc or Hypre users' manuals, [9] and [18], respectively.

It is worth mentioning that both block Jacobi and additive Schwarz algorithms split the original system into smaller blocks where each block is usually computed on a single processor. Therefore, these algorithms require an explicit specification of another preconditioning algorithm or a linear solver, called as sub-preconditioner, for local block computations. As an example, code Listing 5.1 shows an example of usage of the *sequential* PETSc built-in *LU* matrix decomposition subroutine as a sub-preconditioner for the block Jacobi algorithm. As a result, the number of tuning parameters for these two algorithms can grow significantly.

```
-pc_bjacobi_blocks 4
-sub_pc_type lu
-pc_factor_mat_ordering_type rcm
-pc_factor_pivot_in_blocks true
```

Listing 5.1: An example of usage of the PETSc built-in sparse direct linear solver as a sub-preconditioner for the Block Jacobi preconditioning algorithm

5.1.2. Direct Sparse Methods

5.1.2.1. Theory Overview

Direct sparse methods combine the main advantages of direct and iterative methods. In other words, numerical accuracy of the methods is comparable with the standard Gaussian Elimination process while their computational complexity is typically bounded by $O(n^2)$ [48] due to efficient treatment of non-zero matrix elements. As it is in case of direct dense methods, a solution of a system of equations is computed by means of forward and backward substitutions using *LU* decomposition of the corresponding matrix.

The multifrontal method is probably the most representative example of direct sparse solvers, inroduced by Duff and Reid in [16]. The method is, in fact, an improved version of the frontal method [27] and can compute independent fronts in parallel. A front, also called a frontal matrix, can be considered as a small dense matrix resulting from a column elimination of the original system. There also exist left- and right-looking vatiants of the multifrontal method explained in detail in [41].

In this subsection, the theory of multifrontal method is explained, which helps to understand parallel aspects and strong scaling behavior of direct sparse solvers in case of parallel execution. To keep the overview rather simple, we assume that matrix A is symmetric positive definite and sparse. Therefore, matrix decomposition can be conveniently written as follows:

$$A = LDL^T \qquad \text{with} \quad (D)_{ii} > 0 \tag{5.10}$$

The algorithm starts with symbolic factorization of System 5.10 with the aim of predicting a sparsity pattern of factor L. Once it is done the corresponding elimination tree can be constructed.

Figure 5.1 shows an illustrative example of a sparse matrix A and its Cholesky factor L, taking from [36]. The solid circles represent the original non-zero elements whereas hollow ones define fill-in elements of L.

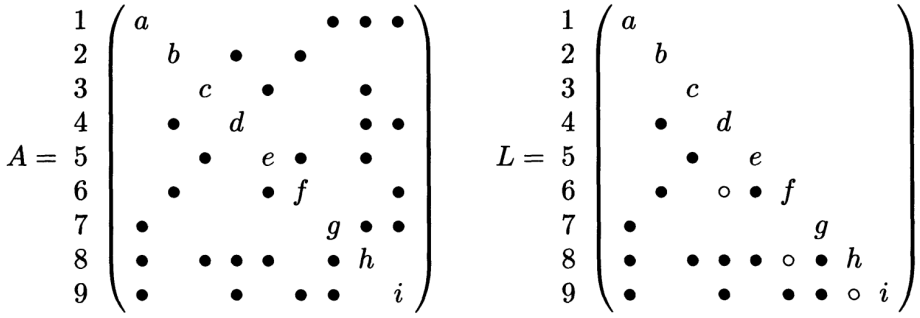


Figure 5.1.: An example of a sparse matrix and its Cholesky factor, [36]

An elimination tree is a crucial part of the method. It can be considered as a structure of n nodes where node p is the parent of j if and only if it satisfies Equation 5.11. It is worth pointing out that Definition 5.11 is not only one possible and one can define a strucutre of an elimination tree in a different way as well, [36].

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

In fact, node p represents elimination of the corresponding column p of matrix A as well as all dependencies of column p factorization on results of eliminations of its descendants.

Given Definition 5.11 and a sparsity pattern of factor L, the corresponding elimination tree can be constructed, as it is shown in Figure 5.2.

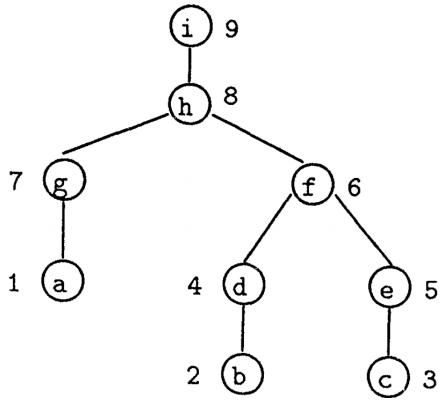


Figure 5.2.: An elimination tree of matrix A of the example depicted in Figure 5.1, [36]

The fundamental idea of the multifrontal method spins around frontal and update matrices. Frontal matrix F_j is used to perform Gaussian Elimination for a specific column j and it is equal to a sum of frame Fr_j and update $\hat{U}_j$ matrices, as it can be observed from Equation 5.12

.

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

where i_0, i_1, i_2, ..., i_r are row subscripts of non-zeros in L_{*j} where $i_0 = j$; r is the number of off-diagonal non-zero elements.

Frame matrix Fr_j is filled with zeros except the first row and column which contain non-zero elements of the jth row and column of the original matrix A. Because of symmetry of matrix A, the frame matrix is square and symmetric.

In order to describe parts of an elimination tree, notation $T[j]$ is introduced to represent all descendants of node j in the tree and node j itself. In this case, update matrix $\hat{U}_j$ can be defined as follows:

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

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. Hence, only those elements of descendant columns are included and considered which correspond to a non-zero pattern of the jth column.

Let's consider partial factorization of a 2-by-2 block dense matrix, Equation 5.15, to better understand the essence of update matrix $\hat{U_j}$. Let's assume that matrix B has already been factorized and can be expressed as follows:

$$B = L_B L_B^T \tag{5.14}$$

$$A = \begin{bmatrix} B & V^T \\ V & C \end{bmatrix} = \begin{bmatrix} L_B & 0 \\ V L_B^{-T} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & C - VB^{-1}V^T \end{bmatrix} \begin{bmatrix} L_B^T & L_B^{-1}V^T \\ 0 & I \end{bmatrix} \tag{5.15}$$

The Schur complement, from Equation 5.15, can be viewed as a sum of the original sub-matrix C and update $-VB^{-1}V^T$. The update can be written as a sum of outer products as follows:

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

Firstly, it can be clearly observed that Equations 5.16 and 5.13 are similar. However, Equation 5.13 exploits sparsity of the corresponding rows and columns of factor L and, therefore, masks unnecessary information. Secondly, frame matrix Fr_j corresponds to block matrix C and brings information from the original matrix A, whereas update matrix $\hat{U_j}$ adds information about the columns that have already been factorized.

As soon as frontal matrix F_j is assembled, i.e. the complete update of column j has been computed, elimination of the first column of matrix F_j can be started which will result in computing of non-zero entries of factor column L_{*j}. The process is denoted as

partial factorization of matrix F_j.

Let's denote $\hat{F}_j$ as a result of the first column elimination of frontal matrix F_j. Then, the elimination process of column j can be expressed as follows:

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

where sub-matrix U_j represents the full update from all descendants of node j and node j itself. Equation 5.18 expresses sub-matrix U_j as a sum of outer products:

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

Update column matrix U_j, also called as a contribution matrix, together with the frontal F_j and update $\hat{U}_j$ matrices, forms the key concepts of the multifrontal method. Let's consider an example, depicted in Figure 5.3, to demonstrate the importance of contribution matrices.

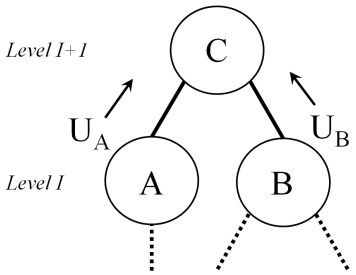


Figure 5.3.: Information flow of the multifrontal method

Let's assume that columns A and B have already been factorized and the corresponding contribution matrices U_A and U_B have already been computed. According to Equation 5.18, it is known that both U_A and U_B matrices contain the full updates of all their descendants including updates of columns A and B as well. Therefore, update column matrices U_A and U_B have already contained all necessary information to construct update matrix $\hat{U}_C$. A detailed proof and careful explanation can be found in [36].

It can happen that only a subset of rows and columns of matrices U_A and U_B is needed due to sparsity of column C. Hence, only relevant elements of the corresponding matrices have to be retrieved to *form* matrix $\hat{U_C}$. For that reason, an additional matrix operation, called ***extend-add***, has been introduced in the theory of direct sparse methods.

As an example, taking from [36], let's consider the *extend-add* operation applied to 2-by-2 matrices R and S which correspond to indices $5,8$ and $5,9$ of a 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 such operation is a 3-by-3 matrix K which can be written as follows:

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

Hence, the formation of frontal matrix F_j can be expressed using the *extend-add* operation and all direct children of node j as follows:

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

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

At this point, it is worth mentioning that the resulting frontal matrix F_j forms a small dense block which has to be factorized along the first column. Partial factorization of the block can be efficiently performed by means of the corresponding dense linear algebra kernels.

After partial factorization of matrix F_j, assembly of contribution matrix U_j must be completed by adding those elements 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. Then, the process continues moving up along the tree. Therefore, complete update matrices are growing in size while the

global elimination process is moving towards the root of the tree.

Manipulations with frontal and contribution matrices play a significant role in performance of the multifrontal method. Sometimes contribution matrices, generated in previous steps, must be stored into a temporary buffer and efficiently retrieved from it later during the global factorization process. This can require to change a column elimination order which can be achieved by some matrix reordering techniques. For instance, *post-ordering*, mentioned by Liu in [36], can be considered as an example of such reordering, in case of symmetric matrices, and can eventually make efficient use of *stack* data structure. Post-ordering is based on topological ordering and thus it is equivalent to the original matrix order. Hence, such reordering results in the same fill-in of the factor [36].

A post-ordered tree implies that each node is ordered before its parent and nodes in each subtree are numbered consecutively. Figure 5.4 shows an example of post-ordering applied to the elimination tree of the matrix shown in Figure 5.1. As a result, consecutive *push* and *pop* operations can be efficiently used during matrix factorization and thus can result in significant simplification of a computer program, see Figure 5.5.

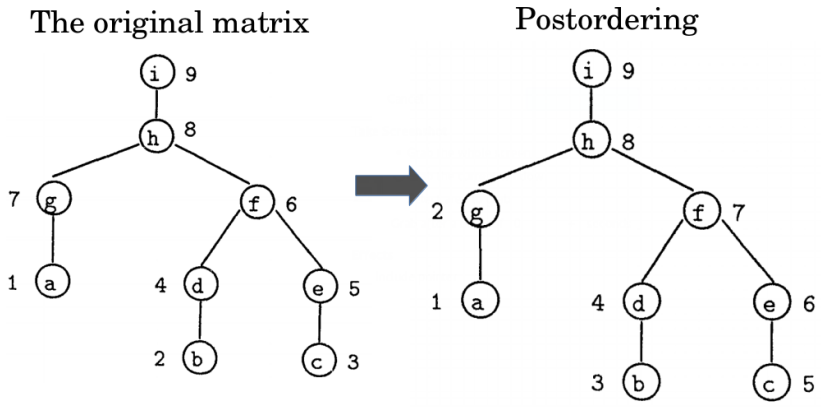


Figure 5.4.: An example of matrix postordering, [36]

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

A super-node is formed by a set of contiguous columns which have the same off-diagonal sparsity structure. Hence, a super-node has two important properties. Firstly,

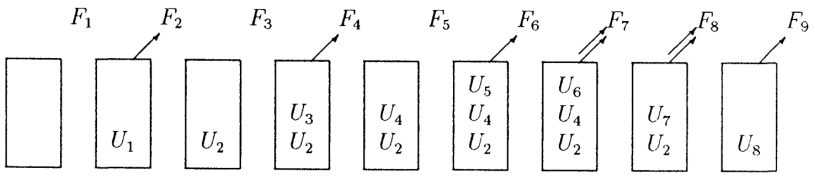


Figure 5.5.: An example of an efficient data treatment during matrix factorization using stacking, [36]

it can be expressed as a set of consecutive column indices, namely: $\{j,\ j+1,\ \ldots,\ j+t\}$ where node $j+k$ is a parent of $j+k-1$ in the corresponding elimination tree. Secondly, the size of super-nodal frontal matrix $\mathcal{F}_j$ is equal to the size of frontal matrix F_j resulted from the original post-ordered tree. As an example, Figure 5.6 shows a post-ordered matrix A, its Cholesky factor L and the resulting super-nodal elimination tree.

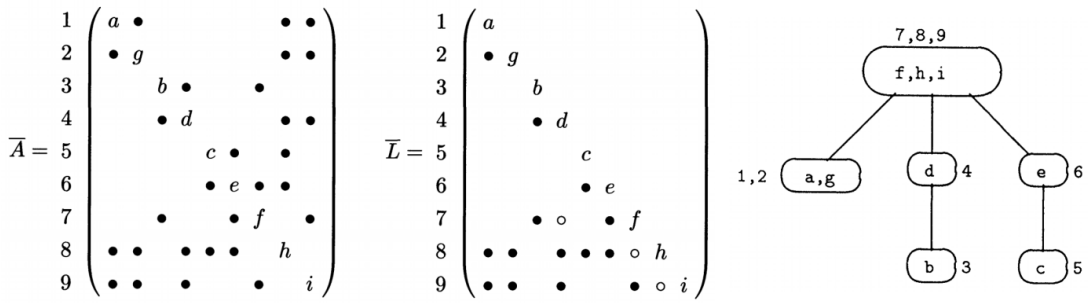


Figure 5.6.: An example of a supernodal elimination tree, [36]

Equation 5.23 shows an assembly process of super-nodal frontal matrix $\mathcal{F}_j$. In contrast to Equation 5.21, frame matrix $\mathcal{F}r_j$ contains more dense rows and columns. As before, the *extend-add* operation is used to construct the full update block from contribution matrices of the children, namely: $U_{c_1},\ U_{c_2},\ \ldots,\ U_{c_s}$.

$$\mathcal{F}_j = \mathcal{F}r_j \ \boldsymbol{\dotplus}\ U_{c_1} \ \boldsymbol{\dotplus}\ \ldots \ \boldsymbol{\dotplus}\ U_{c_s} \tag{5.22}$$

$$\mathcal{F}_j = \begin{bmatrix} a_{j,j} & a_{j,j+1} & \dots & a_{j,j+t} & a_{j,i_1} & \dots & a_{j,i_r} \\ a_{j+1,j} & a_{j+1,j+1} & \dots & a_{j+1,j+t} & a_{j+1,i_1} & \dots & a_{j+1,i_r} \\ \vdots & \vdots & \dots & \vdots & & & \\ a_{j+t,j} & a_{j+t,j+1} & \dots & a_{j+t,j+t} & a_{j+t,i_1} & \dots & a_{j+t,i_r} \\ a_{i_1,j} & a_{i_1,j+1} & \dots & a_{i_1,j+t} & & & \\ \vdots & \vdots & \dots & \vdots & & 0 & \\ a_{i_r,j} & a_{i_r,j+1} & \dots & a_{i_r,j+t} & & & \end{bmatrix} \mathbin{\updownarrow\!\!\!\!\leftrightarrow} U_{c_1} \mathbin{\updownarrow\!\!\!\!\leftrightarrow} \dots \mathbin{\updownarrow\!\!\!\!\leftrightarrow} U_{c_s} \quad (5.23)$$

It is worth mentioning there exist other definitions of super-nodes which allow to amalgamate even more nodes from the original post-ordered tree. For example, Liu pointed out, in [36], that a super-node could be defined without the column contiguity constrain which can result in denser frame matrix $\mathcal{F}r_j$.

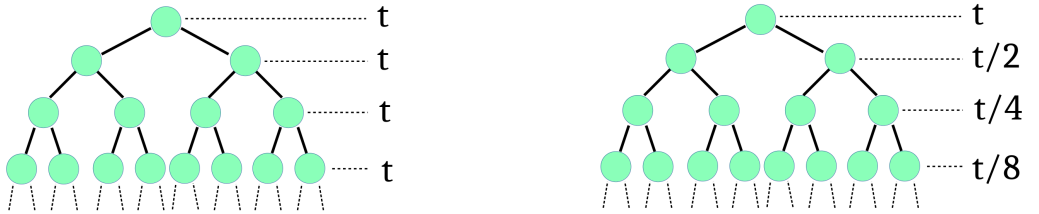
It can be clearly observed that the method consist of three distinct phases, namely: analysis, numerical factorization and solution phases. The analysis phase includes fill reducing matrix reordering, symbolic factorization, post-ordering, amalgamation of nodes, elimination tree construction, etc. During the numerical factorization phase, L and D, or U, factors of the original matrix A are computed based on sequence of partial factorizations of frontal matrices. Given a matrix decomposition, the solution step computes a solution vector x by means of backward and forward substitutions.

5.1.2.2. Parallelization Aspects

In contrast to iterative methods, parallelization of direct sparse methods mainly comes from task-based parallelism where an elimination tree can be considered as a collection of tasks. In fact, the tree represents data dependencies during column partial factorizations and, therefore, reveals dependent and independent tasks. For example, leaves usually locate in separate branches of an elimination tree and thus represent concurrent tasks that can be executed in parallel. On the other hand, parent nodes represent data dependences from their children and cannot be factorized beforehand. Therefore, sparse direct methods have only limited parallelism which swiftly decreases while computations are moving towards the top of an elimination tree.

Let's consider two simple models, that have been developed for this part of the study, in order to demonstrate potential parallel performance of tree-task parallelism. The models, in fact, are perfectly balanced binary trees with different costs per level. Within

a level, the cost is distributed equally among the nodes. The first model, Figure 5.7a, implies quadratic decrease of a computational cost between nodes of adjacent levels whereas the second one, Figure 5.7b, simulates cubic decay of compute-intensity. The models intend to reflect growth of complete update matrices in size, while moving from bottom to top along an elimination tree, and thus increase of floating point operations.



(a) Model 1: equal cost per level

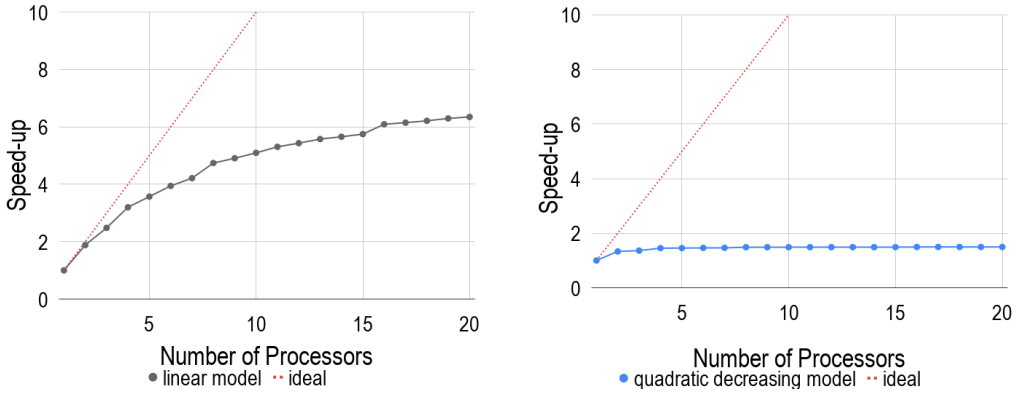
(b) Model 2: quadratically decreasing cost per level

Figure 5.7.: Examples of tree-task parallelism

The models imply parallel computations within a level but sequential execution between them. In other words, to start computations at the next top level, factorizations of all nodes at the current one have to be fully performed. It also means that computations at the next level cannot be started even if there are some available free processors but factorization of the last node at the current level has not been completed yet. Thereby, minimal execution time of both models can be exactly evaluated based on the model descriptions. Essentially, it is equal to a sum of time spent on a single node of each level i.e a sum along the deepest branch. Therefore, it determines asymptotes in the corresponding speed-up graphs.

Figures 5.8a and 5.8b represent strong scaling behavior of both models filled with 65535 nodes i.e. 16 levels. As it can be observed, the models demonstrate a rapid drop of parallel performance, especially in case of the quadratic one, Figure 5.8b . Table 5.2 compares speed-up of two models obtained with 32768 and 20 abstract processors. Number 32768 is equal to the number of leaves at the bottom level and thus implicitly determines the maximum speed-up. It is worth mentioning that two models almost exhaust tree-task parallelism even with 20 processing elements. Further increase of the number of processor can only barely improve the overall parallel performance.

These, rather simple, models reveal the most important fact about tree-task parallelism of sparse direct methods. The performance depends heavily on an elimination tree structure and, in particular, on a distribution of compute-intensity among nodes.



(a) Model 1 (b) Model 2

Figure 5.8.: Theoretical speed-up of models 1 and 2

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

Table 5.2.: Theoretical speed-up of models 1 and 2

As it was mentioned above, the intensity is usually centered on the top part of the tree where task-based parallelism is limited due to data dependencies. As an example, Liu observed that factorizations of the last 6 nodes took slightly more than 25% of the total number of floating point operations in case of an application of the multifrontal method to a $k - by - k$ regular model problem using a nine-point difference operator, [36].

Node-data parallelism is often combined with tree-task one which usually results in an improvement of parallel performance of direct sparse methods. In the general case, frontal matrix $\mathcal{F}_j$ can be distributed across multiple processors and partially factorized in parallel. However, performance of node-data parallelism depends on both a matrix size and the number of processors assigned to perform factorization. Over-subscription of processing elements to a node can result in slow-down induced by communication overheads. Therefore, data parallelism is only applied to top and middle parts of elimination trees because of fine granularity of bottom levels.

By and large, combined tree-task and node-data parallelism improves performance and strong scaling of sparse direct solvers, however, it cannot change the performance trend induced by tree-task parallelism. Therefore, one can still expect stagnation of

speed-up even with a relatively small number of processors.

A parallel implementation of sparse direct solvers demands to expand the analysis phase by adding two more pre-processing steps, namely: process mapping and load balancing. Both mapping and balancing are usually performed statically during the analysis of an elimination tree.

5.1.2.3. Threshold Pivoting and Solution Refinement

Because of an accumulative effect of inexact computer arithmetics due to a floating point representation of real numbers and, as a result, truncations and rounding errors, small numerical values along the main matrix diagonal during the Gaussian Elimination process can result in significant numerical inaccuracy of the process. Therefore, pivoting is a crucial step of Gaussian Elimination. It implies interchanging rows and columns of a matrix in such a way to place distinct and distant values form zero to the main diagonal.

In case of direct dense methods, pivoting is a straightforward operation and can be expressed as multiplication of the original matrix A by a permutation matrix P, where each row and column contain a single 1, at the corresponding place, and 0s everywhere. However, treatment of pivoting in direct sparse methods is an issue.

On the one hand, absence of numerical information during the analysis phase makes it impossible to perform pivoting at this step. On the other hand, an application of pivoting during the numerical factorization phase usually distorts all matrix reorderings and, therefore, the elimination tree structure. As a consequence, pivoting can lead to significant fill-in, load unbalance and, as a result, slow-down of numerical factorization. For that reason, threshold pivoting is commonly used, in practice, for direct sparse methods.

Threshold pivoting means that a pivot $|a_{i,i}|$ is accepted if it satisfies Equation 5.24.

$$|a_{i,i}| \geq \alpha \times max_{k=i\ldots n}|a_{k,i}| \tag{5.24}$$

where $\alpha \in [0, 1]$ and $k = i \ldots n$ represents row indices of column i within the fully summed block of a frontal matrix.

Factorization of a column is suspended i.e. delayed, if Equation 5.24 cannot be satisfied within the fully-summed block of a frontal matrix. In this case, the column and the corresponding row are moved to the parent's frontal matrix as a part of its contribution block where the process repeats again. The process is also known as

delayed pivoting and helps to improve numerical accuracy. Higher values of α lead to more accurate solutions but often generate extra fill-in and lead to load unbalance which, as a result, affect parallel performance. On the other hand, smaller values usually preserve the original elimination tree and, therefore, preserve the load balance computed during the analysis phase by appropriate mapping processors across nodes of the tree. However, in this case, numerical accuracy usually degrades. In practice, values of α lay in the range between 0.01 and 0.1 [37].

A case when parameter α is equal to **0** is known as static pivoting which means that no pivoting is being performed during the numerical factorization phase. This allows to better optimize data layout, load balancing, and communication scheduling, [35], before numerical factorization which is supposed to result in better parallel performance.

By and large, solutions computed by direct sparse methods can be numerically inaccurate, in some degree, and may demand to perform solution refinements. As an example, solution accuracy can be improved using the iterative refinement method. Code Listing 5.3 shows a pseducode of the method where parameter ω represents an estimation of the backward error, Equation 5.25, [6]. In practice, the method usually takes 2 or 4 iterations to achieve sufficient numerical accuracy.

```
# 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 > ω
  # find correction
  d = Solve(factorization=LU, rhs=r)

  # update solution
  x = x - d

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

Listing 5.2: Pseudocode of the iterative refinement method

$$\frac{|b - A\hat{x}|_i}{(|b| + |A||\hat{x}|)_i} \tag{5.25}$$

where $\hat{x}$ is a computed solution; $|\cdot|$ is the element-wise module operation.

As an alternative to the iterative refinement method, one can use the resulting LU decomposition of matrix A as a preconditioner for an iterative solver, for instance GMRES. Based on experience of ATHLET-NUT users, this approach usually takes 1 up to 3 iterations to achieve desired numerical accuracy even with extreme small values of α.

Finally, it is worth mentioning that both refinement techniques, mentioned above, exploit only data-based parallelism and, therefore, are scaled well on distributed-memory machines.

5.1.3. Results and Conclusion

Nowadays, iterative methods is a common choice for solving sparse systems of linear equations because of their possible fast convergence and high parallel efficiency. However, applications of such methods always demand preconditioning for ill-conditioned systems to make methods converge to numerical accurate solutions. It can be clearly observed from Table 4.1 that numerical integration of thermo-hydraulic simulations in ATHLET entails solving ill-conditioned systems based on estimated condition numbers of matrices form GRS matrix set.

As the first step of the study, we tested various preconditioning algorithms together with their tuning parameters, mentioned in Table 5.1, applied to GRS matrix set. GMRES was chosen as an iterative solver with values of relative and absolute convergence tolerances in the residual norm to be equal to $1E-8$ and $1E-4$, respectively. A coarse grid search was used with maximum 3 values for each tuning parameter starting from the default towards more accurate values in order to refine parameter settings of each preconditioning algorithm. Testing results showed that none of them could lead to convergence for the entire set of matrices.

One can assume that a finer grid search can result in finding a suitable preconditioning algorithm with parameter settings that can lead to convergence of GMRES solver for the entire set. However, it is important to point out that the matrices were generated by running the most common GRS thermo-hydraulic test-scenarios and saving them somewhere during the time integration process. Hence, there is no guarantee that the parameters found in such a way can always lead to convergence of GMRES solver in all time steps of any thermo-hydraulic simulation. Therefore, iterative methods may

not satisfy *robustness* criterion stated in Chapter 3 as a non-functional requirement to a sparse linear solver.

Taking into account the above reasoning, we have come to the conclusion that sparse direct methods is the best choice for our problem, in spite of the limited tree-task parallelism described in Subsection 5.1.2.2, because the methods stably result in numerical accurate solutions even in case of ill-conditioned linear systems. Hence, the next objective of the study is to find a suitable sparse direct method and its implementation, and adapt it for HW1 compute-cluster environment in terms of efficient parallel execution.

5.2. Selection of a Sparse Direct Linear Solver

> *Fair to say, there is no single algorithm or software that is the best for all types of linear systems*
>
> — Xiaoye Li, [49]

Nowadays, there exist many different and avaliable sparse direct solvers. Some of them are tunned for specific linear systems whereas others are targeted for the most general cases [49]. Some of them handle tree-task and node-data parallelism in different ways even within the same library depending on sizes of frontal matrices and other criteria [25], [37], [35]. Hence, parallel performance of a direct sparse method depends heavily on its specific implementation. Table 5.3 represents a short summary of almost all available libraries capable to run on distributed-memory machines, at the time of writing, based on works [49] and [8].

It can be clearly observed, from Table 5.3, that only MUMPS, PaStiX and SuperLU_DIST cover requirements induced by GRS, see Chapter 3, in particular: open-source license and a direct interface to PETSc. Additionally, each development group of these solvers provides technical support for its software package. Therefore, it indirectly means that the solvers are maintainable. It is interesting to notice that all libraries, mentioned above, are implementations of different sparse direct methods, namely: multifrontal (MUMPS), left-looking (PaStiX) and right-locking (SuperLU_DIST). Moreover, PaStiX and SuperLU_DIST use only static pivoting [39], [35] whereas MUMPS provides a full implementation of the threshold pivoting strategy [37], described in Subsection 5.1.2.3.

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 5.3.: A list of direct sparse linear solvers adapted for distributed-memory computations, [49], [8], where SPD - Symmetric Positive Definite

To compare the libraries, a couple of flat-MPI tests were performed using GRS matrix set and HW1 compute-cluster. From now onwards, we refer to a flat-MPI test as parallel factorizations of a matrix performed with varying values of the MPI process count from 1 to 20 and conducted on a single compute-cluster node.

PETSc library was compiled and configured with MUMPS (version 5.1.2), PasTiX (version 6.0.0) and SuperLU_DIST (version 5.4) packages using their default parameter settings. An internal built-in PETSc profiler was used to measure execution time. A time limit of 15 minutes was set up for each test-case to prevent blocking of a cluster compute-node from an unexpected long program execution. Results are summarized in Tables 5.4, 5.5, 5.6 and in appendix B where numerical values are given in seconds.

Some problems were detected during SuperLU_DIST library testing. First of all, executions of *cube-64* and *k3-2* test-cases exceeded the set time limit. Secondly, it was noticed the library was crashing during processing of *k3-18*, *cube-645* and (partially) *pwr-3d* test-cases. Debugging revealed that a segmentation fault occurred in function *pdgstrf* during the numerical factorization phase. Nonetheless, it is still unclear whether the problem was software or hardware specific. A solution or a reason of such program behavior has not been found at the moment of writing.

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 5.4.: Comparisons of parallel performance of *cube-5* matrix factorizations using MUMPS, PasTiX and SuperLU_DIST solvers with their default parameter settings

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 5.5.: Comparisons of parallel performance of *cube-64* matrix factorizations using MUMPS, PasTiX and SuperLU_DIST solvers with their default parameter settings

To complete comparison and evaluate parallel performance SuperLU_DIST library, an additional test was conducted using a 2D formulation of the Poisson problem with *100000* unknown. According to the results, SuperLU_DIST managed to complete matrix factorizations within the set time limit without crashing, however, it showed abnormal jagged strong scaling behavior. Moreover, it turned out it was the slowest in comparison to the other solvers. The results are shown in Figure 5.9.

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 5.6.: Comparisons of parallel performance of *k3-18* matrix factorizations using MUMPS, PasTiX and SuperLU_DIST solvers with their default parameter settings

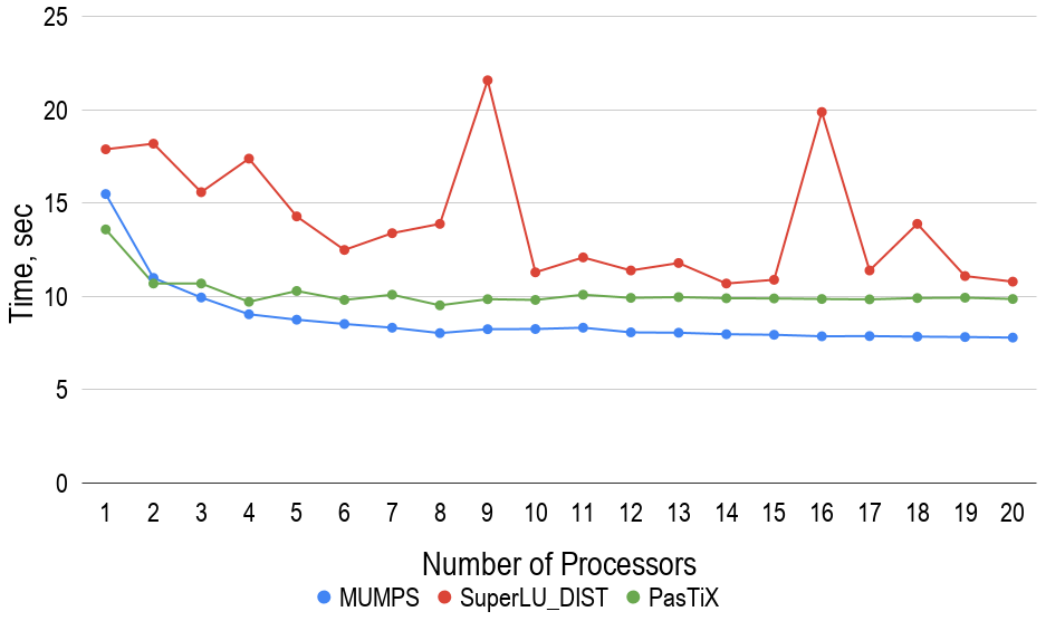


Figure 5.9.: Comparisons of parallel performance of MUMPS, PasTiX and SuperLU_DIST libraries during 5 point-stencil Poisson matrix (1000000 equations) factorizations

According to the initial and additional tests, MUMPS library showed the best parallel performance and scaling in contrast to the other solvers. No abnormal behavior during its operation was detected. In some cases, it only required to increase a multiplicative factor of estimated working space which was used to hold frontal matrices and factors L and U in memory. PaStiX was the second fastest solver according to the results of testing. However, it was often considerably slower than MUMPS. At the same time, SuperLU_DIST showed the worst results. Additionally, as it was mentioned above, we experienced some technical problems during operation of this library.

A literature review showed quite contradictory results and conclusions. For example, Gupta, Koric, and George, in [25], came to nearly the same inference with respect to MUMPS, as we did, comparing parallel performance of WSMP, MUMPS and SuperLU_DIST libraries using their matrix set. However, Kwack, Bauer, and Koric showed, in [31], that SuperLU_DIST spent the least amount of time on solving systems of linear equations in contrast to the other solvers used in their work. It is needless to say that both research groups used different matrix sets and hardware. Nevertheless, it reveals a quite important fact that a selection of a particular method and its implementation can depend heavily on a specific matrix set.

In this section, we have compared different sparse direct methods and their concrete implementations using their default parameter settings with regard to GRS matrix set. Based on the obtained results and literature review, MUMPS library is chosen for the rest of the study. In Section 5.3, we make an overview of the library and its specific traits.

5.3. Overview of MUMPS Library

Originally, MUltifrontal Massively Parallel sparse direct Solver (MUMPS) was a part of the PARASOL Project. The project was an ESPRIT IV long term research with the main goal to build and test a portable library for solving large sparse systems of equations on distributed memory systems [1]. An important aspect of the research was a strong link between the developers of the sparse solvers and industrial end users, who provided a range of test problems and evaluated the solvers [3]. Since 2000, MUMPS had continued as an ongoing project and the library have contained almost 5 main releases, at the moment of writing.

As it was mentioned in Section 5.2, MUMPS is an implementation of the multifrontal method. Therefore, MUMPS performs all three phases in sequence, namely: analysis,

numerical factorization and solution. The numerical factorization and solution phases were fully described in detail in Subsection 5.1.2.1. In this section, the analysis phase of MUMPS is examined since implementations of this phase often vary between libraries due to different performance considerations.

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

1. Fill reducing reordering
2. Symbolic factorization
3. Scaling
4. Amalgamantion
5. Mapping

1) To handle both symmetric and unsymmetric cases, MUMPS performs fill reducing reordering based on $\boldsymbol{A} + \boldsymbol{A^T}$ sparsity pattern. The library provides numerous sequantial algorithms for the reordering such as Approximate Minimum Degree (AMD) [2], Approximate Minimum Fill (AMF), Approximate Minimum Degree with automatic quasi-dense row detection (QAMD) [4], Bottom-up and Top-down Sparse Reordering (PORD) [43], Nested Dissection coupled with AMD (Scotch) [40], Multilevel Nested Dissection coupled with Multiple Minimum Degree (METIS) [29]. Additionally, MUMPS can work together with ParMETIS and PT-Scotch which are extensions of METIS and Scotch libraries for parallel execution, respectively. MUMPS also provides users with an option to select a fill-in reducing algorithm in run-time based on a matrix type, size and the number of processors [37].

2) Sparsity structures of factors L and U are computed during the symbolic factorization pre-processing step, based on permuted matrix A after fill-in reducing reordering. It gives the input information for the elimination tree building process. All computations are performed using an undirected graph $G(A)$ associated with a matrix A at this step.

3) Rows and/or columns of matrix A can be scaled during either the analysis or factorization phase in order to improve numerical solution accuracy. As an additional consequence, this pre-processing step can result in more reliable estimations of required memory space and load balancing, performed during the analysis phase, due to a reduced amount of pivoting during numerical factorization. Different scaling approaches

are adopted in MUMPS, namely: diagonal, column or column-row scalings during the numerical factorization phase, see [37] for details; scalings based on works [14], [13] and [15] for the analysis phase.

4) During the amalgamation step, described in Subsection 5.1.2.1, sets of columns with the same off-diagonal sparsity pattern are group together to create denser nodes, also known as super-nodes. The process leads to restructuring the original elimination tree to an amalgamated one of super-nodes which is also know as the *assembly tree*. The main purpose of this step is to improve efficiency of dense matrix operations.

5) A host process, chosen by MUMPS, creates a pool of tasks where each task refers to partial factorization of a node i.e. a frontal matrix. Each node belongs to one of three different types according to a size of the frontal matrix that a node refers to. Type 1 and 2 nodes represent small- and medium-sized frontal matrices, respectively. Whereas a type 3 node represents the root node of an assembly tree i.e the largest frontal matrix. MUMPS uses different parallel computational strategies, that are explained below, in order to process nodes of different types with the aim of achieving better parallel performance. The host process distributes tasks among all available processes in such a way to achieve good memory and compute balances. Figure 5.10 shows an example of a process distribution in MUMPS.

Type 1 nodes are grouped in subtrees, according to the Geist-Ng algorithm [20], and each subtree is processed by a single process to avoid the finest task granularity, which can cause high communication overheads.

In case of type 2 nodes, the host process assigns each node to one process, called the *master*, which holds fully summed rows and columns of a node as well as performs threshold pivoting and partial factorization. During the numerical factorization phase, in run-time, a master process first receives symbolic information, describing contribution block structures, from its children. Then, the master collects information concerning the load balances of all other processes and decides, *dynamically*, which of them, *slaves*, are going to participate in the node factorization. After that, the master informs the chosen slaves that a new task has been allocated for them; maps them according to a 1D block column distribution and sends them the corresponding parts of the frontal matrix. Then, the slaves communicate with the children of the master process and collect the corresponding numerical values. The slaves are in charge of assembly and computations of the partly summed rows. The computational process is illustrated in Figure 5.18, Subsection 5.4.3.

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

It is important to mention that if the root node size is less than a certain computer depended parameter value, defined internally by MUMPS, the root node will be treated as a type 2 one, [37].

An example of static/dynamic scheduling i.e. process mapping, is depicted in Figure 5.10.

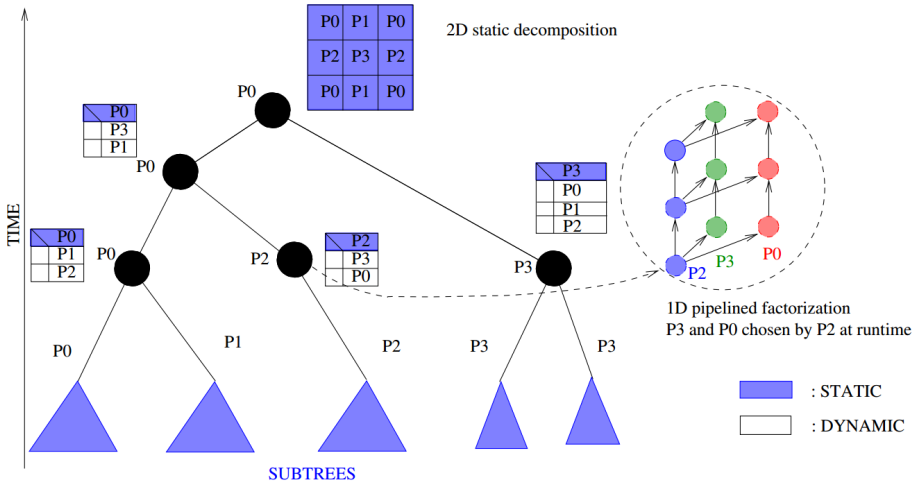


Figure 5.10.: An example of static and dynamic scheduling in MUMPS, [32]

5.4. Configuration of MUMPS Library

This section is organized as follows. Each subsection describes configuration of MUMPS with particular techniques, methods or libraries stated in the title of a subsection, *e.g. Subsection 5.4.3, i.e. "Optimized BLAS Implementations", stands for "Configuration of MUMPS Library with Optimized BLAS Implementations"*.

5.4.1. Fill Reducing Reorderings

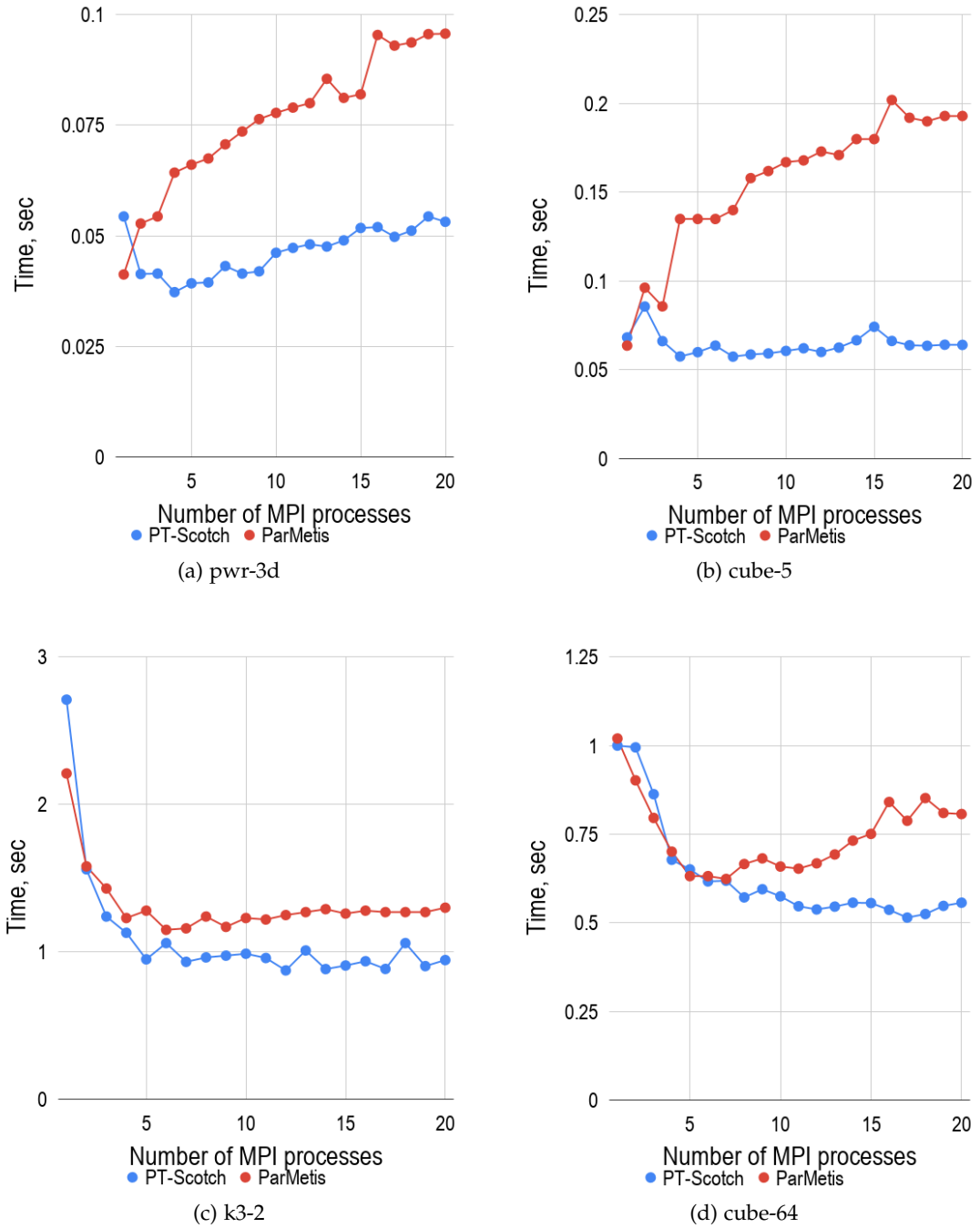
Fill reducing reordering is one of the first and the most important steps of sparse matrix factorization. As the name suggests, the step aims to reduce fill-in of L and U factors. However, it may have a strong and indirect impact on an elimination/assembly tree structure. As we discussed in Subsection 5.1.2.2, the structure defines tree-task parallelism as well as sizes of frontal matrices and, therefore, performance of the method.

MUMPS provides various algorithms for fill reducing reordering, as it was mentioned above. A detailed study and comparison of different methods were done by Guermouche, L'Excellent, and Utard, in [24], for sequential execution of the analysis phase. Guermouche, L'Excellent, and Utard noticed that 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 observed two important things. Firstly, they noticed that 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 in contrast to those trees generated by other reordering techniques, [24].

In this subsection, we are going to investigate an influence of two different parallel fill reducing reordering algorithms provided by PT-Scotch and ParMETIS libraries on parallel performance of MUMPS. The algorithmic difference between the corresponding PT-Scotch and ParMETIS subroutines was mentioned in Section 5.3.

To perform testing, PETSc, MUMPS, PT-Scotch and ParMETIS libraries were downloaded, compiled, configured and linked together, using their default parameter settings. Tests were carried out using only flat-MPI mode on HW1 compute-cluster without any explicit process pinning. The results are shown in Figures 5.11 and 5.12 as well as in appendix C.

According to the results of testing, parallel performance of MUMPS can vary signifi-



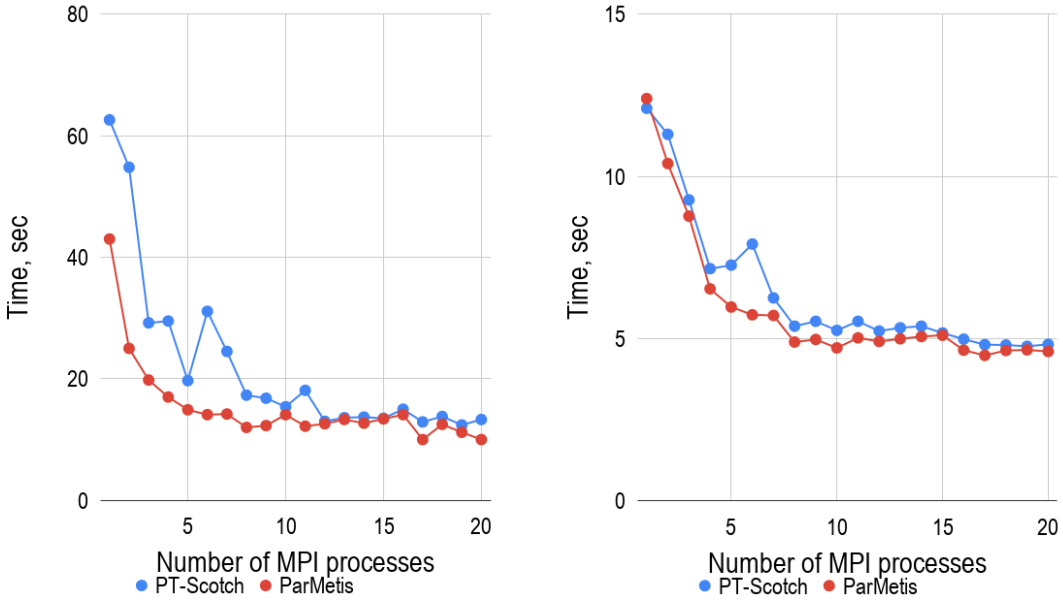
(a) pwr-3d

(b) cube-5

(c) k3-2

(d) cube-64

Figure 5.11.: An influence of different fill reducing algorithms on parallel factorizations of *pwr-3d*, *cube-5*, *k3-2* and *cube-64* matrices



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

Figure 5.12.: An influence of different fill reducing algorithms on parallel factorizations of *k3-18* and *cube-645* matrices

cantly and very sensitive to applied fill-in reducing reordering algorithms. On average, difference in execution time between the algorithms achieves almost **15%**. However, in some particular cases, *cube-5* and *pwr-3d*, the difference varies around **40-55%**.

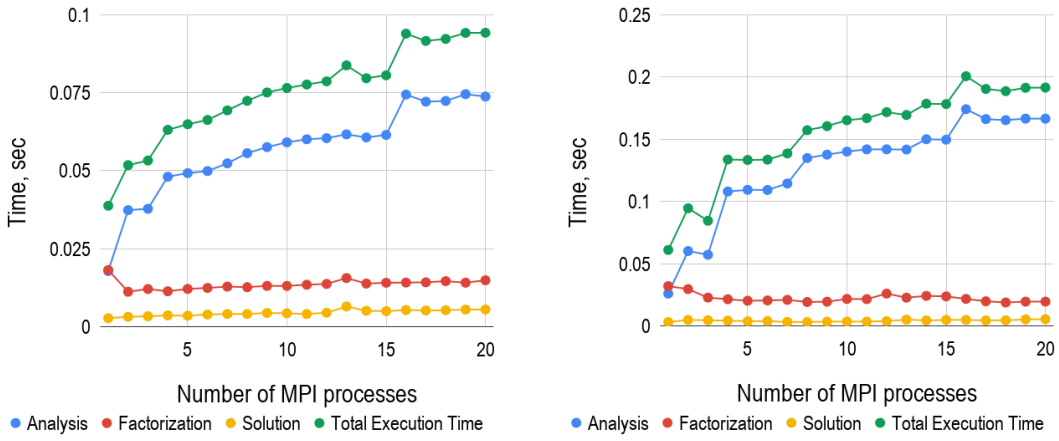
It is important to mention that both algorithms, PT-Scotch and ParMetis, are based on different heuristic approaches. It is relevant to assume that efficiency of a particular heuristic can be very sensitive to a matrix structure and size. This fact makes it difficult to predict in advance which algorithm is better to use for a specific case.

Considering results obtained using GRS matrix set, we can observe that PT-Scotch is the best choice for small- and medium-sized matrices, namely: *pwr-3d*, *cube-5*, *k3-2* and *cube-64* cases. Whereas, PerMetis tends to work better for relatively big systems, such as *k3-18* and *cube-645*, *However, we keep in mind that the number of GRS test-cases may be not enough to make such conclusion and, therefore, the matrix set must be extended considerably for a future study*.

During the testing, we noticed that applications of ParMetis to small systems of equations showed a strong negative effect on parallel performance of MUMPS. The results showed that factorization time of *pwr-3d* and *cube-5* matrices grew with the increase of the number of processing units.

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 were barely decreasing with increase of the number of processing elements, time spent on the analysis phase significantly grew. Therefore, the slow-down of MUMPS in case of these two test-cases mainly came from overheads of the analysis phase.

A careful investigation revealed that the analysis phase contained several peaks at points where the MPI processor count was equal to a power of two. We assumed the cause could result from either fill reducing reordering or process mapping steps. However, a detailed profiling and tracing of the analysis phase, which are out of the scope of this study, are required in order to give the exact answer. The results of profiling are shown in Figure 5.13.



(a) pwr-3d

(b) cube-5

Figure 5.13.: Profiling of MUMPS-ParMetis configuration applied to parallel factorizations of relatively small matrices

In this subsection, we have presented an influence of two different fill-in reducing algorithms on parallel performance of MUMPS. We have observed that a correct choice

of an algorithm can lead to a significant improvement in terms of the overall parallel execution time. We have shown there is no a single algorithm that performs the best for all test-cases. At the moment of writing, we have come to the conclusion that there is no an indirect metric to predict the best algorithm in advance for a specific system of equations. Sometimes PT-Scotch and ParMetis can result in nearly the same performance as it was, for example, in case of *CurlCurl_3* and *cant* matrices, see appendix C. Therefore, from time to time, it can be quite difficult to decide which package to use even with available flat-MPI test results. 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 5.7 and 5.8.

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 5.7.: Assignment of GRS matrices to specific fill-in reducing algorithms

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 5.8.: Assignment of SuiteSparse matrices to specific fill-in reducing algorithms

From now onwards, assignments mentioned in Tables 5.7, 5.8 are going to be used without explicitly referring to it.

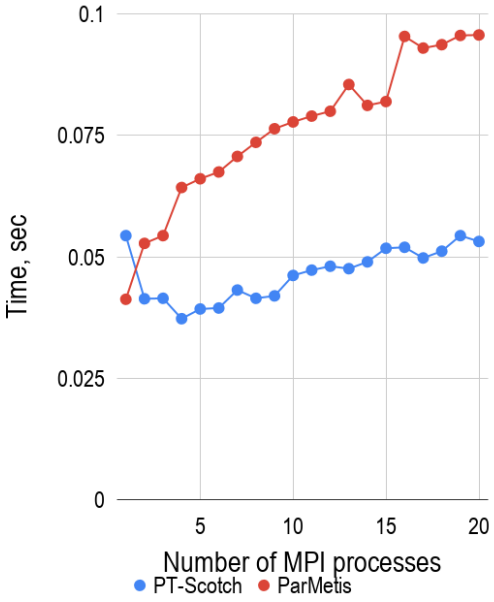
5.4.2. MPI Process Pinning

Due to intensive and complex manipulations with frontal and contribution matrices, one can assume that MUMPS belongs to a group of memory bound applications. In this case, memory access becomes a bottleneck. A common strategy to improve performance of a memory bound computer program running on distributed-memory machines is to distribute MPI processes equally across all available NUMA domains within a compute node. Given the fact that each NUMA domain possesses its own system bus, this strategy allows to reduce conjunction of memory traffic by balancing data requests equally among the memory channels.

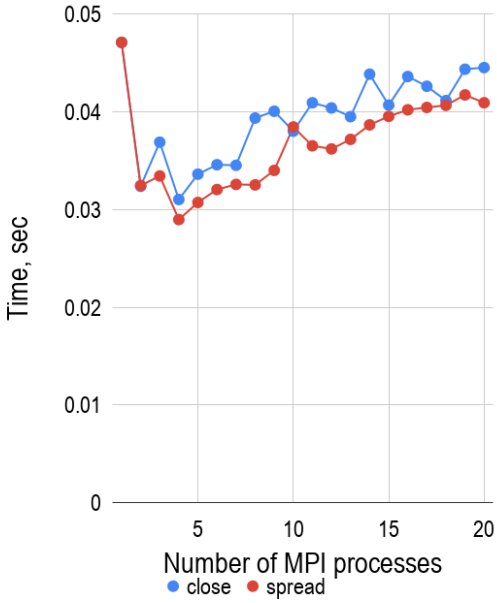
However, due to the fact that MUMPS uses both task and data parallelism as well as a complex task scheduling, it becomes difficult to state which process pinning strategy is better to use i.e. *close* or *spread*, described in Chapter 4.

Therefore, a couple of flat-MPI tests were carried out using both GRS and SuiteSparse matrix sets in order to investigate an influence of different pinning strategies on MUMPS parallel performance. For this group of tests, MUMPS was ran with the default parameter settings but with a specific fill-in reducing algorithm assigned to each test-case according to Tables 5.7 and 5.8. The tests were performed on both HW1 and HW2 machines. A comparison between different hardware also allows to investigate an influence of different numbers of independent system buses within a compute-node on parallel performance of MUMPS since HW1 and HW2 machines have 2 and 4 NUMA domains, respectively. Results are shown in Figures 5.14, 5.15, 5.16 and in appendix D. Each graph depicts the total execution time of MUMPS spent on a test-case i.e. time spent on the analysis, factorization and solution phases.

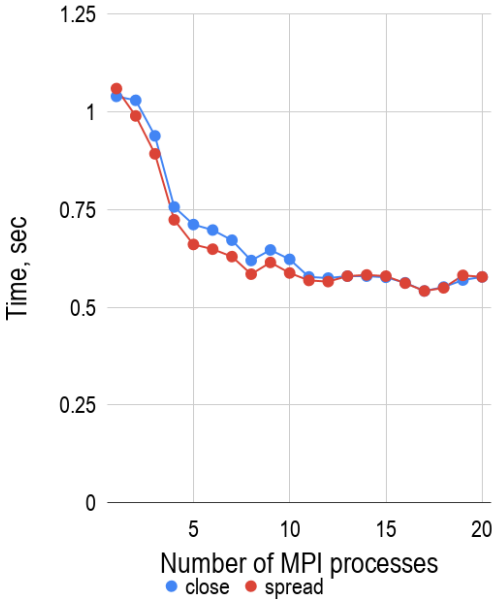
The tests revealed that, in the general case, the *spread*-pinning strategy performed better for both machines. On average, the strategy allows to reduce run-time by approximately **5.5%** and **13.8%** for HW1 and HW2 machines, respectively. The main performance gain can be observed in the middle range of the MPI process count i.e. the range between 2 and 12 MPI processes, where performance curves of *spread* and *close* strategies noticeably deviate. On the other hand, the difference becomes less and less prominent while the process count is reaching either its maximum or minimal values. In these cases, the difference between process distributions of both strategies becomes less noticeable as well. As an extreme example, the points where the process count is equal to 1 and 20 show the same performance, in case of HW1 machine which possesses only 20 cores in a compute-node, because the points basically represent exactly the same process distributions.



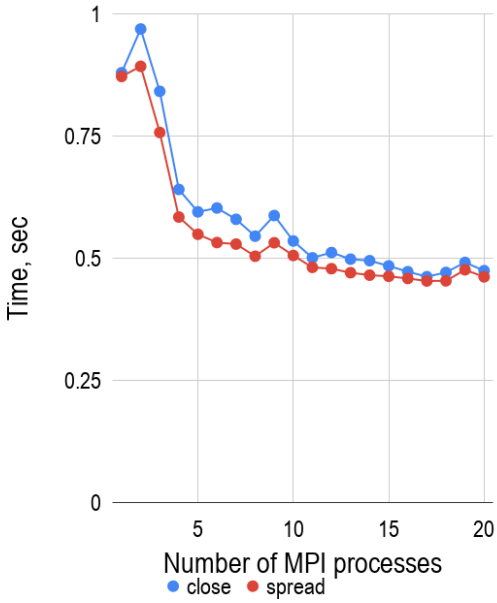
(a) HW1 - pwr-3d



(b) HW2 - pwr-3d

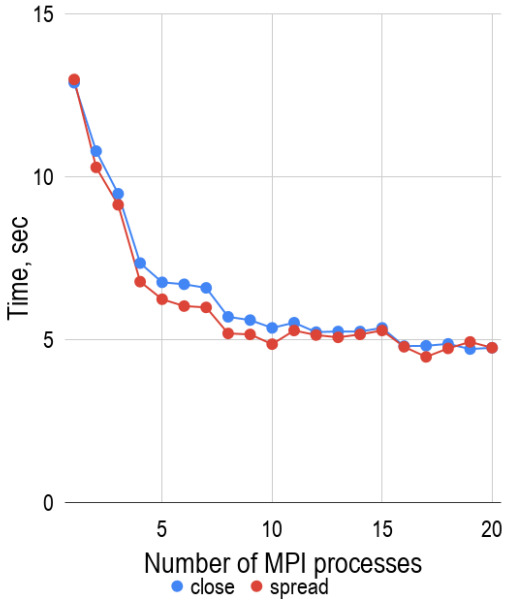


(c) HW1 - cube-64

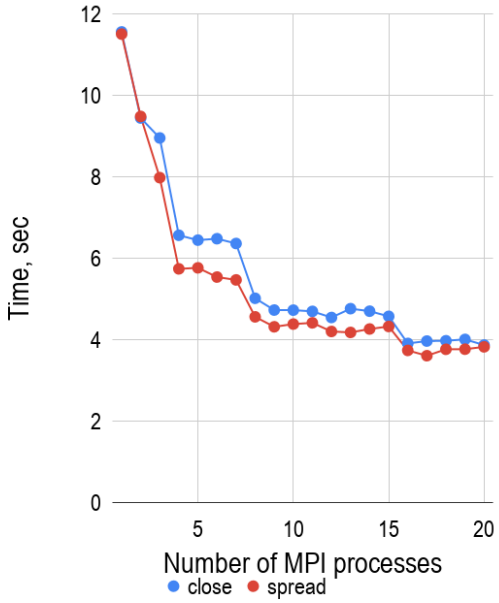


(d) HW2 - cube-64

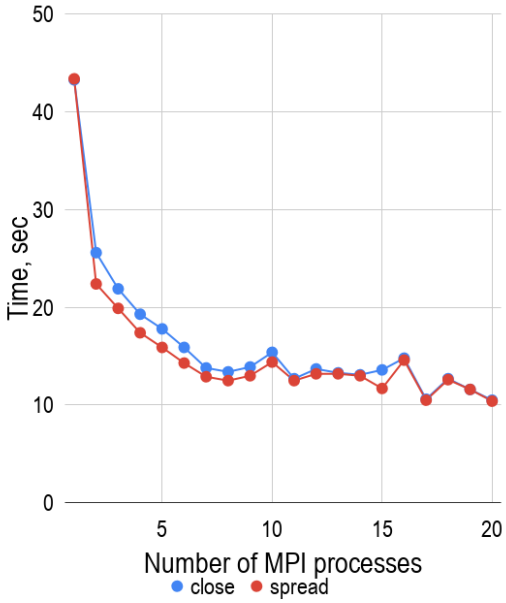
Figure 5.14.: Comparisons of *close* and *spread* pinning strategies applied to parallel factorizations of *pwr-3d* and *cube-64* matrices



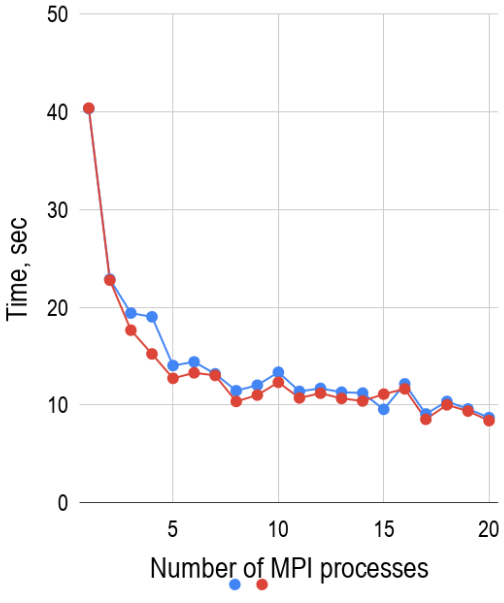
(a) HW1 - cube-645



(b) HW2 - cube-645

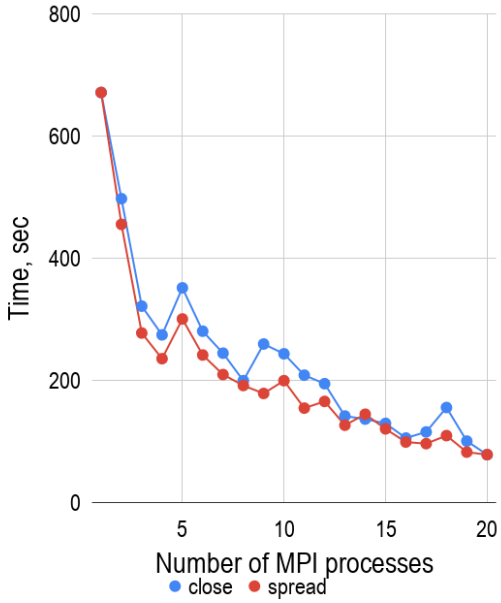


(c) HW1 - k3-18

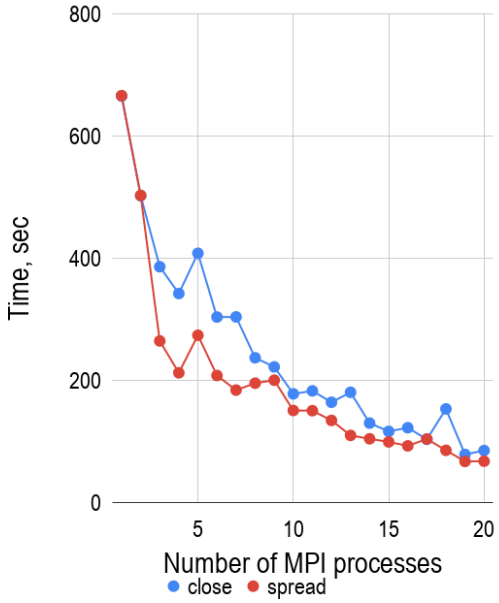


(d) HW2 - k3-18

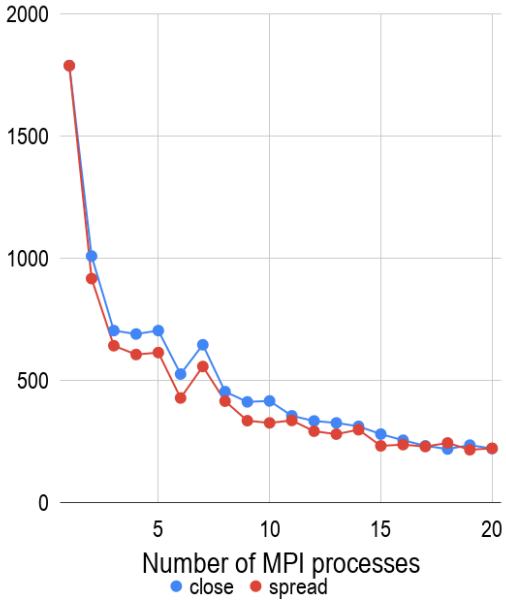
Figure 5.15.: Comparisons of *close* and *spread* pinning strategies applied to parallel factorizations of *cube-645* and *k3-18* matrices



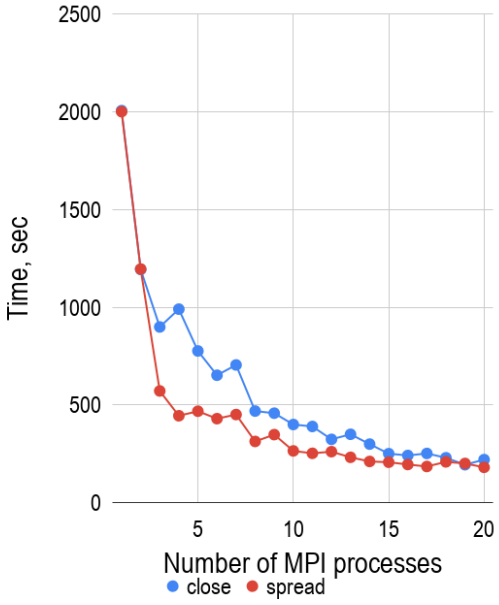
(a) HW1 - PFlow_742



(b) HW2 - PFlow_742



(c) HW1 - CurlCurl_3



(d) HW2 - CurlCurl_3

Figure 5.16.: Comparisons of *close* and *spread* pinning strategies applied to parallel factorizations of *PFlow_742* and *CurlCurl_3* matrices

It is also important to investigate performance gain around saturation points i.e. points after which further increase of the MPI process count results in either stagnation or drop of computer program speed-up. It is worth pointing out that, in our case, it becomes difficult to decide where saturation points locate because of jagged behavior of speed-up curves. For this reason, a careful analysis of each performance graph was performed based on values of speed-up, efficiency and our subjective opinion. The results are summarized in Tables 5.9 and 5.10 where each table is organized as follows. Each row of a table contains five fields and provides information about parallel performance of MUMPS at the saturation point of a test-case relatively to the *spread* pinning strategy. The first one is the name of a test-case. The second, fourth and fifth ones show values of the MPI process count, speed-up and parallel efficiency at the saturation point, respectively. The third field shows a gain in parallel factorization time of the *spread* pinning strategy over the *close* one at this point, in percent.

	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 5.9.: Comparisons of MUMPS parallel performance at the saturation points in case of factorization of GRS matrix set

A study of Tables 5.9 and 5.10 reveals that HW2 machine performs slightly better in contrast to HW1 one with respect to parallel performance around the saturation points. This results are different from the overall performance gain mentioned above, however, they reflect the same trend. Additionally, it can be clearly observed that increase of NUMA domains always results in improving efficiency and speed-up of MUMPS.

In this subsection, we have shown an influence of different MPI process distributions and the number of NUMA domains on MUMPS parallel performance. We have observed that application of the *spread* process distribution is always advantageous together with increase of the number of NUMA domains.

Matrix Name	HW1 MPI	HW1 Gain w.r.t "close", %	HW1 Speed up	HW1 Effi-ciency	HW2 MPI	HW2 Gain w.r.t "close", %	HW2 Speed up	HW2 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 5.10.: Comparisons of MUMPS parallel performance at the saturation points in case of factorization of SuiteSparse matrix set, where ROM stands for Run Out of Memory

The result of this study can be relevant for energy-efficient parallel computing where strong requirements to program efficiency are applied. This fact usually forces the user to reduce the process count and go away from the saturation point in order to keep values of efficiency around **0.7-0.8**. In this case, performance of MUMPS can be improved by **15-20%** in contrast to a straightforward process pinning i.e. *close* strategy.

Taken into account results of testing, *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 5.3: An example of setting *spread* process pinning using advanced OpenMPI command line options

5.4.3. Optimized BLAS Implementations

To perform column eliminations of fully summed blocks of type 2 nodes, MUMPS intensively calls GEMM, TRSM and GETRF subroutines [33] which are parts of BLAS and LAPACK libraries, see Figures 5.17 and 5.18 as an example. Additionally, MUMPS

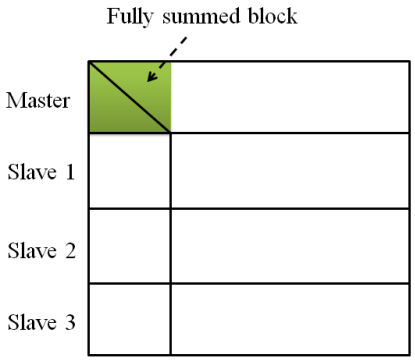


Figure 5.17.: One dimensional block column distribution of a type 2 node in MUMPS

uses ScaLAPACK library subroutines to perform parallel factorization of the root [37], according to the procedure described in Section 5.3.

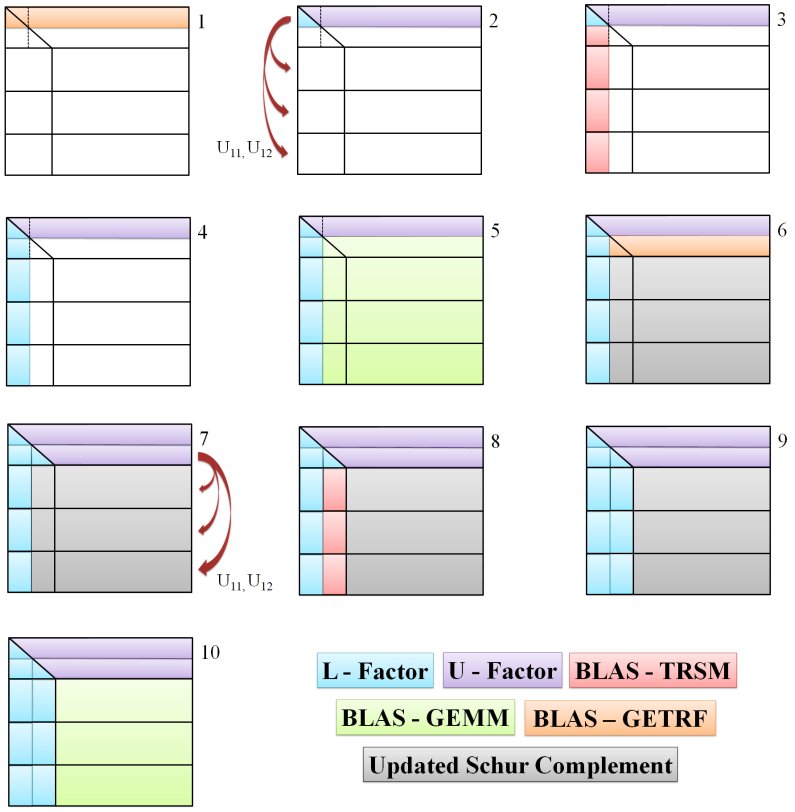


Figure 5.18.: An example of type 2 node factorization implemented in MUMPS

BLAS, LAPACK and ScaLAPACK libraries 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, [38].

The goal of BLAS library is provision of high efficient implementations of common dense linear algebra kernels achieved by high rates of floating point operations per memory access, low cache and Translation Lookaside Buffer (TLB) miss rates.

Both LAPACK and ScaLAPACK are designed in such a way so that as much as possible computations are performed by calling BLAS subroutines. Figure 5.19 represents software dependencies between the libraries. Hence, this software architecture allows to achieve high computational performance for operations such as LU, QR, SVD decompositions, triangular solve, etc., on modern computers and distributed-memory machines by an efficient implementation of BLAS library. However, the Netlib BLAS implementation is written for an abstract general-purpose central processing unit where hardware parameters are based on market statistics. Therefore, it is not possible to achieve the maximum possible performance on specific hardware.

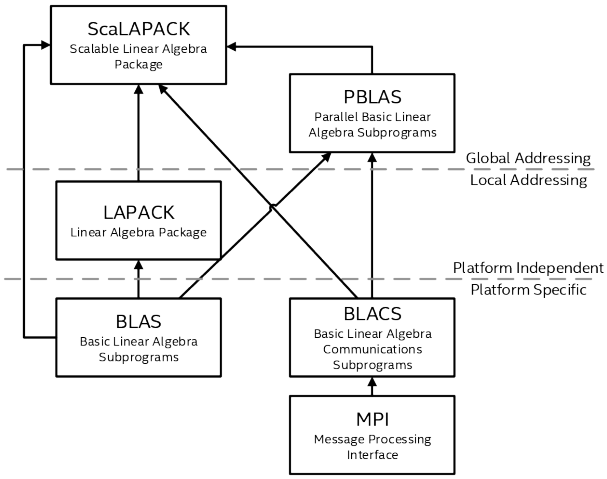


Figure 5.19.: Software dependencies between Netlib libraries, [26]

There exist special-purpose, hardware-specific implementations of BLAS 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 as a standard, or a reference, and thus

overwrite all its subroutines with additional tuning and optimization. This approach makes it easy to replace one BLAS implementation by another one by substituting the corresponding object files during the linking stage. As a result, the source code of an application which calls BLAS, LAPACK or ScaLAPACK subroutines remains the same without a need to perform any source code modifications.

Table 5.11 shows commercial and open-source tunned BLAS implementations available on the market today.

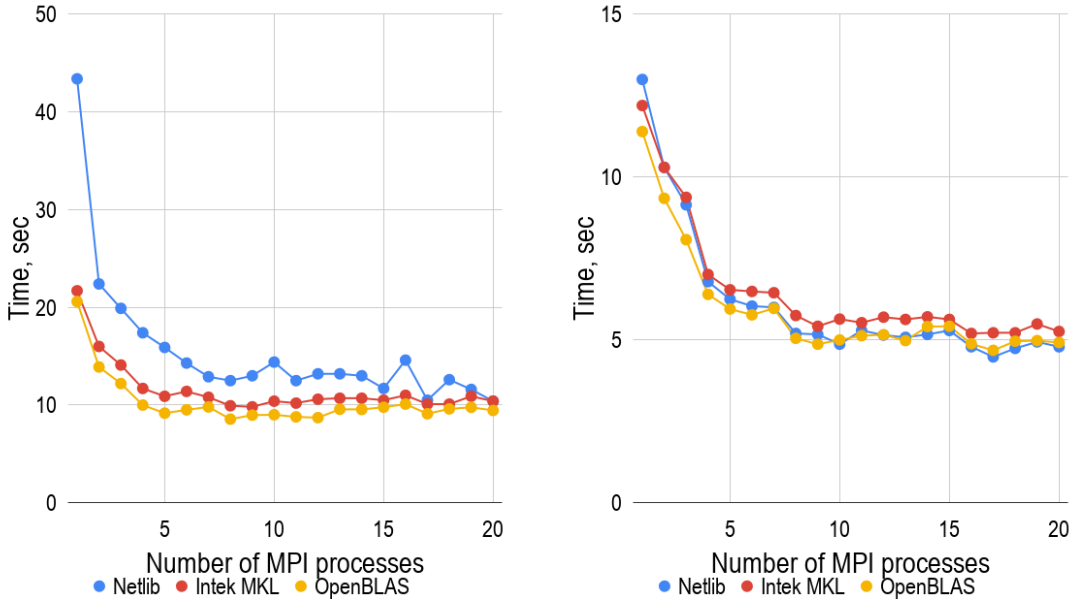
Name	Description	License
Accelerate	Apple's implementation for macOS and iOS	proprietary license
ACML	BLAS implementation for AMD processors	proprietary license
C++ AMP	Microsoft's AMP language extension for Visual C++	open source
ATLAS	Automatically tuned BLAS implementation	open source
Eigen BLAS	BLAS 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 5.11.: Commercial and open source BLAS libraries [46]

Among all libraries listed in Table 5.11 there were only four available in HW1 machine environment, 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 ATLAS configuration routines to find the best loop transformation parameters for 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 study and we excluded it from our primary list of candidates. Moreover, during installation, one has to explicitly specify the number of OpenMP threads that are going to be forked 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 the library. Thus, only 3 versions of MUMPS-PETSc (linked with Netlib BLAS, Intel MKL and OpenBLAS) library were compiled, installed and tested using both GRS and SuiteSparse matrix sets and 1 thread per MPI process i.e. flat-MPI mode. Results of testing were obtained on HW1 machine and are represented in Figures 5.20, 5.21 and appendix E.

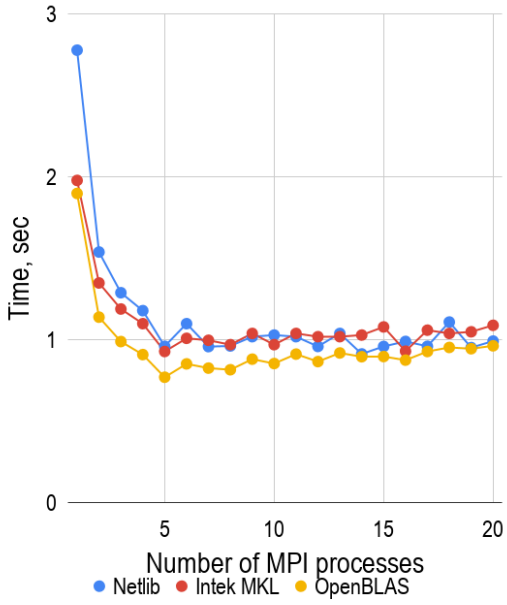
The tests show that OpenBLAS outperforms both Netlib and Intel MKL libraries in case of GRS matrix set. On average, OpenBLAS is about **13%** faster than the default Netlib implementation and approximately **21%** faster than Intel MKL library. It is interesting to notice that Intel MKL library turns out to be slower than the default Netlib BLAS implementation for small- and medium-sized GRS matrices in almost **52%** and **2%**, respectively. At the same time, both tuned libraries, OpenBLAS and Intel MKL, show significant performance gain in comparison to the standard Netlib BLAS implementation in case of SuiteSparse matrix set. The libraries reduce the execution time by almost **50%** on an average. In opposite to GRS matrix set, it turns out that Intel MKL is often faster than OpenBLAS for almost all test-cases from SuiteSparse matrix set. However, the difference between them is negligibly small. The result of the comparison are summarized in Tables 5.12 and 5.13.

It can be clearly observed from the tables that test-cases derived from GRS matrix set demonstrate insignificant improvements in execution time in contrast to the tests generated with SuiteSparse matrix set. This may be explained by relatively small numbers of type 2 nodes in assembly trees resulted from GRS test-cases. In this case, the trees are mainly formed with the root and type 1 nodes. As it was mentioned in Section 5.3, type 1 nodes are grouped in subtrees and each subtree is processed by a single MPI process. According to the documentation, it is not clear whether MUMPS calls BLAS subroutines while processing a type 1 node. Even if it is a case performance of BLAS can be limited because of small sizes of frontal matrices of such nodes.

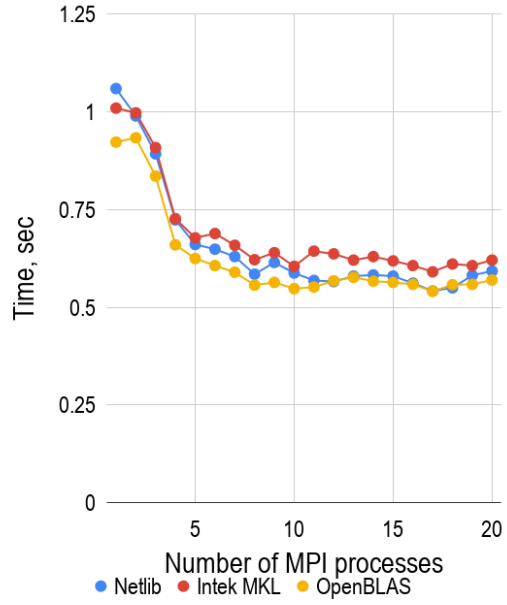


(a) k3-18

(b) cube-645

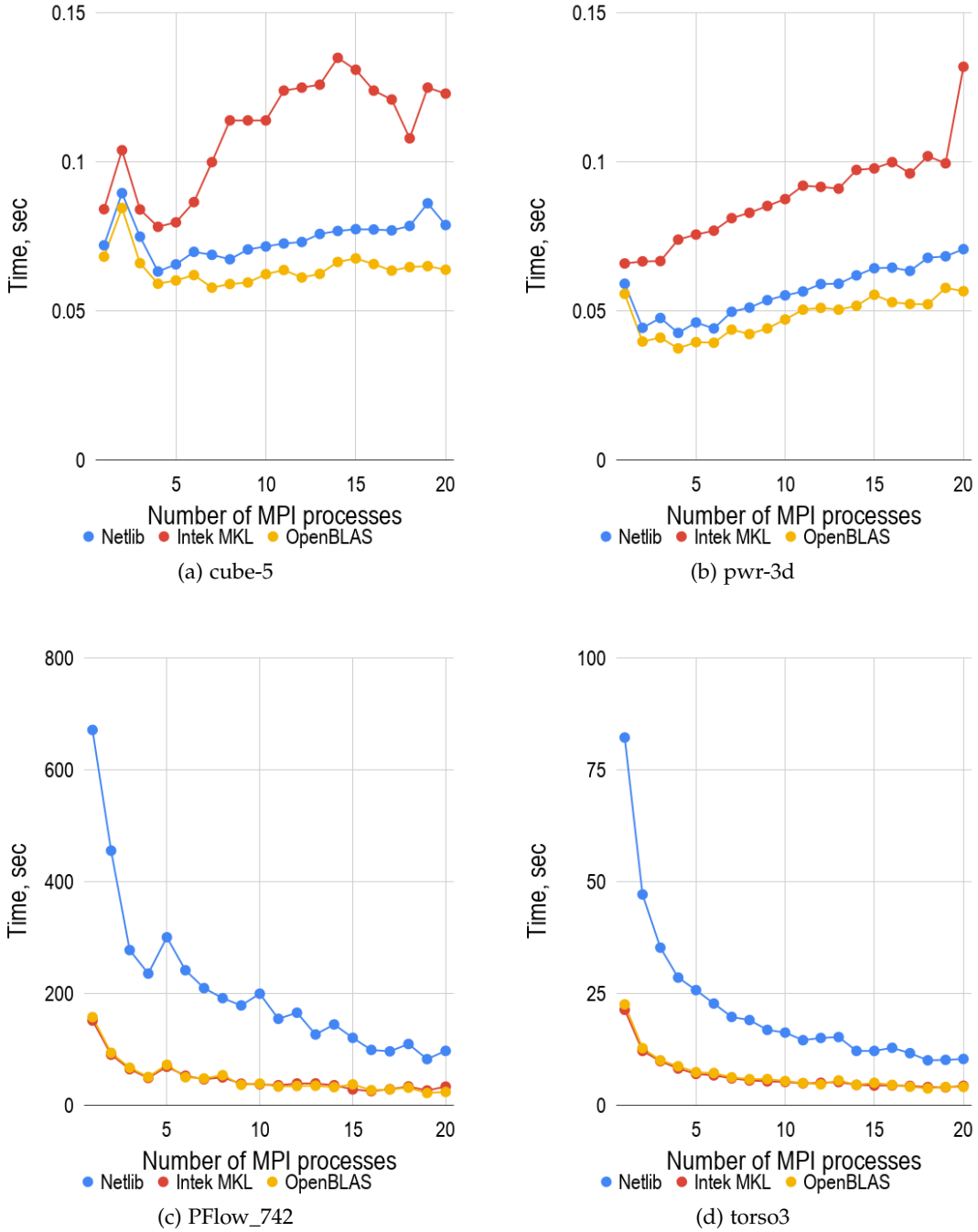


(c) k3-2



(d) cube-64

Figure 5.20.: Comparisons of parallel factorization of GRS matrix set performed on HW1 machine using MUMPS solver linked to different BLAS implementations



(a) cube-5

(b) pwr-3d

(c) PFlow_742

(d) torso3

Figure 5.21.: Comparisons of parallel factorizations of GRS and SuiteSparse matrix sets performed on HW1 machine using MUMPS solver linked to different BLAS implementations

Matrix Name	Average performance gain of OpenBLAS relatively to Netlib %	Average performance gain of IntelMKL relatively to Netlib %	Average 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 5.12.: Comparisons of different MUMPS-BLAS configurations applied to GRS matrix set

Matrix Name	Average performance gain of OpenBLAS relatively to Netlib %	Average performance gain of IntelMKL relatively to Netlib %	Average 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 5.13.: Comparisons of different MUMPS-BLAS configurations applied to SuiteSparse matrix set

We assume that, in the general case, lack of type 2 nodes in an assembly tree can be due to an inefficient amalgamation process of the corresponding elimination tree resulted from the matrix sparsity pattern.

Based on the obtained results, comparisons between different matrix sets and our reasoning, we presume that ATHLET generates linear systems resulting in such trees where type 1 nodes predominate over the others. We can assume it is due to specifics of numerical spacial and time integration explained in Section 2.1.

In this subsection, we have discussed where and how MUMPS utilizes BLAS, LAPACK and ScaLAPACK libraries. We have compared two tuned BLAS implementations with the baseline, Netlib BLAS, using two different matrix sets. We have shown the overall statistics of the obtained results and come to the conclusion that MUMPS-OpenBLAS configuration is the best one for GRS matrix set. Additionally, we have given reasoning for a noticeable difference between results obtained from different matrix sets as well as we have talked about probable specifics of linear systems generated by ATHLET.

5.4.4. Hybrid MPI/OpenMP Computing

As it was mentioned in Section 5.3, 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 [10], of exploiting shared memory parallelism in MUMPS. The authors showed that it was possible to achieve some improvements in multicore systems using multithreading, given a purely MPI application. However, later L'Excellent and Sid-Lakhdar mentioned, in [33], that adaptation of the existing code for NUMA architecture was still a challenge because of memory allocation, memory affinity, thread pinning and other related issues.

In spite of an advantage of natural data locality of message-passing applications, a general motivation for switching to a hybrid mode, a mixed MPI/OpenMP process/thread distribution, is to reduce communication overheads between MPI processes. According to the profiling results obtained by Chowdhury and L'Excellent, MUMPS contained four main regions 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 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. Hence, parallelization of region 1 can be achieved by linking a suitable BLAS library whereas regions 2, 3 and 4 can be multi-threaded by inserting appropriate OpenMP directives above the corresponding loop statements.

A detailed review of works [33] and [10] reveals that, in general, a pure OpenMP or mixed MPI/OpenMP strategy can reduce run-time of MUMPS. On average, factorization time is reduced by **14.3%** and in some special cases improvements reach about **50.4%**, according to analysis performed on data provided in the papers. However, at the same time, the results also show that sometimes a flat-MPI mode can significantly outperform other hybrid mixed strategies.

By and large, the results show two important aspects. Firstly, performance of a specific strategy depends heavily on a resulting assembly tree and thus on a matrix sparsity pattern and applied fill reducing reordering. 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 dependeds on a ratio of large and small fronts. For example, they noticed that more threads per MPI process resulted in better parallel performance in case of high ratios. 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 the notion of small and large ratios in [33].

It is also interesting to notice that parallelization of region 1 using a multi-threaded BLAS library gives the most of the parallel performance improvement for mixed or pure OpenMP strategies, according to analysis of results from [33]. Whereas, multi-threading of regions 2, 3, 4 has only a small positive effect i.e it reduces numerical factorization run-time by only **0.66%** on an average.

This outcome is expected because BLAS subroutines, especially level 3, re-use data stored in caches as much as possible and thus achieve high ratios of floating point operations per memory access which is essential for efficient multi-threading. Meanwhile, regions 2, 3, 4 mainly perform initialization of variables, data movements and

executions of *if-statements* which always result in low computational intensity.

We have to admit that both works, [10] and [33], 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 4th release of MUMPS library, the developers have persistently recommended to use only hybrid strategies e.g. *one MPI process per socket and as many threads as the number of cores*, [37].

As an initial test, we compared an 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 correct way, without any conflicts between them, the following OpenMP environment variables were set as follows:

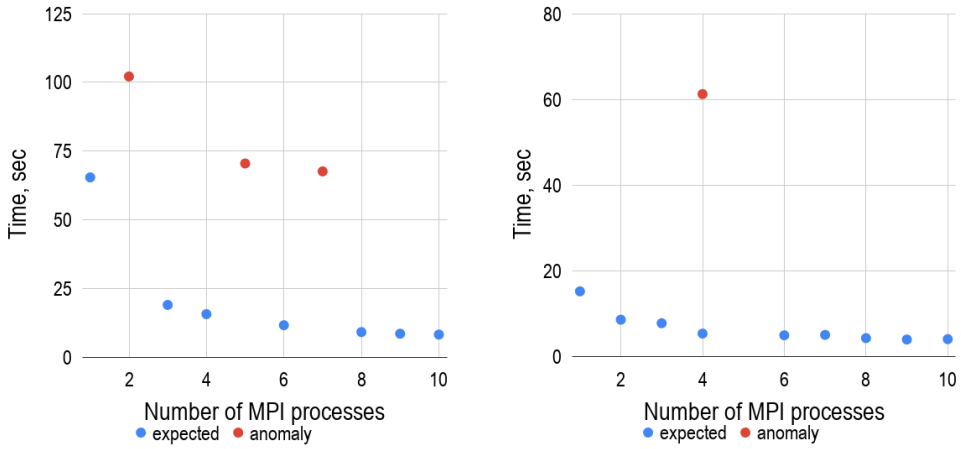
- OMP_PLACES=cores
- OMP_PROC_BIND=spread

During the testing, we found that sometimes execution time of MUMPS-OpenBLAS configuration abnormality increased. For instance, in case of parallel factorization 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* software as an interactive process viewer. Figure 5.23 shows a snapshot taken during factorization of matrix *k3-18* running with 1 MPI process and 20 threads.

It is difficult to state what exactly caused such behavior. However, Chowdhury and L'Excellent also reported about the same problem 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 [10]. For this reason, we decided to use only Intel MKL library for the rest of the study because there were no such thread-conflicts detected during operation of MUMPS-Intel MKL configuration.

Only common mixed MPI/OpenMP strategies were tested in order to check an influence of shared-memory parallelism on parallel performance of MUMPS as well as to limit an amount of testing. The following strategies were chosen: 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



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

Figure 5.22.: Anomalies of parallel executions of MUMPS-OpenBLAS configuration during factorizations of large-sized GRS matrices running with 2 OpenMP threads per MPI process

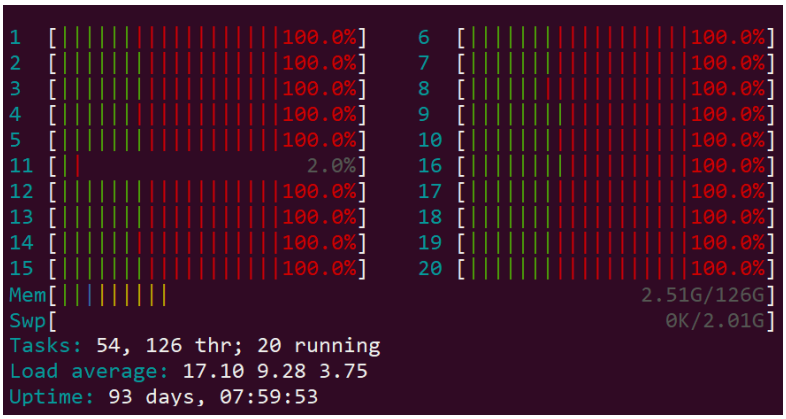


Figure 5.23.: Thread conflicts of MUMPS-OpenBLAS configuration detected during parallel factorization of matrix *k3-18*, where green - application threads, red - system threads

with the aim of checking whether results would be consistent between different hardware running under different operating and environment settings. Results of testing are represented in Tables 5.14 5.15, 5.16 and 5.17 where numerical values 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 5.14.: Compassions of different hybrid MPI/OpenMP modes used for parallel factorization of 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 5.15.: Compassions of different hybrid MPI/OpenMP modes used for parallel factorization of GRS matrix set on HW2

According to analysis of obtained results and flat-MPI performance graphs from Subsection 5.4.3, we have noticed that an optimal hybrid MPI/OpenMP mode locates near the saturation point of the corresponding flat-MPI test. Generally speaking, a location of the saturation point is specific for each matrix and, therefore, there is no way to predict a mode in advance. However, having known the point, an amount of testing can be considerably reduced by searching around and applying different mixed MPI/OpenMP strategies.

The results show that average performance gain is around **2.1%** in case of GRS matrix set for HW1 hardware, excluding small test-cases such as *cube-5* and *pwr-3d* from statistics. We consider these two scenarios, *cube-5* and *pwr-3d*, as specific ones because their execution time with 20 MPI processes using flat-MPI mode is originally slower in contrast to the sequential execution and, therefore, it is relevant to assume

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 5.16.: Compassions of different hybrid MPI/OpenMP modes used for parallel factorization of 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 5.17.: Compassions of different hybrid MPI/OpenMP modes used for parallel factorization of SuiteSparse matrix set on HW2, where ROM stands for Run Out of Memory

that the improvement came only from reducing of the MPI process count. At the same time, much optimistic results were obtained from experiments conducted on HW2 machine where performance gain reached almost **31%** for the same test-cases.

Results obtained with SuiteSparse matrix set demonstrate much better performance improvements from hybrid parallel computing obtained on both hardware. On average, execution time improves by more than **15%** running tests on HW1 and approximately by **41%** on HW2, excluding *Geo_1438* from the statistics. The best result was obtained

exactly in case of *Geo_1438* test-case on both machines where execution time dropped about **3 times** for all hybrid modes in contrast to the corresponding flat-MPI one. We assume it may occur because of a high ratio of large and small fronts of this particular test-case.

According to the outcomes of testing, we have observed a negligible improvement in MUMPS parallel performance from the application of the multi-threaded Intel MKL BLAS library to GRS matrix set. Such unimpressive results can be explained with the same reasoning given in Subsection 5.4.3 i.e. lack of type 2 nodes. Moreover, in case of GRS matrix set, parallel efficiency drops significantly probably due to inefficient utilization of additional processing elements i.e cores. However, at the same time, results obtained using SuiteSparse matrix set have shown an advantage of hybrid parallel computing, especially in case of *Geo_1438* matrix factorization.

These contradictory results obtained from two different matrix sets second our reasoning about specifics of linear systems generated by ATHLET software. Again, we presume that assembly trees resulted from GRS matrices are mostly formed with subtrees filled with type 1 nodes where each subtree is processed by a single MPI process. Hence, parallel factorization of GRS matrices mainly gets benefit from MPI parallelization that can be clearly observed from the results.

In this subsection, we have discussed how MUMPS adopts hybrid parallel computing. As it is in case of fill reducing reordering algorithm selection, Subsection 5.4.1, it is not possible to find an optimal mixed MPI/OpenMP strategy in advance without performance testing. We have come to the conclusion that flat-MPI mode is the best one for GRS matrix set and provided our reasoning for that. Generally speaking, there are 3 reasons to use this mode in our case. Firstly, the mode always resulted in more efficient hardware utilization. Secondly, MUMPS-Intel MKL configuration running with optimal hybrid MPI/OpenMP strategies can deteriorate performance gain obtained with MUMPS-OpenBLAS flat-MPI configuration, shown in Subsection 5.4.3. Finally, efficient utilization of flat-MPI strategy only demands to find an optimal MPI process count i.e the saturation point on a performance graph. Hence, it leads to a significant reduction of testing due to a reduced number of parameters which are needed to be taken into account.

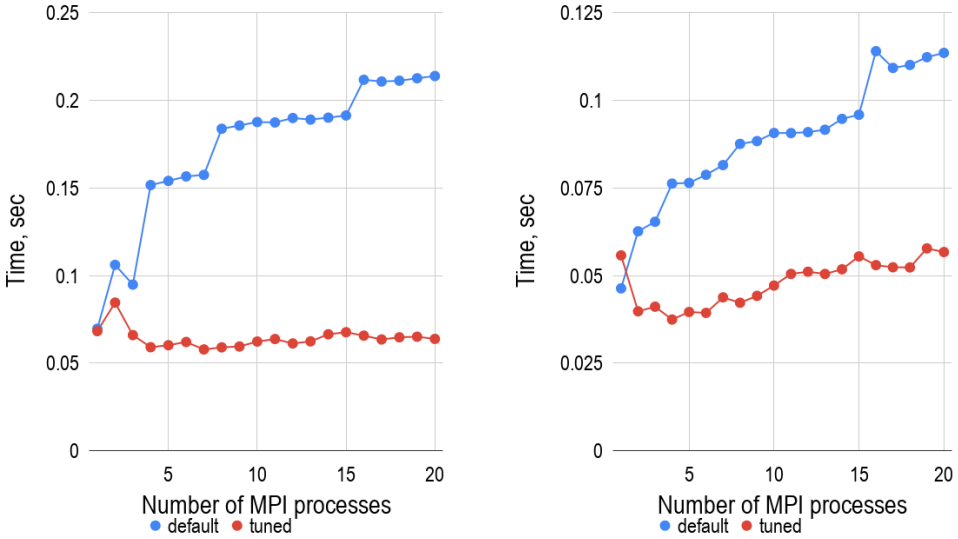
5.5. Results

Figures 5.24 and 5.25 show comparisons of MUMPS parallel performance before and after applications of the optimal MUMPS-MPI parameter settings, found in Subsections 5.4.1, 5.4.2, 5.4.3, and 5.4.4, to GRS matrix set. Results labeled as *default* were obtained using the fill reducing reordering algorithm provided by ParMetis library because it had been used by ATHLET users before the current study.

On average, factorization time is reduced by **51.4%** for small-sized linear systems, *cube-5* and *pwr-3d*. As it was expected, the most significant performance gain mainly comes from a correct choice of a fill reducing reordering algorithm. Moreover, the application of PT-Scotch to these systems of equations results in a drastic change of strong scaling behavior, see Figures 5.24a and 5.24b, which allows to reduce execution time by approximately **17%** in contrast to the sequential execution of MUMPS running with the default parameters.

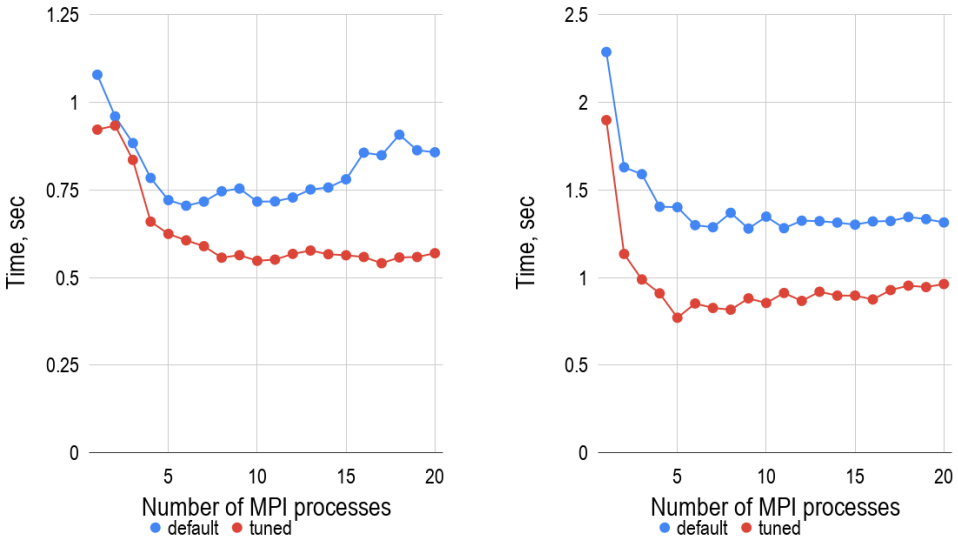
Execution time spent on factorizations of medium-sized systems, such as *cube-64* and *k3-2*, drops in **1.4** times on an average. We have noticed that strong scaling of *cube-64* matrix factorization considerably improves. Additionally, the application of PT-Scotch to *cube-64* test-case results in shifting of the optimal MPI process count, the saturation point, from 5 to 10 and, as a result, it reduces execution time of parallel factorization. The application of all optimal settings results in reduction of execution time around the corresponding saturation points by almost **31%** on average for this type of GRS matrices.

Improvements in parallel factorization of large-sized GRS systems comes only from optimal processes pinning and configuration of MUMPS with OpenBLAS library because of usage of the same fill reducing reordering algorithm, namely: ParMetis, according to the assignment Table 5.7. On average, performance increased by almost **20%** in case of *k3-18* test-case and only by **1.3%** for *cube-645* one. This difference in results can be explained by the fact that the assembly tree of *cube-645* test-case may lack type 2 nodes. However, the saturation points of both test-cases are shifted towards lower values of the MPI process count which result in a considerably improvement of hardware utilization. For example, a detailed study of *k3-18* performance graph, Figure 5.25b, shows the optimal value of the MPI process count decreases from 17 to 8 and, at the same time, execution time drops by almost **19%**. These two effects result in almost **13%** jump of parallel efficiency. The same trend can be observed for *cube-645* test-case as well.



(a) small system: cube-5

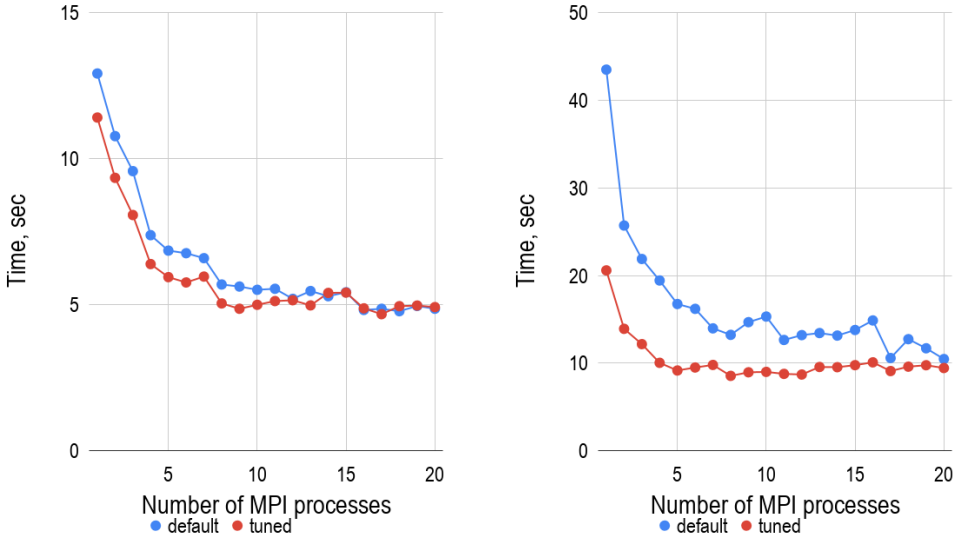
(b) small system: pwr-3d



(c) medium system: cube-64

(d) medium system: k3-2

Figure 5.24.: Comparisons of parallel factorizations of small- and middle-sized GRS matrices between applications of the default and optimal MUMPS configurations



(a) large system: cube-645 (b) large system: k3-18

Figure 5.25.: Comparisons of parallel factorizations of large-sized GRS matrices between applications of the default and optimal MUMPS configurations

By and large, in this subsection we have shown that applications of the optimal parameter settings to MUMPS lead to total accumulative improvements in both factorization time and hardware utilization.

5.6. Conclusion

In this chapter, we have examined different types of sparse linear solvers applied to linear systems generated by ATHLET software resulting from numerical integration of thermo-hydraulic computations. We have come to the conclusion that, in spite of better scalability and parallel efficiency of iterative methods due to efficient data-based parallelism, direct sparse linear solvers are well suitable for this purpose because of their robustness, see Subsection 5.1.3.

In Section 5.2, we tested different direct sparse solvers, namely: MUMPS, SuperLU_DIST and PasTiX. MUMPS showed better parallel performance among the others according to the results of testing and, therefore, was chosen for the following study where we mainly focused on performance tuning of the library.

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

1. correct selection of a fill reducing reordering algorithm
2. destribution of MPI processes among multiple NUMA domain within a compute node
3. configuration of MUMPS with an optimal, tuned BLAS library implementaion
4. execution of MUMPS with optimal hybrid MPI/OpenMP process/thread distributions

Testing was performed using two different matrix set, GRS and SuiteSparse, on two different computer-clusters, HW1 and HW2, see Chapter 4, in order to check consistency of obtained results. In this section, we give most general conclusions relevant to only GRS matrix set and HW1 cluster as targets of the study. The reader can become familiar with detailed conclusions relative to both matrix sets and hardware given at the end of each subsection that we are going to reference to.

1. In Subsection 5.4.1, it has been shown that parallel performance of MUMPS is quite sensitive to applied fill-in reducing reordering algorithms. A correct choice of the algorithm can lead to a significant improvement in execution time and strong scaling behavior. We have noticed that MUMPS performs factorizations of small- and medium-sized matrices faster using PT-Scotch library whereas large-sized problems tend to get benefit from the algorithm provided by ParMetis. We assume that the obtained conclusion may be inaccurate due to a small size of GRS matrix set. At the moment of writing, we have not found any indirect metric to predict a correct choice of an algorithm beforehand. Hence, we encourage ATHLET users to perform similar testing described in the subsection before running thermo-hydraulic simulations on distributed-memory machines to achieve better performance of parallel computations.

2. In Subsection 5.4.2, an influence of different process pinning strategies on MUMPS parallel performance has been investigated. The tests have shown that an equal distribution of MPI process among all available NUMA domains always results in additional performance gain.

3. In Subsection 5.4.3, we tested MUMPS configured with 3 different implementations of BLAS library, namely: Netlib, OpenBLAS and Intel MKL. The results have shown

the application of OpenBLAS library always results in better parallel performance.

4. In Subsection 5.4.4, we have investigated an impact of various MPI/OpenMP process/thread distributions on parallel factorizations of GRS matrices within a compute-node. We have observed that multi-threading of OpenBLAS library in MUMPS leads to multiple thread conflicts which sometimes result in significant slow-down of the solver. Results obtained with MUMPS-Intel MKL configuration have demonstrated a negligible improvement in solver execution time resulting in a significant parallel efficiency drop, probably due to inefficient usage of additional processing elements utilized by forked Intel MKL threads. At the end, we have concluded that flat-MPI mode is the best one for matrices generated by ATHLET software.

In Section 5.5, we have studied the overall impact of introduced configuration changes found in Subsections 5.4.1, 5.4.2, 5.4.3 and 5.4.4. Testing shows the changes result in a positive accumulative effect leading to considerable improvements of both factorization time and hardware utilization.

During the study, we have noticed that optimal values of the MPI process count lay within the range between 1 and 4 in case of small-sized GRS matrices and 4 and 8 for middle- and large-sized problems. An exact value is impossible to predict beforehand and, therefore, it always demands individual, problem-specific testing.

6. Improvement of ATHLET-NUT Communication

6.1. Jacobian Matrix Compression

The main goal of Jacobian matrix compression is minimization of a number of non-linear function evaluations which are usually quite expensive computational operations. The 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, a finite difference method can be used to compute a Jacobian matrix approximation in the following way:

$$\frac{1}{\epsilon}(F(y+\epsilon e_k) - F(y)) \approx J(y)e_k, \quad 1 \leq k \leq N \tag{6.1}$$

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, ϵ is a small step size.

Equation 6.1 does not exploit Jacobian matrix sparsity and thus such estimation of the Jacobian matrix requires N function evaluations.

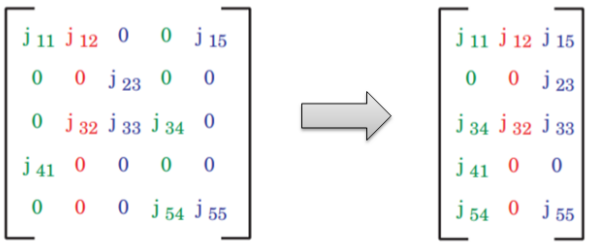


Figure 6.1.: An example of matrix coloring and compression, [19]

A compression algorithm is based on a notion of *structurally orthogonal* columns i.e. columns which do not share any non-zero entry in a common row. Figure 6.1 shows

an example of matrix compression where each color denotes independent *structurally orthogonal* columns.

Having obtained a compressed form of the Jacobian, another set of vectors $d \in \mathbb{R}^N$, also known as seed vectors, can be used to perform function perturbations 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 [19]. 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 $F(y + d)$ [19].

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. It means it is a NP-hard problem and, therefore, a huristical approach is required. Gebremedhin, Manne, and Pothen, in [19], conducted one of the most recent studies in this field and summarized different matrix partitioning algorithms proposed over the last 20 years. Currently, a Jacobian matrix compression algorithm has been successfully implemented in NUT by means of the corresponding built-in PETSc subroutines. The algorithm is used by ATHLET via the corresponding NUT interface.

Figure 6.2 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 that column vector lengths of the compressed Jacobian form are gradually decreasing. Figure 6.3 provides a detailed and clear view in the problem, using data from Figure 6.2 as an example, where bars represent the corresponding column lengths.

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, Figure 6.3 determines the communication pattern during the Jacobian matrix transfer for the example shown in Figure 6.2.

Code Listings 6.1 and 6.2 represent the default implementation of a compressed Jacobian matrix transfer between ATHLET and NUT. This code is used as a baseline for the remaining part of the study.

All **code listings**, presented in this part of the study, are written in **pseudocode** and intended for convenience of reading. The aim is to show and display the main ideas skipping non-relevant parts of the actual source code. The **pseudocode** is a mixture of

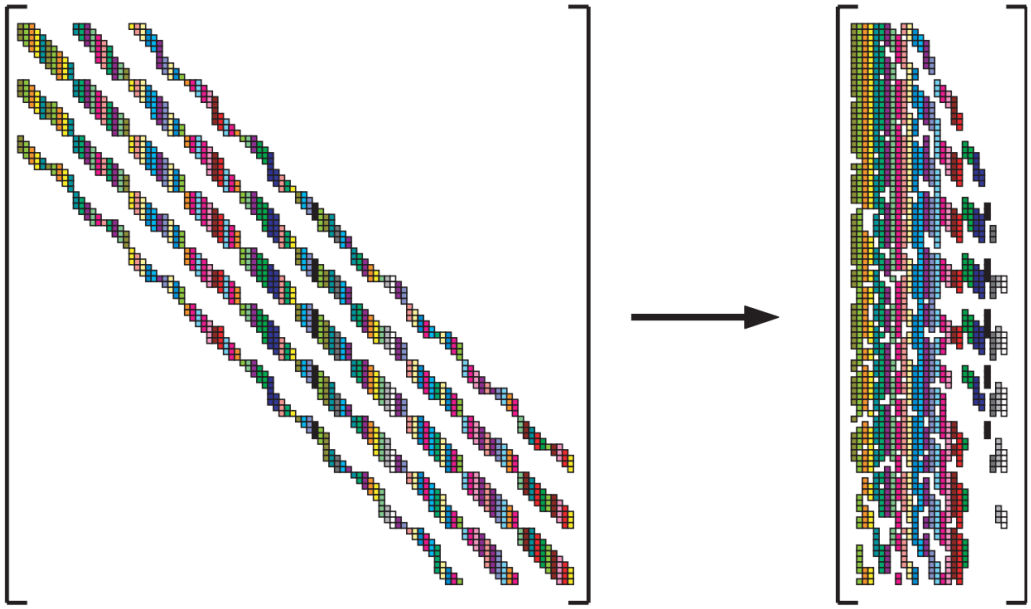


Figure 6.2.: An example of an efficient Jacobian matrix partitioning, [19]

several programming languages, namely: **Python**, **C/C++**, **Fortran**, (**MPI**).

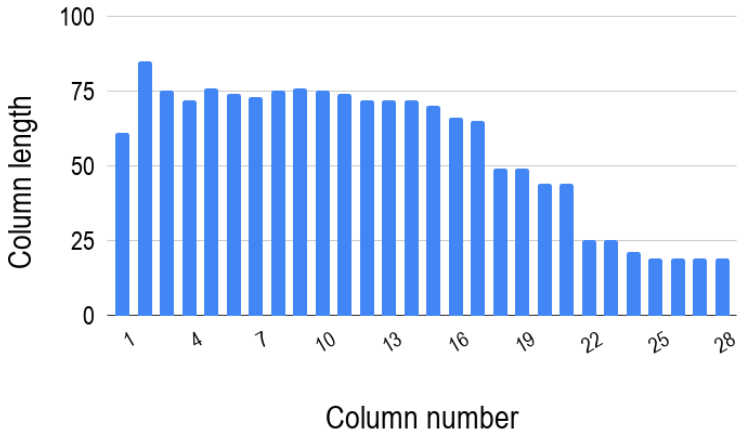


Figure 6.3.: A column-length distribution of the example depicted in Figure 6.2

```
# 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 6.1: Pseudocode of the original 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 6.2: Pseudocode of the original ATHLET-NUT coupling: NUT part

6.2. Accumulator Concept

A simple concept, named *accumulator*, has been proposed to improve MPI communication during Jacobian matrix transfers preserving the current ATHLET-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 the compressed coordinate sparse matrix format, till the critical array length equaled to $L = F \cdot N$; where N is the 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 6.4. Having swapped the pointers and reset control variables, the accumulation process can be immediately resumed together with an immediate instantiation of the corresponding non-blocking broadcast operation with respect to *send buffer* content.

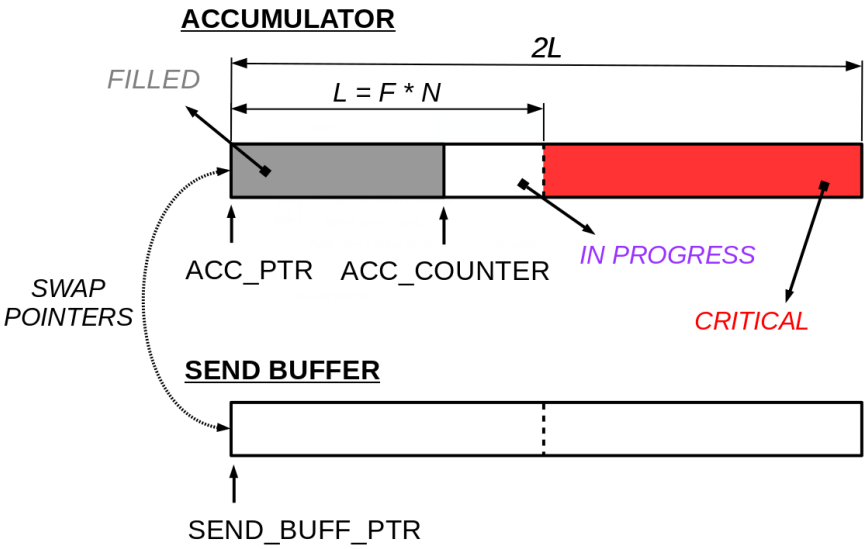


Figure 6.4.: *Accumulator* concept

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. Additionally, this event triggers a signal for a regular pointer swap and, therefore, the subsequent non-blocking data transfer.

The factor F, depicted in Figure 6.4, can be used by the user for two purposes. Mainly, it allows the user to adjust *send buffer* length L till the point of saturation of physical interconnection bandwidth, see Figure 6.5 as an example, and thus achieve efficient resource utilization. Additionally, it reduces an amount of handshakes, i.e. an amount of resource acquisition requests, between NUT and a client. 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.

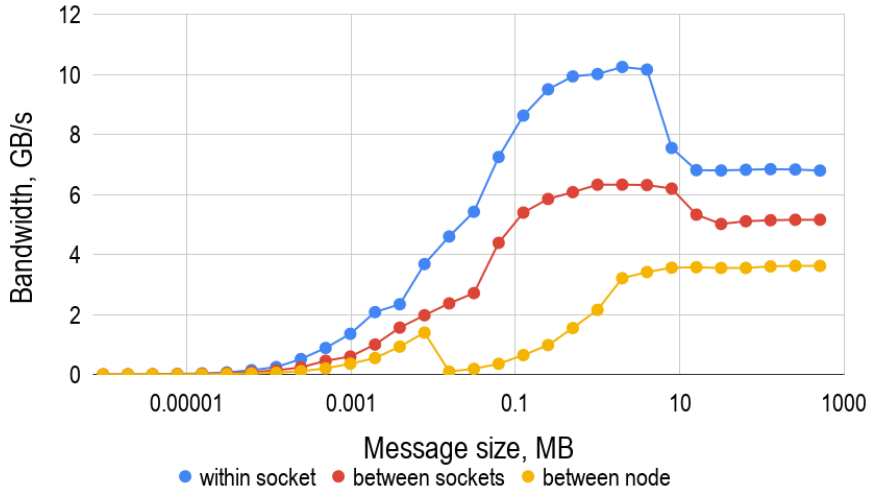
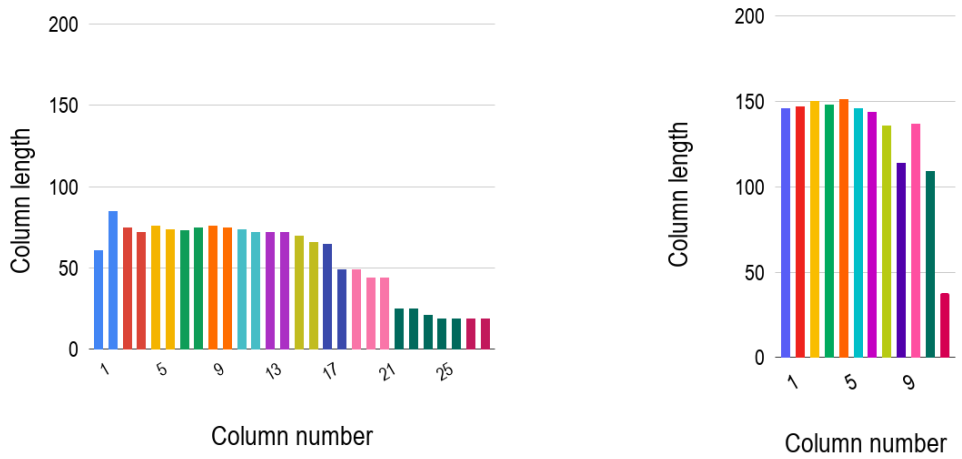


Figure 6.5.: Technical characteristics of HW1 hardware interconnection

Figure 6.6 depicts an application of the *accumulator* algorithm to the example represented in Figure 6.3 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 an original distribution shape to a more or less rectangular one which, in turn, allows to transfer a matrix in approximately equal chunks.

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 MPI operation. 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.



(a) Before

(b) After

Figure 6.6.: An application of the *accumulator* concept to the example depicted in Figure 6.3, with $N = 100$ and $F = 1$

6.3. Benchmark and Test Data

ATHLET is a dedicated industrial CFD package meant for simulation of thermal-hydraulic circuits in various nuclear power plant facilities. Besides the main part, i.e. 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 original ATHLET implementation and the new data transfer concept. It focuses only on compressed Jacobian matrix transfers and, therefore, does not include any compute-expensive operations such as non-linear function perturbations with seed vectors. The approach allowed to sufficiently speed up time of development, comparison and testing which, in turn, helped in designing the final concept described in Section 6.2.

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 lengths corresponding to that in the recordings, filled with random numbers. Figure 6.7 shows an example of a part of *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, described in Section 2.1.

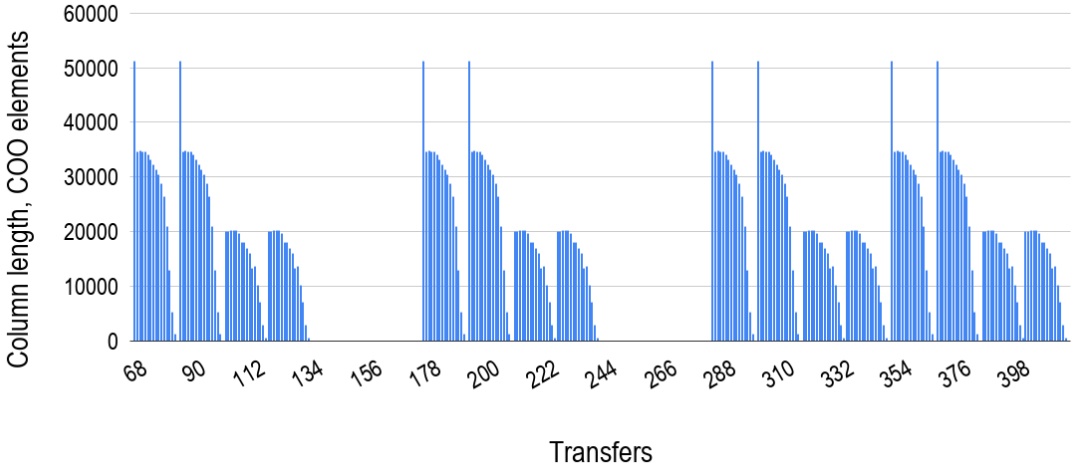


Figure 6.7.: A part of *cube-64* communication pattern

According to the *accumulator* concept, described in Section 6.2, 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 6.3 represents an additional auxiliary class used for data accumulation. Pseudocode of the benchmark client side is in Listing 6.4.

```
# 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 6.3: Pseudocode of an auxiliary *Accumulator* 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 6.4: Pseudocode of a modified client side of the benchmark

6.4. Results

The benchmark was ran on HW1 compute-cluster 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 cluster are connected via an *infiniband* interconnect with the characteristics shown in Figure 6.5. In order to estimate an effect of pure data accumulation, the benchmark, Listing 6.3, 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

non-blocking data transfers and pure data accumulation.

Figure 6.8 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. Factor F was equal to 1.

Figure 6.8a shows that *accumulator* approach results in more than **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 reduces the run-time by almost **9%** by means of more efficient utilization of intra-node interconnection. The obtained results also demonstrate that overall accumulative effect of both accumulation and non-blocking data transfers reduces the run-time of *BM1* benchmark in more than **26%**. Table 6.1 summarizes results obtained for all three client-server distributions within the same range of the recorded communication pattern displayed in Figure 6.7.

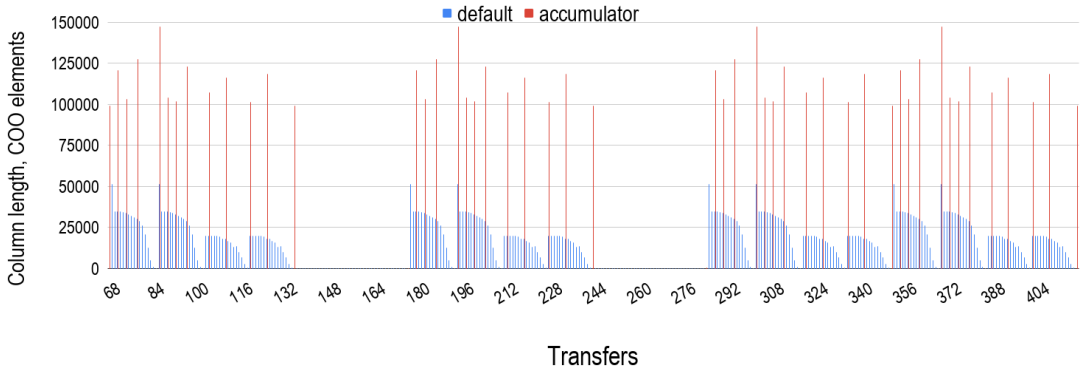
Benchmark name	BM2, %	BM1, %
within a socket	7.61	13.84
between sockets	9.04	26.26
between nodes	-2.06	3.20

Table 6.1.: Time reduction of data transfers with respect to the original implementation in case of execution of *cube-64* communication pattern

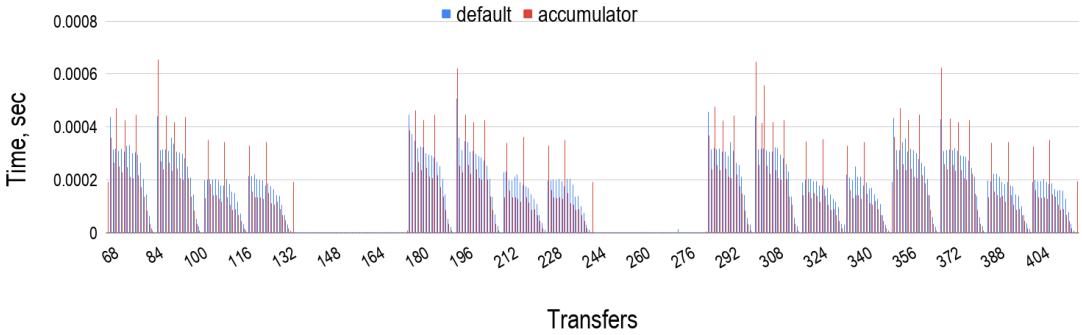
It turns out that *BM2* benchmark is slower than the original ATHLET approach in approximately **2%** in case of inter-node communication. However, non-blocking data broadcasts, according to *BM1* benchmark, help to alleviate the slow-down and achieve almost **3%** of improvement.

Unimpressive results of non-blocking 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 transfers in case of *cube-64* test-case, see Figure 6.9. Thus, the execution control was probably suspended by MPI library at each subsequent call of *MPI_Wait()* function.

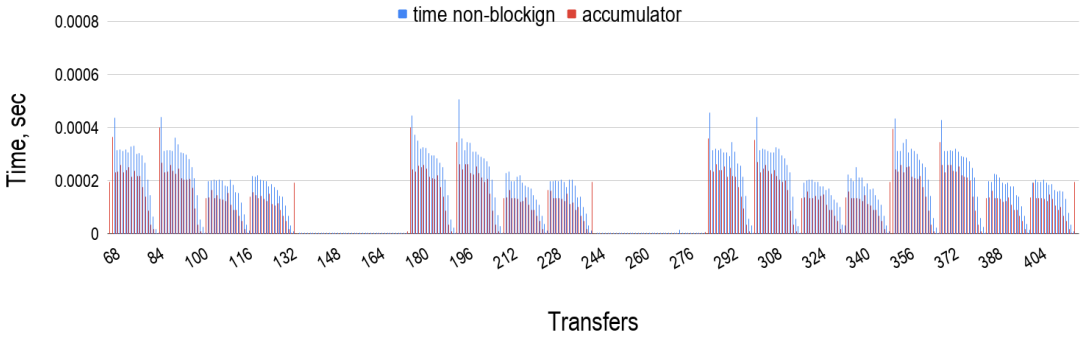
The slow-down resulting from pure data accumulation could be explained by automatic MPI protocol switching, namely: *Eager* and *Rendezvous* [34]. The protocols are



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



(b) BM2: A comparison of the data accumulation concept using blocking communication with the original approach



(c) BM1: A comparison of the data accumulation concept using non-blocking communication with the original approach

Figure 6.8.: Comparisons of the benchmarks running a recorded part of *cube-64* communication pattern between two sockets of a node

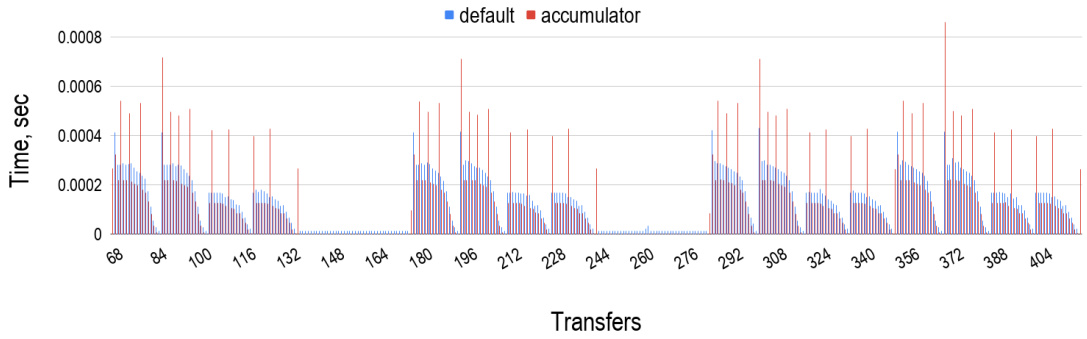


Figure 6.9.: A comparison of *BM1* benchmark with the original ATHLET-NUT implementation running a recorded part of *cube-64* communication pattern between two compute-nodes

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

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

Ideas, expressed in *BM1* benchmark, Listings 6.3 and 6.4, were successfully implemented in NUT, namely: in 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 *pwr-3d*. Verification of the modified code did not detect any deviations of numerical results from the original implementation. Additionally, all tests showed considerable improvements in 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 allocations, respectively, for *pwr-3d* scenario, taking it as the most representative simulation test-scenario known in GRS. However, the overall improvement of applied changes achieved only **0.14%** on average, regardless of a client-server allocation. Profiling showed the communication part of the original

implementation took around **0.24%** of the total time spent on matrix evaluations and transfers. This fact explains this negligible overall performance gain, resulted from the source code modification, that was observed in all conducted final tests.

6.5. Conclusion

In this part of the study, we have designed and implemented the *accumulator* concept for efficient transfers of sparse compressed Jacobian matrices 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 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, a reduced amount of NUT process synchronizations
3. overlaps of communications with computations

The study has shown that non-blocking data transfers are 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 slight slow-down from pure data accumulation in case of inter-node communication due to probable MPI protocol switching. However, as it has been shown, it is always compensated by means of communication/computation overlaps.

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 may be much better in case of multi-client operation of NUT, especially when clients are sharing common NUT processes; a reduced number of data transfers results in a reduced amount of handshakes which are always accompanied by the resource acquisition mechanism, described in Section 2.3. Unfortunately, it is difficult to design and prepare a set of valid tests to verify this statement.

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

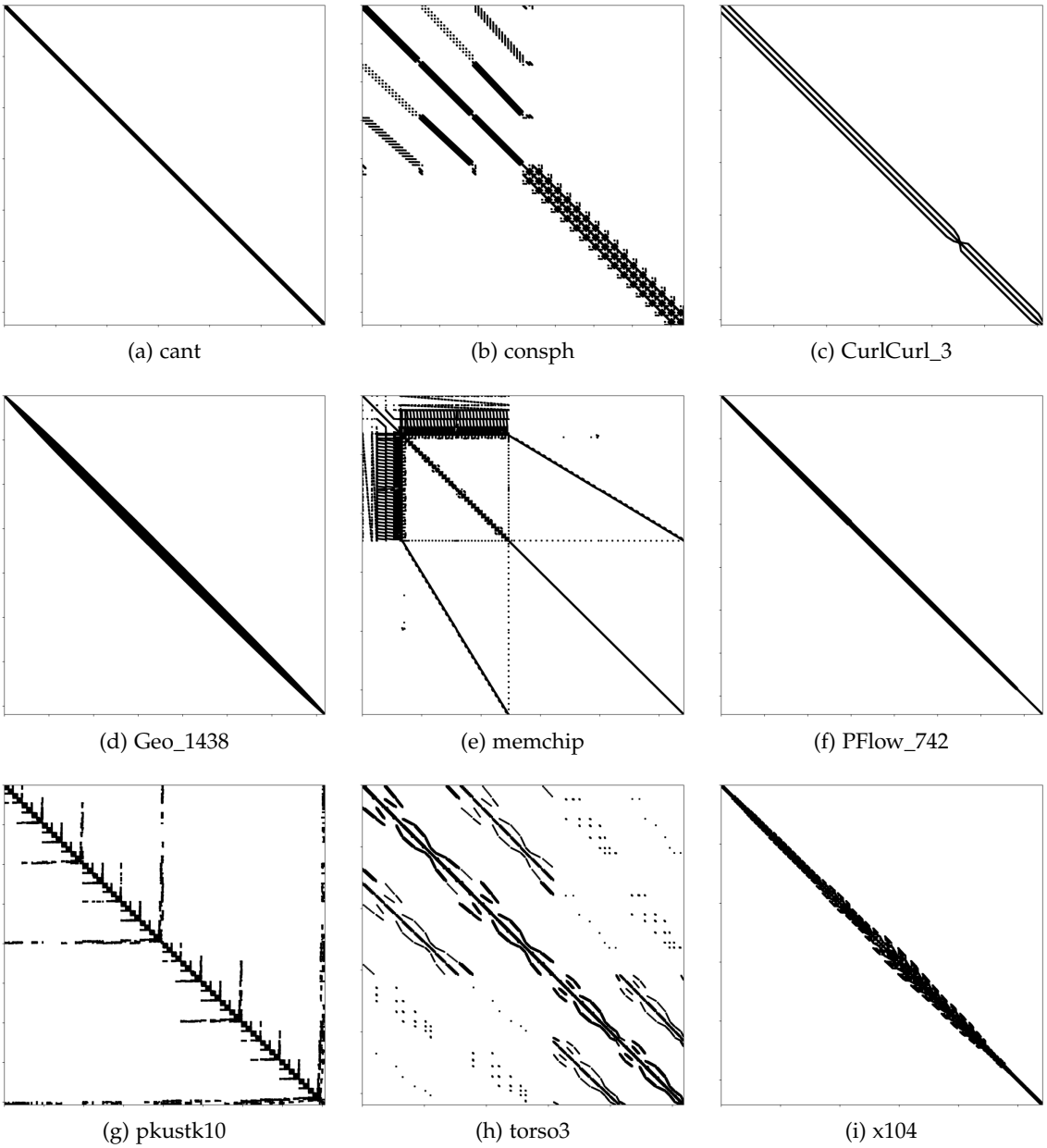
gain. The study has also shown the main bottleneck is, indeed, the non-linear function evaluation.

It is worth mentioning that only the sequential ATHLET code, capable to run only in a single core, was 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 a reduced fraction of execution time spent on non-linear function evaluations. This fact also shows that 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

Sparsity patterns of the GRS matrix set are not available for any online publication. Please, contact GRS representatives to get access to the data.



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

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

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

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

B. Selection of a Sparse Direct Linear Solver

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.: Comparisons of parallel performance of *pwr-3d* matrix factorizations using MUMPS, PasTiX and SuperLU_DIST libraries with their default parameter settings

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

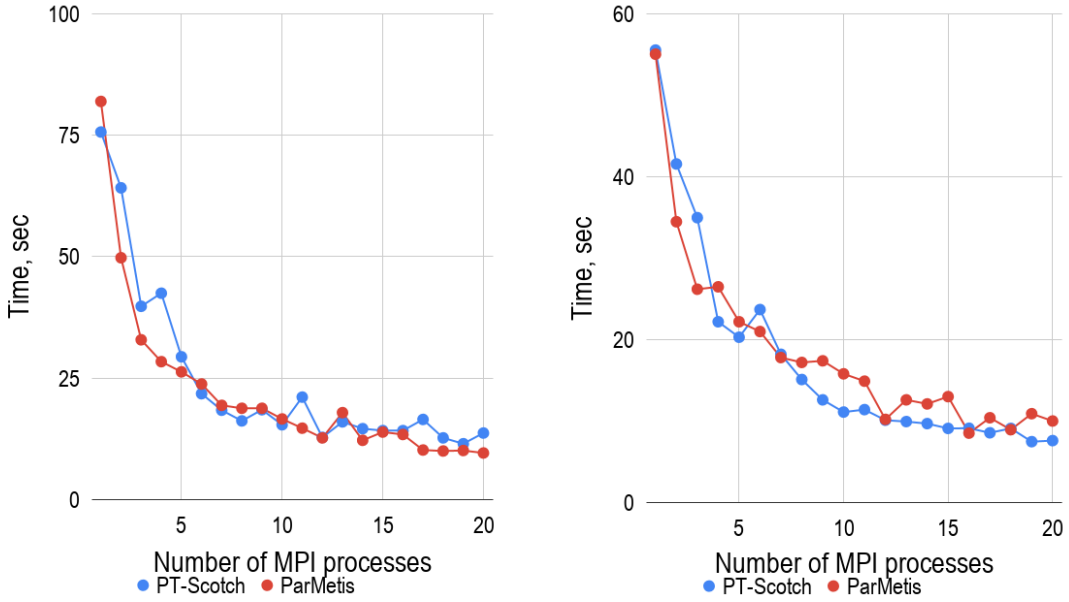
Table B.2.: Comparisons of parallel performance of *k3-2* matrix factorizations using MUMPS, PasTiX and SuperLU_DIST libraries with their default parameter settings

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

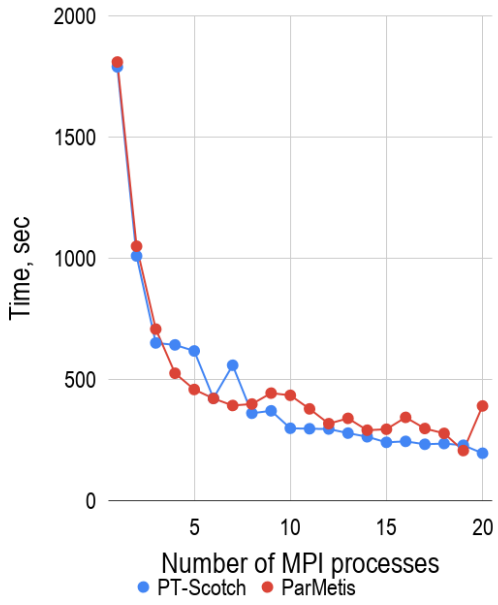
Table B.3.: Comparisons of parallel performance of *cube-645* matrix factorizations using MUMPS, PasTiX and SuperLU_DIST libraries with their default parameter settings

C. Fill Reducing Reorderings

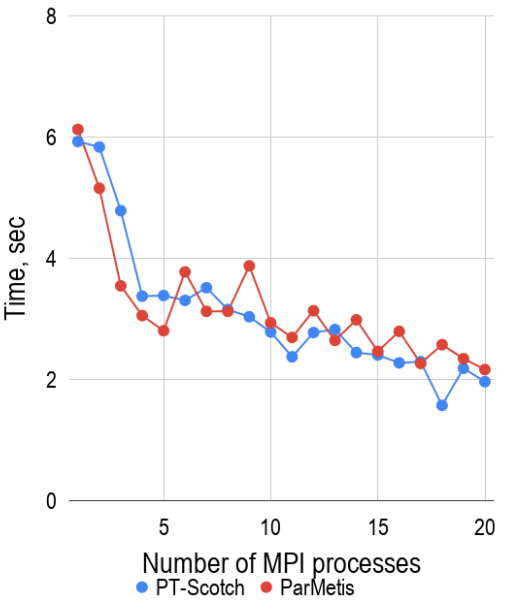


(a) torso3

(b) consph

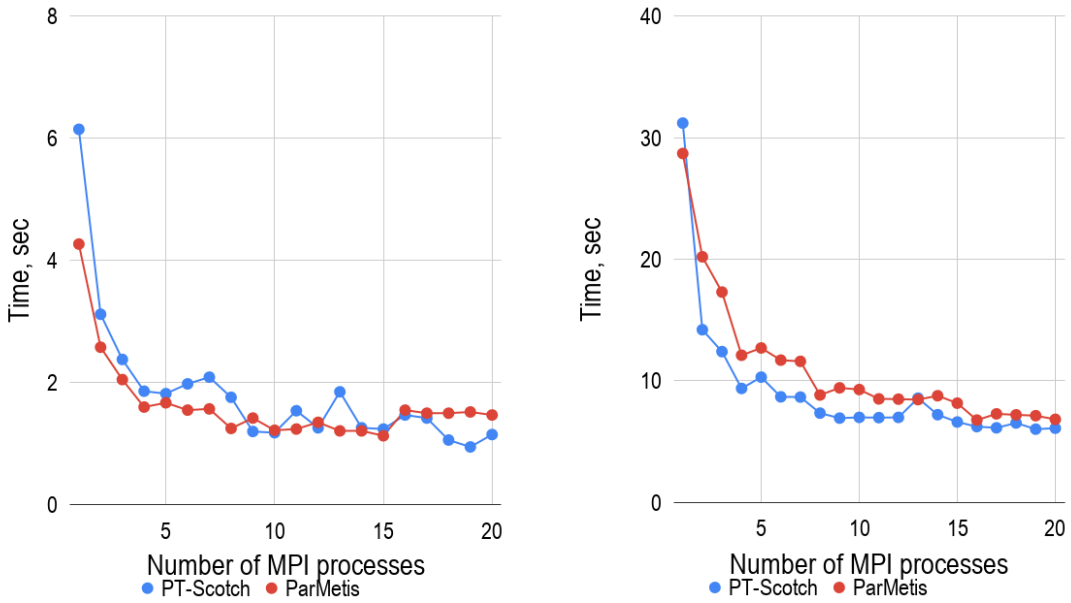


(c) CurlCurl_3



(d) x104

Figure C.1.: An influence of different fill reducing algorithms on parallel factorizations of *torso3*, *consph*, *CurlCurl_3* and *x104* matrices

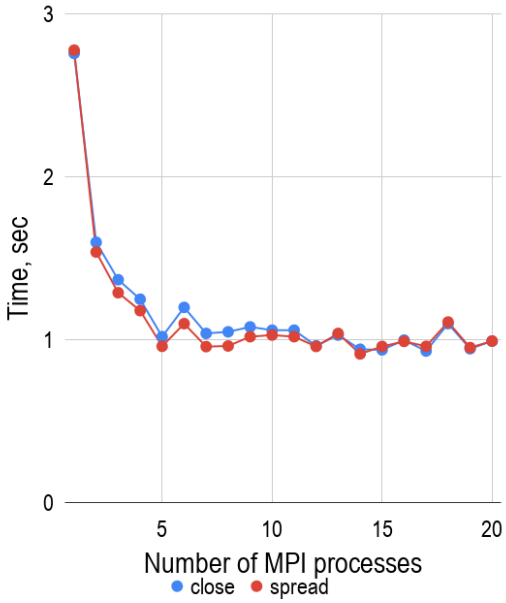


(a) cant

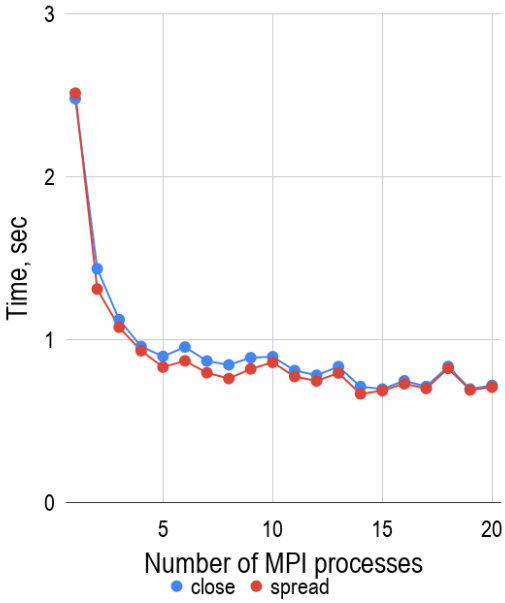
(b) memchip

Figure C.2.: An influence of different fill reducing algorithms on parallel factorizations of *cant* and *memchip* matrices

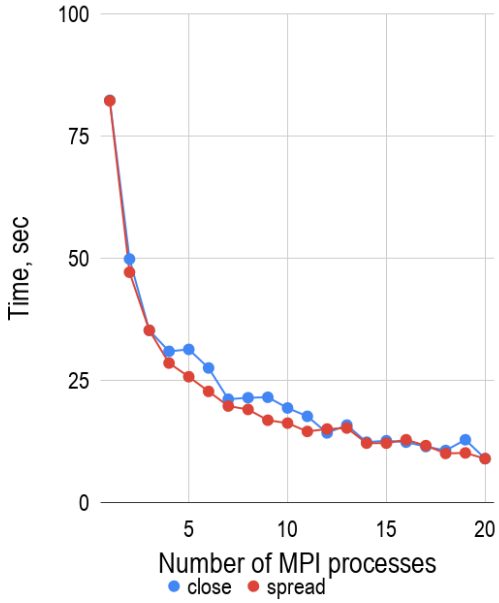
D. MPI Process Pinning



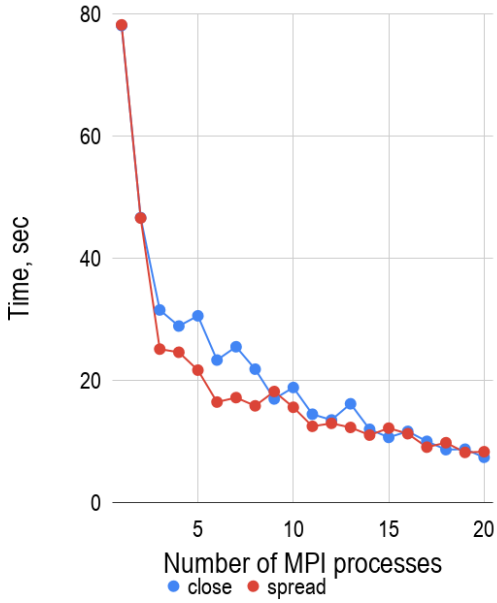
(a) HW1 - k3-2



(b) HW2 - k3-2

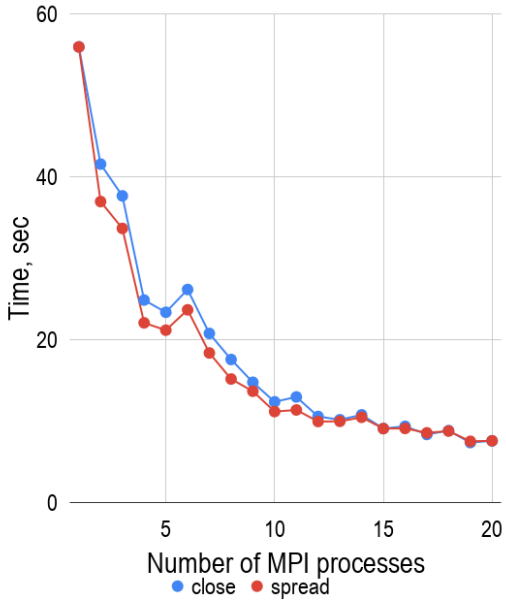


(c) HW1 - torso3

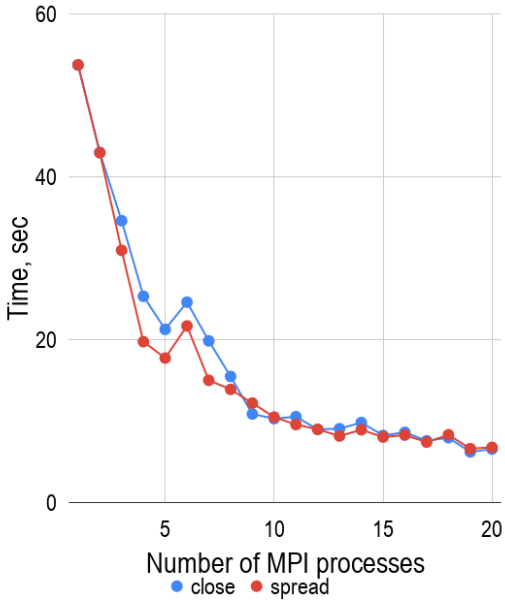


(d) HW2 - torso3

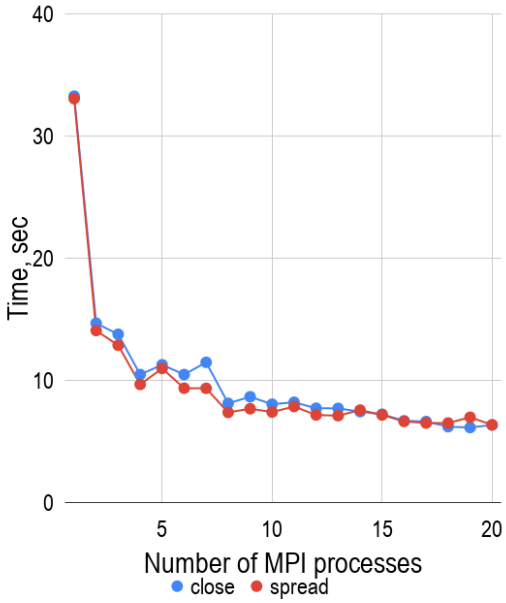
Figure D.1.: Comparisons of *close* and *spread* pinning strategies applied to parallel factorizations of *cube-64* and *torso3* matrices



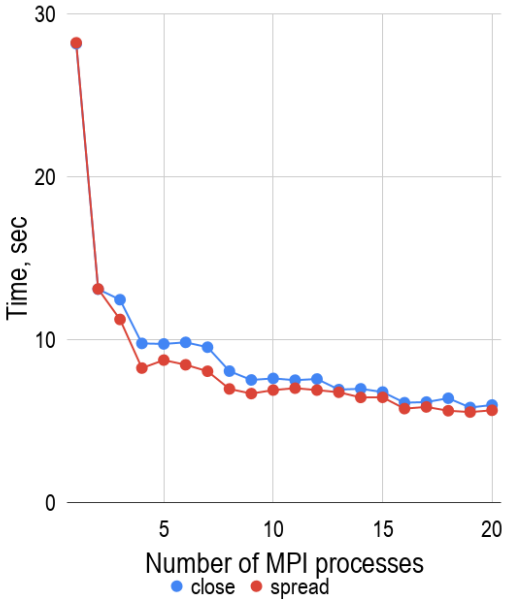
(a) HW1 - consph



(b) HW2 - consph



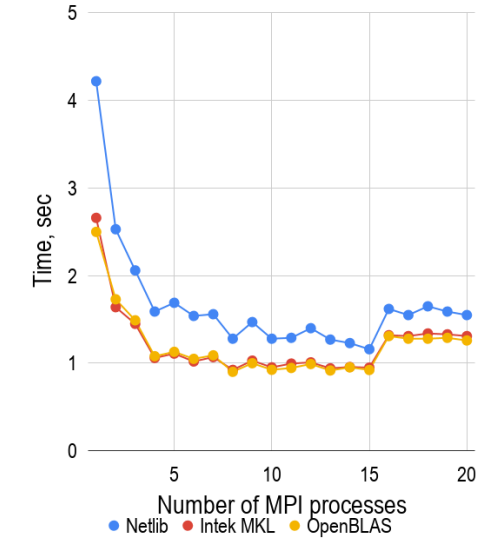
(c) HW1 - memchip_3



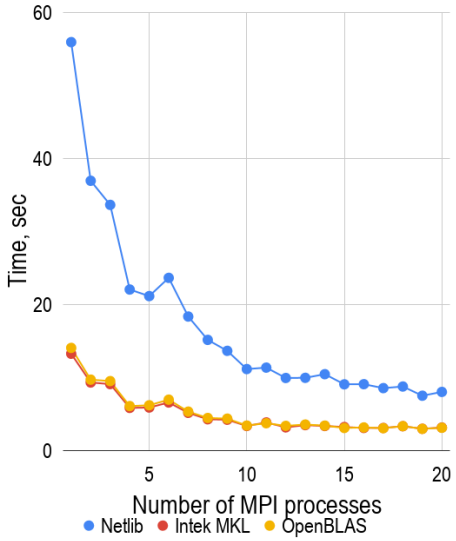
(d) HW2 - memchip

Figure D.2.: Comparisons of *close* and *spread* pinning strategies applied to parallel factorizations of *consph* and *memchip* matrices

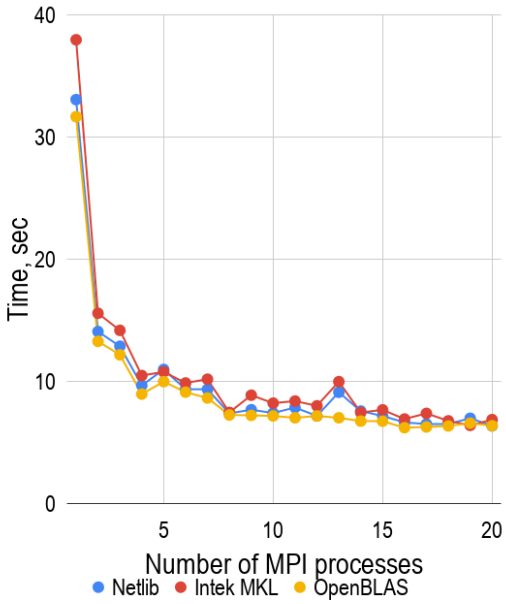
E. Optimized BLAS Libraries



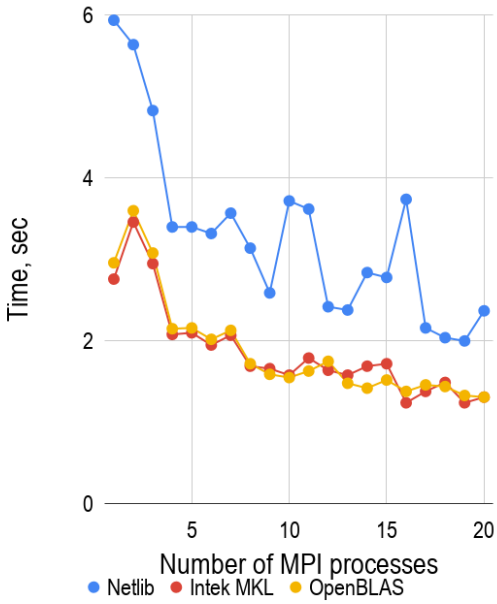
(a) cant



(b) consph

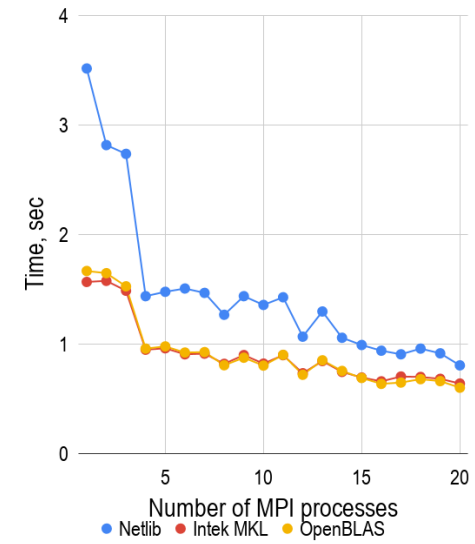


(c) memchip

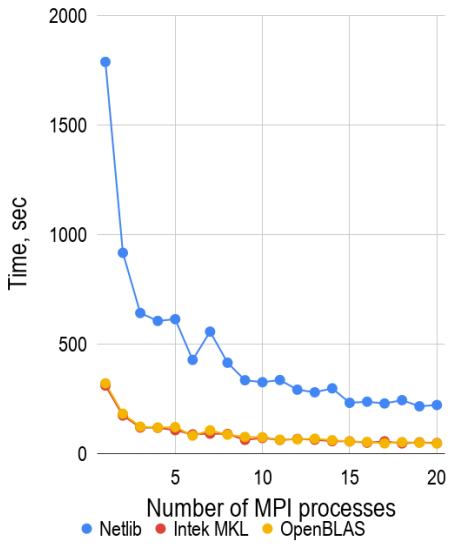


(d) x104

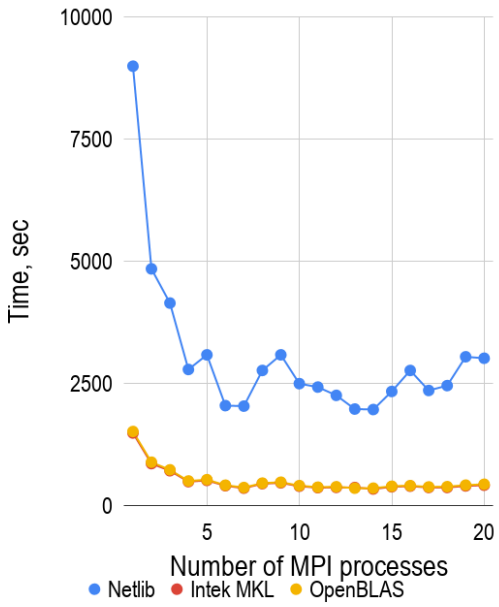
Figure E.1.: Comparisons of parallel factorizations of *cant*, *consph*, *memchip* and *x104* matrices performed on HW1 machine using MUMPS solver linked to different BLAS implementations



(a) pkustk10



(b) CurlCurl_3



(c) Geo_1438

Figure E.2.: Comparisons of parallel factorizations of *pkustk10*, *CurlCurl_3* and *Geo_1438* matrices performed on HW1 machine using MUMPS solver linked to different BLAS implementations

Bibliography

[1] 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.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. "An approximate minimum degree ordering algorithm". In: *SIAM Journal on Matrix Analysis and Applications* 17.4 (1996), pp. 886–905.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. "MUMPS: A multifrontal massively parallel solver". In: *ERCIM News* 50 (2001), pp. 14–15.

[4] 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.

[5] P. Arbenz. *Lecture notes of Numerical Methods for Solving Large Scale Eigenvalue Problems: Arnoldi and Lanczos algorithms*. 2018. URL: `http://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter10.pdf`.

[6] M. Arioli, J. W. Demmel, and I. S. Duff. "Solving sparse linear systems with sparse backward error". In: *SIAM Journal on Matrix Analysis and Applications* 10.2 (1989), pp. 165–190.

[7] H. Austregesilo, C. Bals, A. Hora, G. Lerchl, P. Romstedt, P. Schöffel, D. Von der Cron, and F. Weyermann. *ATHLET Mod 3.1A – Models and Methods*. distributed with ATHLET. Mar. 2016.

[8] 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.

[9] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, et al. *PETSc Users Manual: Revision 3.10*. Tech. rep. Argonne National Lab.(ANL), Argonne, IL (United States), 2018.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] I. S. Duff and J. K. Reid. "The multifrontal solution of indefinite sparse symmetric linear". In: *ACM Transactions on Mathematical Software (TOMS)* 9.3 (1983), pp. 302–325.

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

[18] R. Falgout, A. Cleary, J. Jones, E. Chow, V. Henson, C. Baldwin, P. Brown, P. Vassilevski, and U. Yang. "Hypre User's Manual, Version 2.7". In: *Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory, Livermore, CA* (2010).

[19] 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.

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

[21] 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.

[22] Gesellschaft für Anlagenund Reaktorsicherheit (GRS) gGmbH. *Development and Validation of Simulation Codes*. URL: https://www.grs.de/en/content/development-and-validation-simulation-codes.

[23] Gesellschaft für Anlagenund Reaktorsicherheit (GRS) gGmbH. *GRS – Safety for man and the environment*. URL: https://www.grs.de/en/company.

[24] 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.

[25] A. Gupta, S. Koric, and T. George. "Sparse matrix factorization on massively parallel computers". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM. 2009, p. 1.

[26] *Intel Math Kernel Library Documentation, Overview of ScaLAPACK Routines*. 2017. URL: https://software.intel.com/en-us/mkl-developer-reference-c-overview-of-scalapack-routines.

[27] B. M. Irons. "A frontal solution program for finite element analysis". In: *International Journal for Numerical Methods in Engineering* 2.1 (1970), pp. 5–32.

[28] 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.

[29] 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.

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

[31] J. Kwack, G. Bauer, and S. Koric. "Performance test of parallel linear equation solvers on Blue Waters–Cray XE6/XK7 system". In: *Preceedings of the Cray Users Group Meeting (CUG2016), London, England*. 2016.

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

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

[34] Lawrence Livermore National Laboratory. *MPI Performance Topics: MPI Message Passing Protocols*. 2014. URL: https://computing.llnl.gov/tutorials/mpi_performance/.

[35] X. Li, J. Demmel, J. Gilbert, iL. Grigori, M. Shao, and I. Yamazaki. *SuperLU Users' Guide*. Tech. rep. LBNL-44289. `http://crd.lbl.gov/~xiaoye/SuperLU/`. Last update: August 2011. Lawrence Berkeley National Laboratory, Sept. 1999.

[36] J. W. Liu. "The multifrontal method for sparse matrix solution: Theory and practice". In: *SIAM review* 34.1 (1992), pp. 82–109.

[37] *Multifrontal massively parallel solver (MUMPS 5.1.2) users' guide*. 2017. URL: `http://mumps.enseeiht.fr/doc/userguide_5.1.2.pdf`.

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

[39] *PaStiX User's manual*. 2013. URL: `https://gforge.inria.fr/docman/view.php/186/5707/pastixman.pdf`.

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

[41] A. Pothen and S. Toledo. *Elimination Structures in Scientific Computing*. 2004.

[42] 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* 7.3 (1986), pp. 856–869.

[43] 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.

[44] T. Steinhoff. "Singly implicit FiterRK methods for thermal-hydraulic simulations". ANODE. 2018.

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

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

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

[48] C. J. Wu. "Partial Left-Looking Structured Multifrontal Factorization & Algorithms for Compressed Sensing". PhD thesis. UC Berkeley, 2012.

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