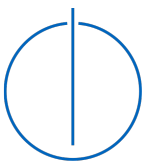
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Computational Science and Engineering

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

Ravil Dorozhinskii



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, produces electricity. According to the recent estimations, thermal efficiency of modern nuclear power plants lies in the range of 35-45% which is comparable to conventional fossil fueled power plants [**intro:efficiency-of-nuclear-power-plants**]. In spite of considerable initial investment, nuclear power plants have low operating costs and longevity which makes them particularly cost effective.

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

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 [**intro:eu-nuclear-industry-general**].

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

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 [**grs:grs-general-info**]. 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 [**grs:grs-general-info**]. Due to the inability to create various nuclear accident test scenarios, which by their very nature could be catastrophic, GRS develops and provides numerous simulation software products to cope with this problem. A short description of the main software packages developed by GRS is provided in table 1.1.

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

Table 1.1.: A general overview of software developed by GRS [**grs:grs-general-info**]

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

2. Overview of ATHLET and NuT software

2.1. ATHLET

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

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

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

6. Vapor energy

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

7. Volume vapor fraction

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

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

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

place for figure 2.1

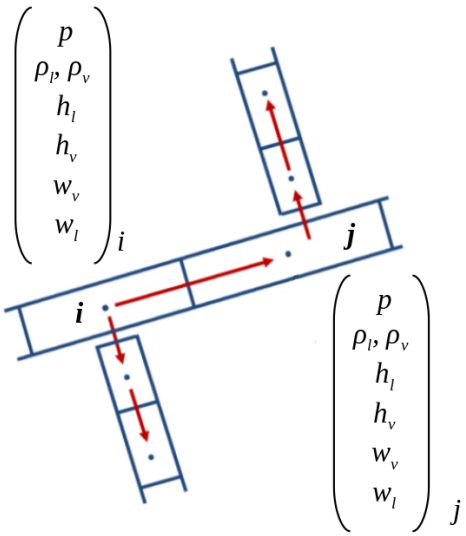


Figure 2.1.: ATHLET: one dimensional finite volume formulation of the problem

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

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

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

Analysis of system 2.8 shows the problem is rather stiff and thus must to be solved with an implicit solver. Rosenbrock methods are a class of linear implicit methods which is capable of solving such stiff systems of ODEs efficiently. The methods replace non-linear systems with a sequence of linear ones, however, some stability and accuracy properties are usually lost [**blom2013rosenbrock**]. An additional drawback of the methods is evaluation of the exact Jacobian at every time step which affects computational cost.

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

place for figure 2.2

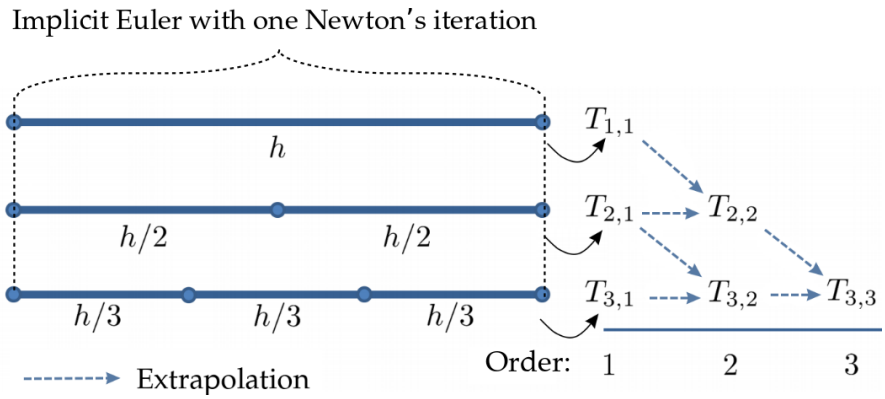


Figure 2.2.: A general view on the W-method solver implemented in ATHLET

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

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

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

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

2.2. NuT

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

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

2.3. ATHLET-NuT coupling

place for figure 2.3

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

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

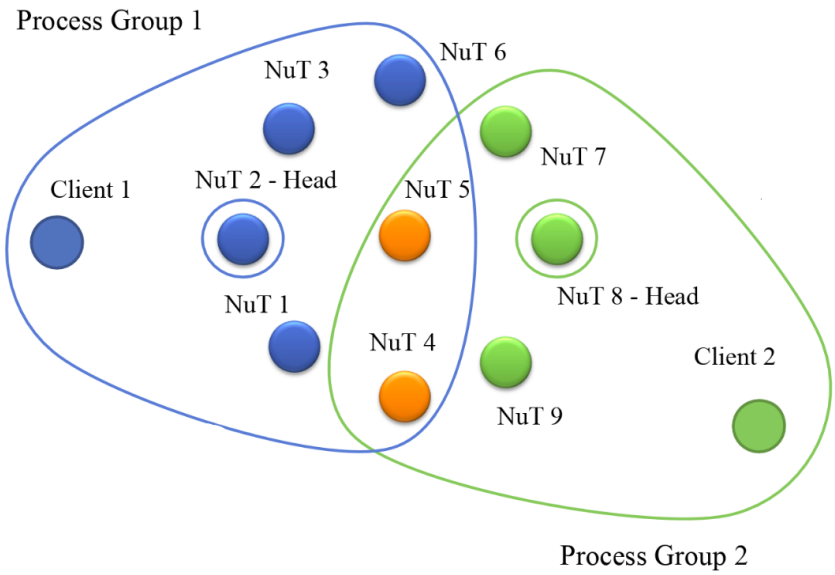


Figure 2.3.: NuT process groups

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

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

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

communication between client and server is synchronous i.e. the client gets blocked while waiting for a result from the server.

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

ask for an updated image

place for figure 2.4

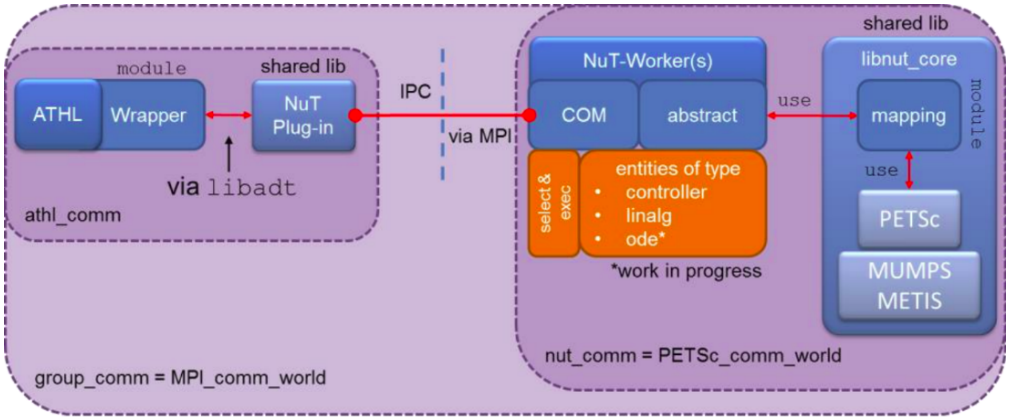


Figure 2.4.: ATHEL-NuT software coupling

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

3. Problem Statement

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

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

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

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

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

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

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

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

- open-source license
- direct interface to PETSc

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

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

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

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

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

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

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

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

4. Experimental Setup and Matrix Sets

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

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

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

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 *(SYMM - symmetric; NON-SYMM - non-symmetric; SYMM-PTRN- non-symmetric but with symmetric sparsity pattern)*

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

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 *(SYMM - symmetric; NON-SYMM - non-symmetric; SYMM-PTRN- non-symmetric but with symmetric sparsity pattern)*

The objective of this study is to find and configure a sparse linear solver which can fulfill all requirements listed above for the GRS matrix set. It is worth pointing out, as it was mentioned in section 2.1, ATHLET performs many mathematical transformations upon the original system and, finally, generates an approximation of a Jacobian matrix. For that reason, one can assume that GRS matrix set can be structurally different from matrices coming naturally from finite-volume, finite-elements discretization or optimization problems. Therefore, SuiteSparse matrix set was used, from time to time, to examine this statement.

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

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

To make process pinning explicit and deterministic, a python script was developed to generate rank-files automatically based on the number of MPI processes, OpenMP threads per MPI process, the maximum number of processing elements and the number

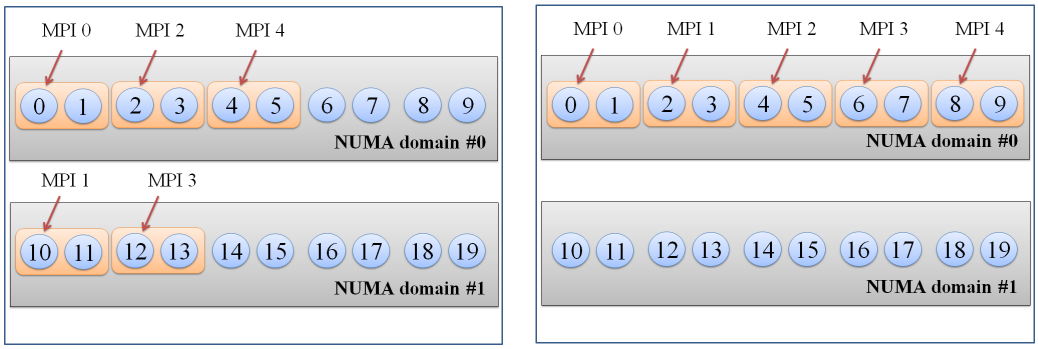
	HW1 (GRS)	HW2 (LRZ Linux)
Architecture	x86_64	x86_64
CPU(s)	20	28
On-line CPU(s) list	0-19	0-27
Thread(s) per core	1	1
Core(s) per socket	10	14
Socket(s)	2	2
NUMA node(s)	2	4
Model	62	63
Model name	E5-2680 v2	E5-2697 v3
Stepping	4	2
CPU MHz	1200.0	2036.707
Virtualization	VT-x	VT-x
L1d cache	32K	32K
L1i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

Table 4.3.: Hardware specification

of NUMA domains. The scrip always leaves appropriate gaps between MPI processes to allow each process to fork the corresponding number of threads within a parallel region.

A rank-file specifies explicit mapping between MPI processes (ranks) and actual processing elements, cores, within a machine. The script has two modes, namely: *spread* and *close*. Given a certain number of ranks, *spread* mode tries to distribute them as spread as possible across multiple available NUMA domains in a round-robin fashion. In contrast to *spread* strategy, *close* one groups ranks as close as possible to keep the maximum number of ranks within a single NUMA domain. Figure 4.1 shows an example of how these two modes work in case of 5 MPI ranks, 2 OpenMP threads per rank, on a compute node equipped with 20 cores and 2 NUMA domains (HW1).

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



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

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

5. Configuration of a sparse linear solver

5.1. Overview of Sparse Linear Solver Types

5.1.1. Iterative Methods

5.1.1.1. Theory Overview

Given an initial guess, iterative methods generate a sequence of successively improving approximated solutions. The 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 filed of scientific computing because of it is rather fast convergence in case of solving of well conditioned systems.

The most well-known methods among the Krylov family are Conjugate Gradient (CG) in case of symmetric positive definite matrices, Minimal Residual Method (MINRES) for symmetric indefinite systems and Generalized Minimal Residual Method (GMRES) for non-symmetric systems of linear equations. There also exist different variants of GMRES 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, $A^3b, \dots A^nb$. 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. Essentially, the methods convert solution of a system of a linear equations to a minimization problem and search for a solution in the corresponding subspace.

For example, GMRES aims to minimize the Euclidean norm of the residual r_m of a solution vector x_m in mth Krylov subspace $\mathcal{K}_m$. However, the basis vectors of Krylov

subspace $\mathcal{K}_m$ are usually close to linearly dependent and, thus, solution vector x_m is constructed in an orthonormal basis U_m which forms the same subspace $\mathcal{K}_m$.

$$r_m = \min_{x \in U_m} ||Ax_m - b||^2 = \min_{x \in U_m} ||AU_m y_m - b||^2 \tag{5.1}$$

where the vector x can be written in the basis U_m as:

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

In the general case, orthogonalization of the basis can be performed in different ways. Saad and Schultz, in work **[sparse-la:gmrese-origin]**, proposed to use the Arnoldi process [REF] for constructing an l_2-orthogonal basis. As a result, equation 5.1 can be written in a form:

reference

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

where H_m is an upper Hessenberg matrix i.e. the matrix with zeros below the first sub-diagonal. Application of the Givens rotation algorithm [REF] results in computing of the corresponding QR decomposition of Hessenberg matrix H_m. After the substitution of matrix H_m with the decomposition and some mathematical transformations, equation 5.3 can be written as follows:

reference

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

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 orthonormal basis U_m, equation5.2, solution vector x_m can be written as:

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

The solution significantly improves with growth of subspace $\mathcal{K}_m$. This, in turn, leads to a considerable increase of computational cost and storage space, as a result. Therefore, the algorithm usually runs till 20 - 50 column vector evaluations of the

Krylov subspace and restarts using the 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. Application of these linear algebra kernels allows 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 parallelism. Hence, the main drop of parallel performance mainly comes from process-communication overheads.

However, some methods can have sequential parts that may effect 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 criteria of Krylov methods is convergence. The convergence of iterative methods strongly depends on a 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 CG method. It can be clearly observed that a big condition number 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 in order to reduce its condition number. As a result, the solution process of the modified system is significantly 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 has to result in *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 execution as well.

There exist numerous techniques to compute a preconditioner P for a 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. Practice has been shown that some techniques can work particularly well for matrices derived from a certain 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 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 user's manuals, [**balay2018petsc**] and [**falgout2010hypre**], respectively.

It is worth mentioning 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 usage of a *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

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.: Parallel preconditioning algorithms available in PETSc

can grow significantly.

```
1 -pc_bjacobi_blocks 4
2 -sub_pc_type lu
3 -pc_factor_mat_ordering_type rcm
4 -pc_factor_pivot_in_blocks true
```

Listing 5.1: Usage of a PETSc built-in sparse direct linear solver as a sub-preconditioner for a PETSc 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 $O(n^2)$ [**complexity-of-spdm**] 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.

A multi-frontal method is probably the most representative example of direct sparse solvers, inroduced by **mult-frontal-original:1** in work [**mult-frontal-original:1**]. The method is, in fact, an improved version of the frontal method [**frontal-original**] 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 multi-frontal method explained in detail in [**elimination-tree**].

In this subsection, the theory of multi-frontal 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 matrix A is real, symmetric and sparse. In this case, LU decomposition of matrix A boils down to Cholesky factorization, equation 5.10.

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

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

place for figure 5.1

Figure 5.1 shows an illustrative example of a sparse matrix A and its Cholesky factor L, taking from [**mult-frontal-original:2**]. The solid circles represent original non-zero elements whereas hollow ones define fill-in factors of L.

The elimination tree is a crucial part of the method. It can be considered as a structure of n nodes where node p is the parent of j if and only if it satisfies equation 5.11. It is worth pointing out the definition 5.11 is not only one possible and one can define a

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

Figure 5.1.: An example of a sparse matrix and its Cholesky factor **[mult-frontal-original:2]**

strucutre of the elimination tree in a different way as well, **[mult-frontal-original:2]**.

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

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.

place for figure 5.2

The fundamental idea of multi-frontal 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

.

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

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

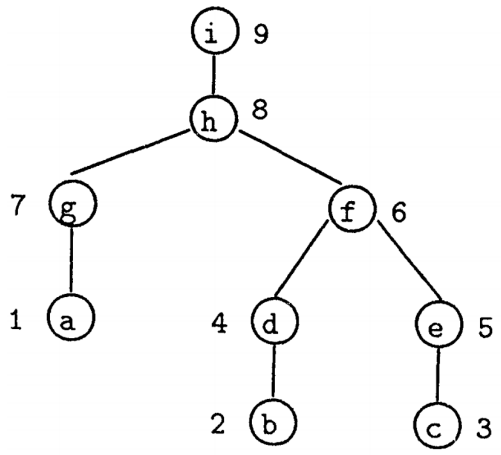


Figure 5.2.: An elimination tree of matrix A from example depicted in figure 5.1 **[mult-frontal-original:2]**

Frame matrix Fr_j is filed 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:

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

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 the non-zero pattern of the jth column.

Let's consider a 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:

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

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

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 matrix can be written in a vector form 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. At the same time, 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 a 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 in a vector form: check formula

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

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 multi-frontal method. Let's consider an example, depicted in figure 5.3, to demonstrate importance of contribution matrices.

place for figure 5.3

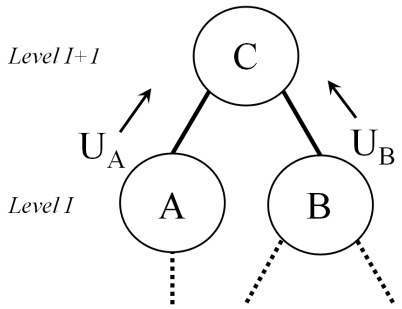


Figure 5.3.: Information flow of the multifrontal method

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 [**mult-frontal-original:2**].

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 be introduced in the theory of direct sparse methods.

As an example, taking from [**mult-frontal-original:2**], 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:

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

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

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

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

At this point, it is worth mentioning the resulting frontal matrix F_j forms a small dense block which has to be factorized along the first column. The partial factorization of such a 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 multi-frontal 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 column elimination order which can be achieved by some matrix re-ordering thecniques. For instance, *post-ordering*, mentioned by **mult-frontal-original:2** in [**mult-frontal-original:2**], can be considered as an example of such re-ordering, 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 re-ordering results in the same fill-in of the factor [**mult-frontal-original:2**].

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.

place for figure 5.4

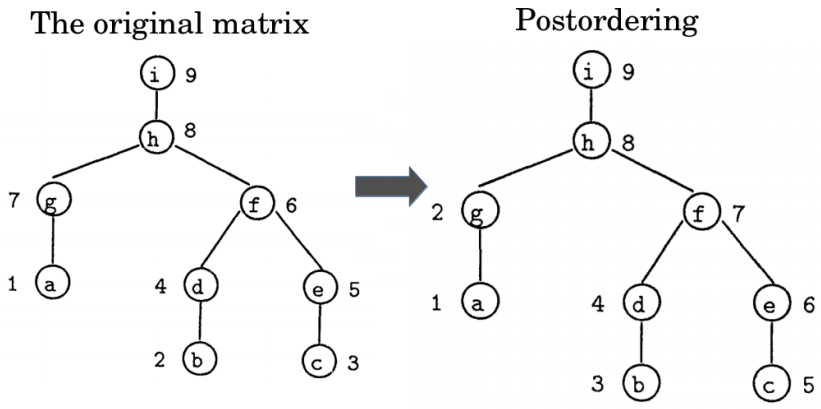


Figure 5.4.: An example of matrix postordering from [**mult-frontal-original:2**]

place for figure 5.5

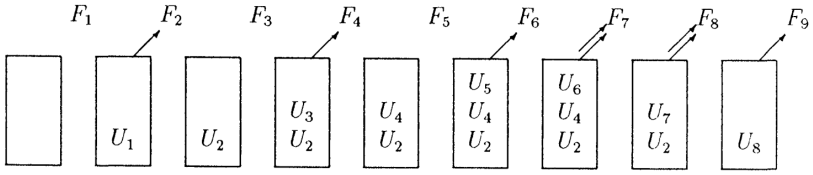


Figure 5.5.: The stack contents for the postordering [**mult-frontal-original:2**]

In practice, an improved version of multi-frontal method, called super-nodal 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 identical off-diagonal sparsity structure. Hence, a super-node has two important properties. Firstly,

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.

place for figure 5.6

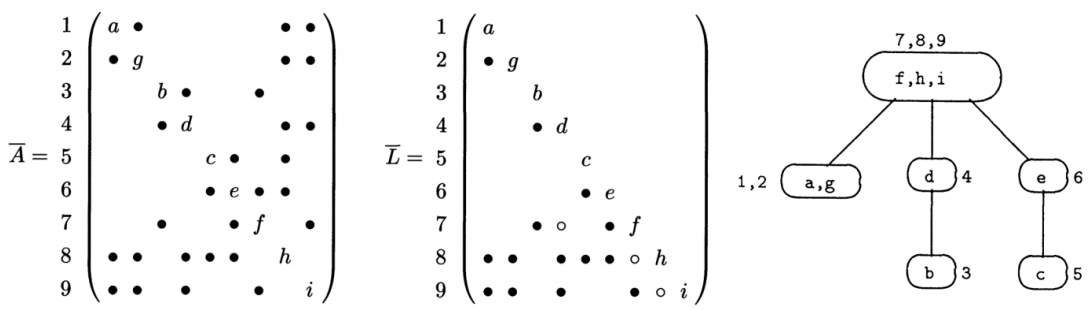


Figure 5.6.: An example of a supernodal elimination tree [**mult-frontal-original:2**]

Equation 5.23 shows an assembly process of super-nodal frontal matrix $\mathcal{F}_j$. In contrast to 5.21, the 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 children, namely: U_{c_1}, U_{c_2}, $\ldots$, U_{c_s}.

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

$$\mathcal{F}_j = \begin{bmatrix} a_{j,j} & a_{j,j+1} & \ldots & a_{j,j+t} & a_{j,i_1} & \ldots & a_{j,i_r} \\ a_{j+1,j} & a_{j+1,j+1} & \ldots & a_{j+1,j+t} & a_{j+1,i_1} & \ldots & a_{j+1,i_r} \\ \vdots & \vdots & \ldots & \vdots & & & \\ a_{j+t,j} & a_{j+t,j+1} & \ldots & a_{j+t,j+t} & a_{j+t,i_1} & \ldots & a_{j+t,i_r} \\ a_{i_1,j} & a_{i_1,j+1} & \ldots & a_{i_1,j+t} & & & \\ \vdots & \vdots & \ldots & \vdots & & 0 & \\ a_{i_r,j} & a_{i_r,j+1} & \ldots & a_{i_r,j+t} & & & \end{bmatrix} \boxplus U_{c_1} \boxplus \ldots \boxplus U_{c_s} \tag{5.23}$$

It is worth mentioning there exist other definitions of a super-node which allow to amalgamate even more nodes from the original post-ordered tree. For example, **mult-frontal-original:2** pointed out, in [**mult-frontal-original:2**], 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 the methods consist of three distinct phases, namely: analysis, numerical factorization and solution phases. The analysis phase includes fill reducing matrix re-ordering, symbolic factorization, post-ordering, node amalgamation, elimination tree construction, estimation of required memory space 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 matrix decomposition, the solution step computes a solution vector x by means of backward and forward substitutions.

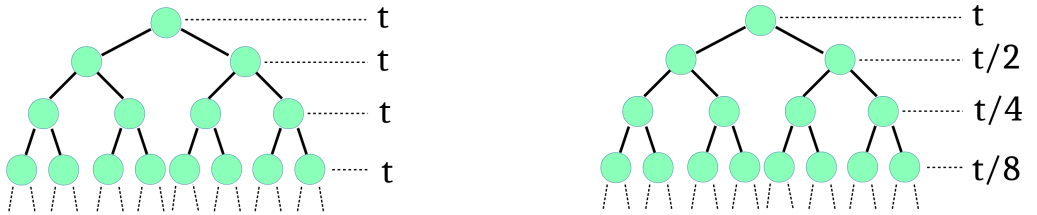
5.1.2.2. Parallelization Aspects

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 a 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 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.

place for figure 5.7

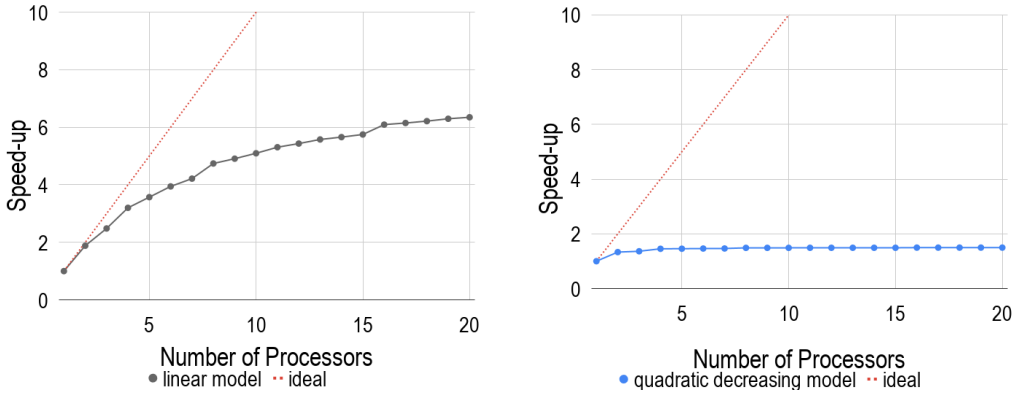
The models imply parallel computations within a level but sequential execution between them. In other words, to start computations at the next top level, factorizations of all nodes at the current one have to be fully performed. It also means that computations at the next level cannot be started even if there are some available free processors but factorization of the last node at the current level has not been completed yet. Thereby,



(a) Model 1: equal cost per level

(b) Model 2: quadratically decreasing cost per level

Figure 5.7.: Simple parallel models of the multifrontal method



(a) Theoretical speed-up of Model 1

(b) Theoretical speed-up of Model 2

Figure 5.8.: Theoretical speed-up

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 single node of each level. Therefore, it determines asymptotes on the corresponding speed-up graphs.

Figures 5.7a and 5.7b 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 5.7b . 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 maximum speed-up. It is worth mentioning that two models almost exhaust tree-task parallelism even with 20 processing elements. Further increase of processor count can only barely improve the overall parallel performance.

place for figure 5.8

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

Table 5.2.: Potential speed-up of linear and quadratic models

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 distribution of compute-intensity among nodes. 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 dependency. As an example, **mult-frontal-original:2** observed that factorization of the last 6 nodes took slightly more than 25% of the total number of floating point operations in case of application of the multi-frontal method to a $k - by - k$ regular model problem using a nine-point difference operator, [**mult-frontal-original:2**].

Node-data parallelism is often combined with tree-task parallelism which usually results in 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 the 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 mainly 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 the cumulative 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 matrix main diagonal can result in significant numerical inaccuracy of the Gaussian Elimination process. Therefore, partial pivoting is a crucial step of Gaussian Elimination. It implies to interchange 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, partial 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, it becomes a problem when it deals with direct sparse methods.

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

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 a frontal matrix.

Factorization of a column is suspended i.e. delayed, if equation 5.24 cannot be satisfied within a 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 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 leads 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, keep load balance computed during the analysis phase by appropriate mapping of processors across nodes of the tree. However, in this case, numerical accuracy usually degrades. In practice, values of α lay in a range of 0.2 to 0.04 . REF

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

By and large, solutions computed by direct sparse methods are usually numerically inaccurate, in some degree, and often demands to perform solution refinement. As an example, solution accuracy can be improved using the iterative refinement method. Code listing 5.3 shows a psedu-code of the method where parameter ω represents an estimation of the backward error, equation 5.25, [**mm-backward-error**]. In practice, the method usually takes 2 or 4 iterations to achieve sufficient numerical accuracy.

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

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

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

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

# compute initial residual
r = A * x - b

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

  # update solution
  x = x - d

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

Listing 5.2: Iterative refinement

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. The practice, in particular of ATHLET users, has been shown that it usually

takes 1 . . . 3 iterations to achieve desired numerical accuracy even with extreme small values of α.

At the end, it is worth mentioning that both refinement techniques, mentioned above, exploit only data-based parallelism and, therefore, scale well on distributed-memory machines.

5.1.3. Results and Conclusion

Direct sparse methods do not require a preconditioner, in fact, they can be used as a preconditioner of an iterative solver.

SDS: start - read comments

Preconditioners start: At the first step, some methods, table BRA, were tried with tier default settings. The algorithms were chosen because of their availability in PETSc and to run in parallel.

SDS: end

Table

Table with comparisons of different preconditioning for our test cases

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

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

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

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

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

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

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

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

grammar

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

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

to fullfill robustness creteria

GMRES.

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

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

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

proofreading

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

insert in the correct place: start

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

In general, the estimation of working space can be tricky due to pivoting. Because pivoting happens only during the numerical factorization it is not always possible to estimate enough space correctly beforehand. There exist some heuristics which allow to

use some numerical matrix information during symbolic factorization to better predict the amount of required space [**wsmp:direct-solution-of-general-system**].

insert in the correct place: end

5.2. Selection of a Sparse Direct Linear Solver Library

Fair to say, there is no single algorithm or software that is the best for all types of linear systems [**list-of-sparse-direct-solvers**].

Nowadays, there exist many different avaliable sparse direct solvers. Some of them are tunned for specific linear systems whereas others are targeted for the most general cases [**list-of-sparse-direct-solvers**]. 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 [**wsmp**], [**mumps-manual**], [**superlu-manual**]. 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 packages capable to run on distributed-memory machines, at the time of writing, based on works [**list-of-sparse-direct-solvers**] and [**petsc-web-page**].

It can be clearly observed, from table 5.3, that only MUMPS, PaStiX and SuperLU_DIST cover all requirements induced by GRS, see chapter 3, namely: open-source license and a direct interface to PETSc. It is interesting to notice that all libraries, mentioned above, are implementations of different sparse direct methods, namely: multi-frontal (MUMPS), left-looking (PaStiX) and right-locking (SuperLU_DIST). Moreover, PaStiX and SuperLU_DIST use only static pivoting [**pastix-manual**], [**superlu-manual**] whereas MUMPS provides a full implementation of the threshold pivoting strategy [**mumps-manual**], described in subsection 5.1.2.3.

To compare the libraries, a couple of flat-MPI tests were performed using GRS matrix set on HW1 machine. 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 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 unexpected long program execution. Results are summarized in tables 5.4, 5.5, 5.6 and in appendix A where numerical values are given in seconds.

Some problems were detected during SuperLU_DIST library testing. First of all, execution of *cube-64* and *k3-2* test-cases exceeded the set time limit. Secondly, it was

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.: List of packages to solve sparse linear systems using direct methods on distributed memory parallel machines [**list-of-sparse-direct-solvers**], [**petsc-web-page**].
Open* - the interface is not officially supported by the PETSc team

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.

To complete comparison and evaluate parallel performance SuperLU_DIST library, an additional test was conducted using 2D formulation of the Poisson problem with *100000* unknown. According to the results, SuperLU_DIST managed to complete matrix factorization 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.

place for figure 5.9

According to the initial and additional tests, MUMPS library showed the best parallel performance and scaling in contrast to 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

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

L and U in memory. PaStiX was the second fastest solver in our test. 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 that library.

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

In this chapter, we compared different sparse direct methods and their concrete implementations with their default settings with regard to GRS matrix set. Based on the obtained results and literature review, MUMPS library was chosen for the following study. In subsection 5.3.1, we make a detail review of MUMPS libraries and its specific traits.

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

MPI	MUMPS	PaStiX	SuperLU
1	1.55E+02	6.44E+01	crashed
2	6.28E+01	4.84E+01	crashed
3	5.06E+01	5.02E+01	crashed
4	4.17E+01	4.50E+01	crashed
5	2.52E+01	3.98E+01	crashed
6	2.58E+01	4.29E+01	crashed
7	2.65E+01	4.30E+01	crashed
8	2.59E+01	3.73E+01	crashed
9	1.95E+01	4.08E+01	crashed
10	1.91E+01	3.81E+01	crashed

MPI	MUMPS	PaStiX	SuperLU
11	1.77E+01	3.81E+01	crashed
12	1.60E+01	3.75E+01	crashed
13	1.42E+01	3.58E+01	crashed
14	1.45E+01	3.59E+01	crashed
15	1.47E+01	3.57E+01	crashed
16	1.41E+01	3.52E+01	crashed
17	1.54E+01	3.45E+01	crashed
18	1.52E+01	3.31E+01	crashed
19	1.52E+01	3.31E+01	crashed
20	1.38E+01	3.16E+01	crashed

Table 5.6.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and the matrix *k3-18* (1155955 equations)

5.3. Configuration of MUMPS library

5.3.1. Review of MUMPS Library

read comments

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 [**PARASOL**]. An important aspect of the researh 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

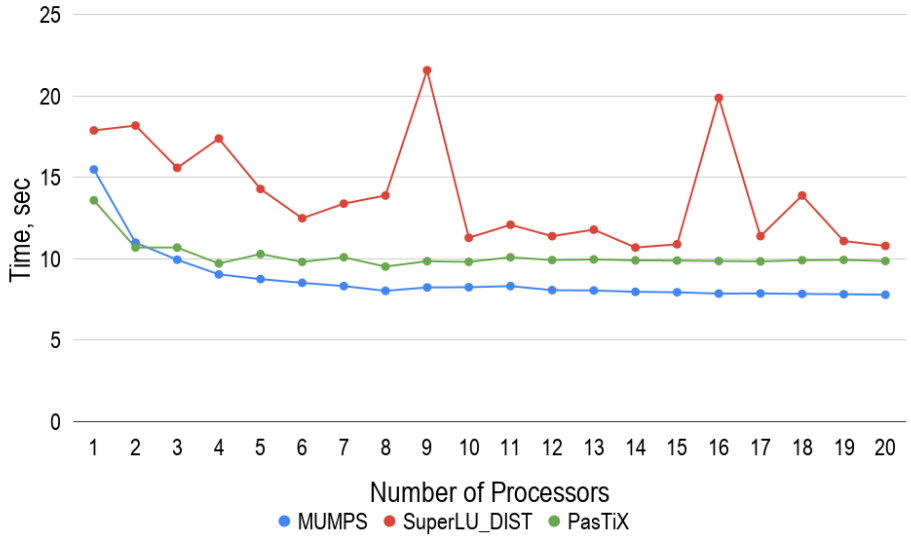


Figure 5.9.: Results of a flat-MPI test of MUMPS, PasTiX and SuperLU_DIST libraries with their default settings and a 5 point-stencil Poisson matrix (1000000 equations)

[**MUMPS:description**]. Since 2000 MUMPS had continued as an ongoing project and, by the time of writing, the library have contained almost 5 main releases.

As it was mentioned in section 5.2, MUMPS is an implementation of the multi-frontal 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 subsection, the analysis phase of MUMPS is examined since implementation of this phase often varies between libraries due to different parallel performance considerations.

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

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

5. Mapping

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

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 elimination tree building process. All computations at this step are performed using a directed graph $G(A)$ associated with the matrix A.

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

4) During 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 of the initial 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 belongs to one out of three different types, figure 5.10. Then, the host distributes tasks among all available processes in such a way to achieve good memory and compute balance.

Type 1 nodes are grouped in subtrees, according to the Geist-Ng algorithm [**geist1989task**],

and each subtree is processed by a single process to avoid the finest granularity, which can cause high communication overheads.

In case of type 2 nodes, the host process assigns each node to one process, called the *master*, which holds fully summed rows and columns of a node as well as performs 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 balance of all other processes and decides, *dynamically*, which of them, *slaves*, are going to participate to the node factorization. After that, the master informs the chosen slaves that a new task has been allocated for them, maps them according to 1D block column distribution and sends them the corresponding parts of the frontal matrix. Then, the slaves communicate 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 5.3.4.

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

It is important to mention if the root node size is less than a certain computer depended parameter, defined internally by MUMPS, the root node will be treated as the type 2, [**mumps-manual**].

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

place for figure 5.10

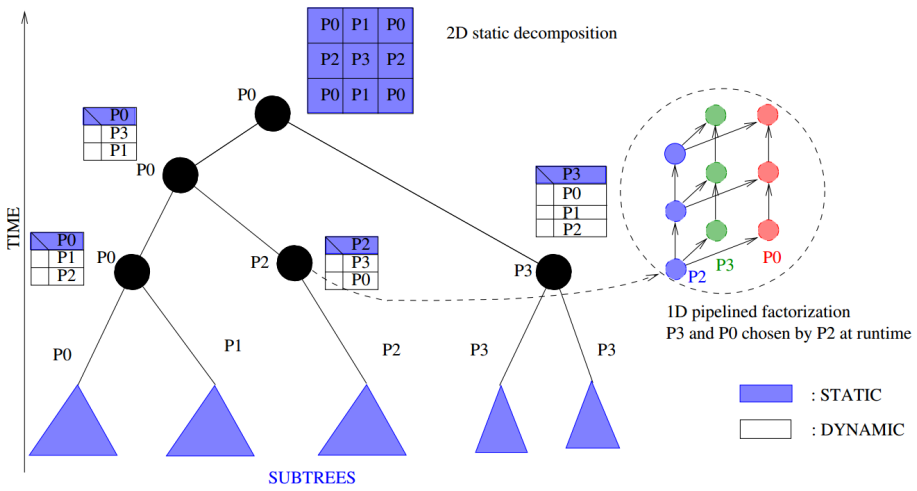


Figure 5.10.: MUMPS: static and dynamic scheduling **[l2012multifrontal]**

5.3.2. Choice of Fill Reducing Reordering

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 the 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 **guermouche2003memory**, in work [**guermouche2003memory**], for sequential execution of the analysis phase. **guermouche2003memory** 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 to 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, [**guermouche2003memory**].

In this subsection, we are going to investigate 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 subsection 5.3.1.

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

place for figure 5.11

According to the results, parallel performance of MUMPS can vary significantly and very sensitive to a applied fill-in reducing reordering algorithm. In average, the difference 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.

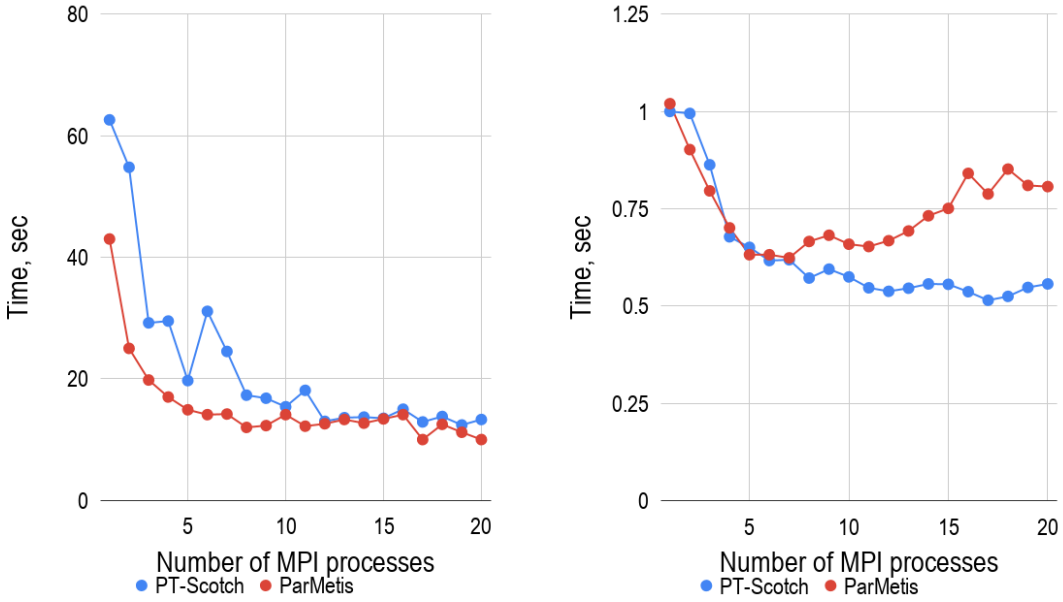
Considering results obtained with GRS matrix set, we can observe that PT-Scotch is the best choice for small and medium sized matrices, namely: *cube-5*, *cube-64*, *k3-2* and *pwr-3d* cases. Whereas, PerMetis tends to work better for relatively big systems, such as *cube-645* and *k3-18*, *However,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

During the test, we noticed that application 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, figure 5.12.

place for figure 5.12

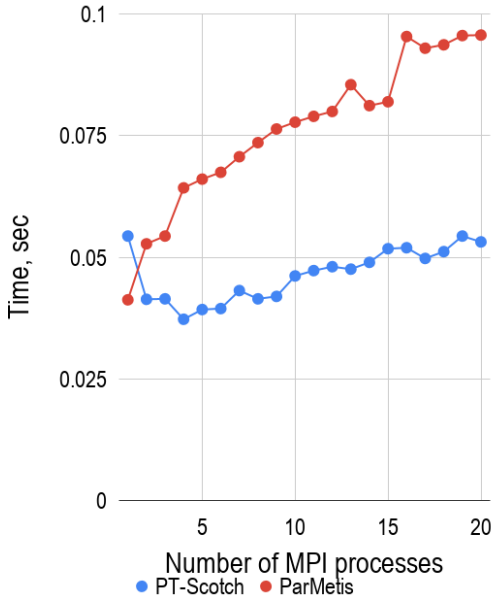
A simple profiling showed two important things. Firstly, numerical factorization time and time spent on the analysis phase had approximately the same order in case of sequential execution i.e. 1 MPI process. Secondly, while numerical factorization time were barely decreasing with increase of the number of processing elements, time spent on 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 the analysis phase contained several peaks at points where the processor count was equal to a power of two. We assumed the cause could

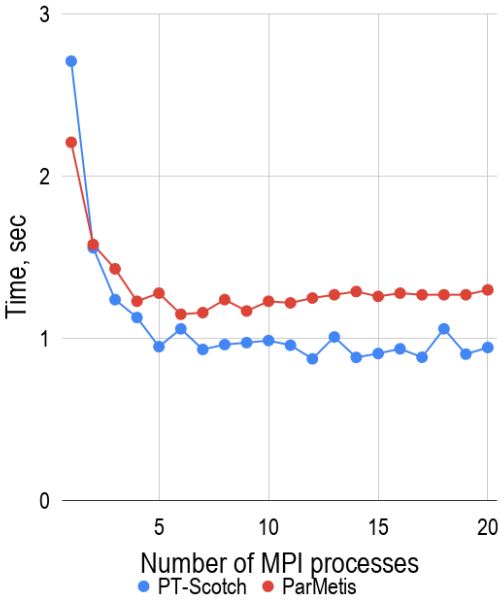


(a) k3-18

(b) cube-64

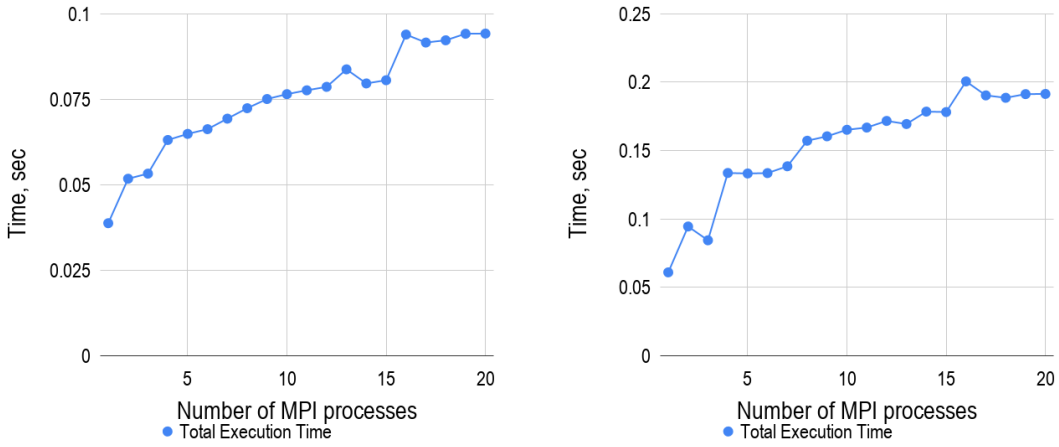


(c) pwr-3d



(d) k3-2

Figure 5.11.: Comparison of different fill-reducing algorithms



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

Figure 5.12.: MUMPS-ParMetis parallel performance in case of relatively small matrices

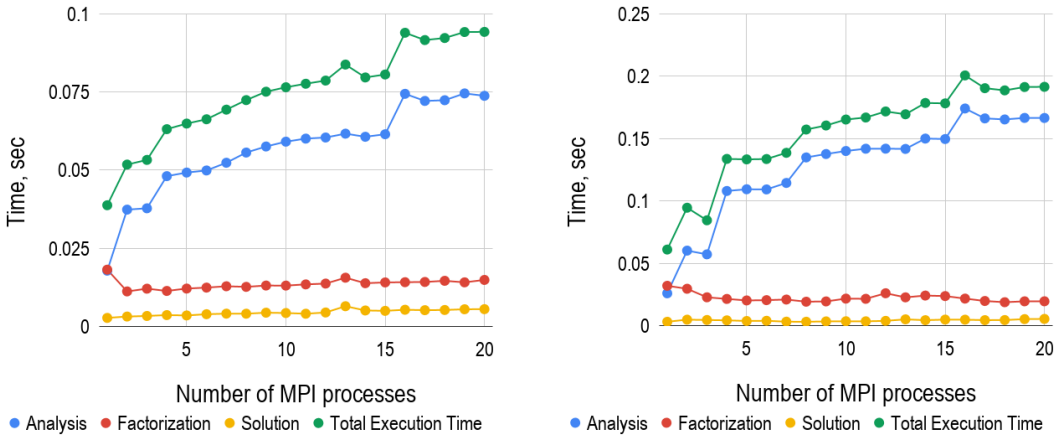
be due to either fill reducing reordering or process mapping steps. However, a detailed profiling and tracing of the analysis phase, which are out of the scope of this study, are required in order to give the exact answer. The results of profiling are shown in figure 5.13.

place for figure 5.13

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

In this subsection, we have presented the 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 significant improvements in terms of the overall execution time. We have showed there is no a single algorithm that performs the best for all test-cases. At the moment of writing, we have come to a conclusion there is no an indi-



(a) pwr-3d

(b) cube-5

Figure 5.13.: Profiling of MUMPS library with using ParMetis as a fill-in reducing algorithm in case of factorization of relatively small matrices

rect 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 B. 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.

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

5.3.3. MUMPS: 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 way to improve performance of a memory bound computer program running on distributed-memory machines is to distribute MPI processes equally among 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

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

equally among the memory channels.

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

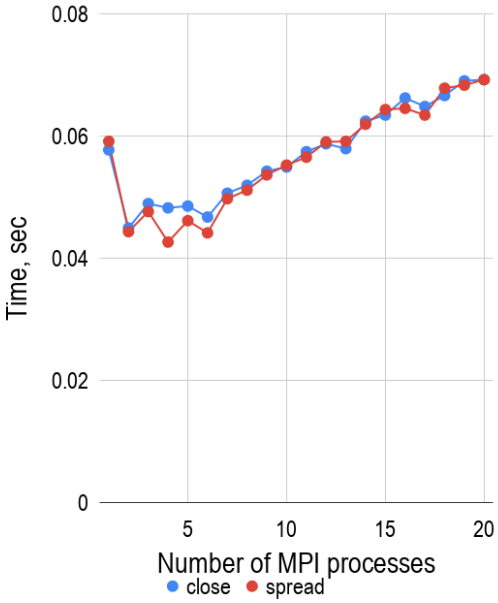
Therefore, a couple of tests were carried out with both GRS and SuiteSparse matrix sets in order to investigate influence of different pinning strategies on MUMPS parallel performance. For this group of tests, MUMPS was ran with the default settings but with a specific fill-in reducing algorithm assigned for each test-case according to tables 5.7 and 5.8. The tests were performed on both HW1 and HW2 machines using only the flat-MPI mode i.e. 1 OpenMP thread per MPI process. Comparison between different hardware also allows to investigate 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 of NUMA domains, respectively. Results are shown in figures 5.14, 5.15, 5.16 and in appendix C. The graphs depict 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, *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 process count i.e. the range from 2 to 12 MPI processes, where performance curves of *spread* and *close* strategies noticeably

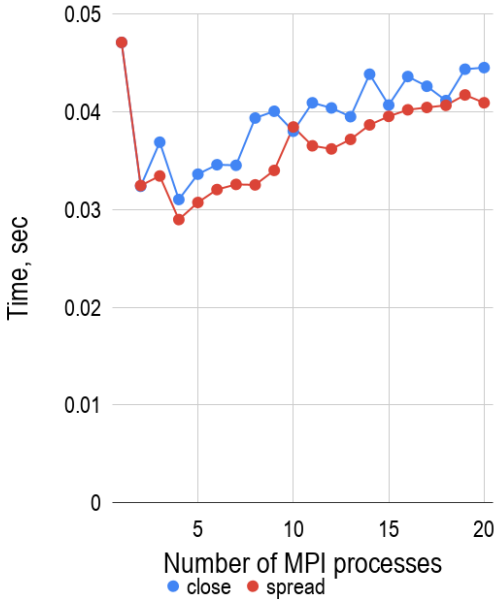
place for figure 5.14

place for figure 5.15

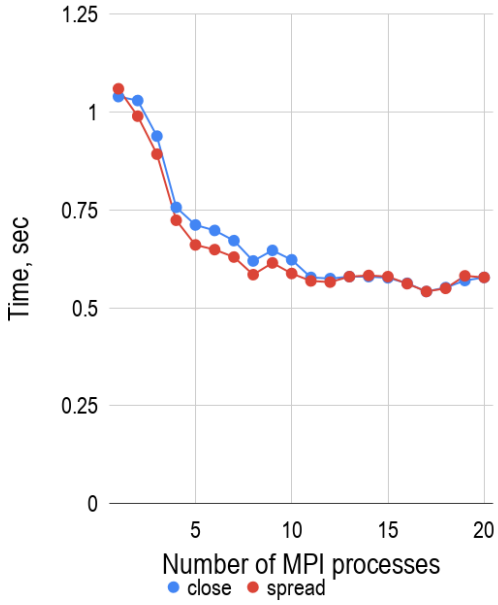
place for figure 5.16



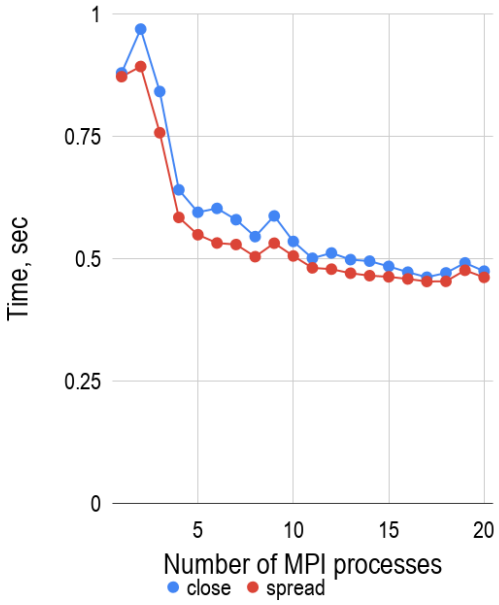
(a) HW1 - pwr-3d



(b) HW2 - pwr-3d

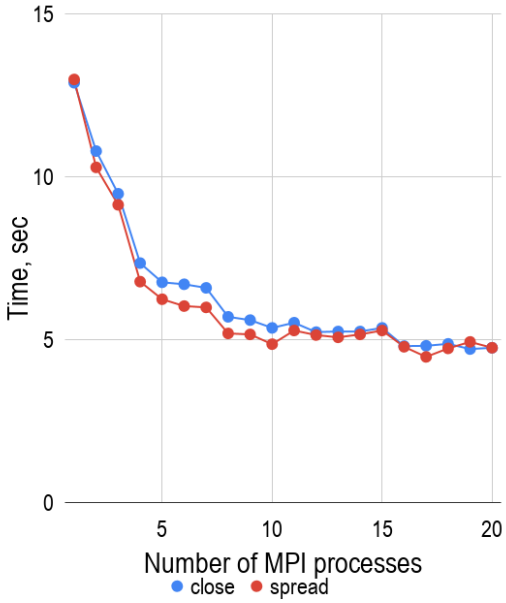


(c) HW1 - cube-64

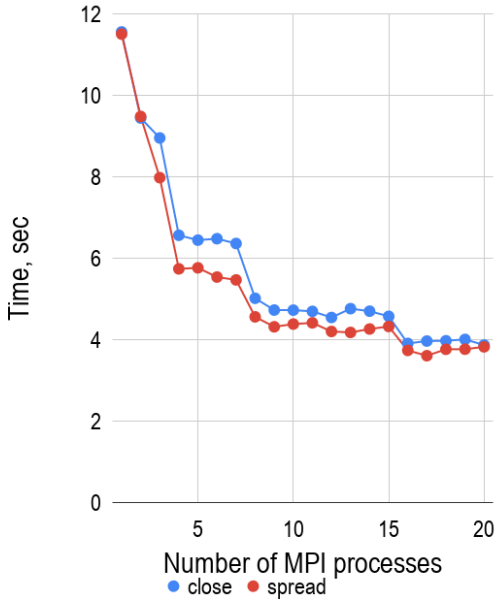


(d) HW2 - cube-64

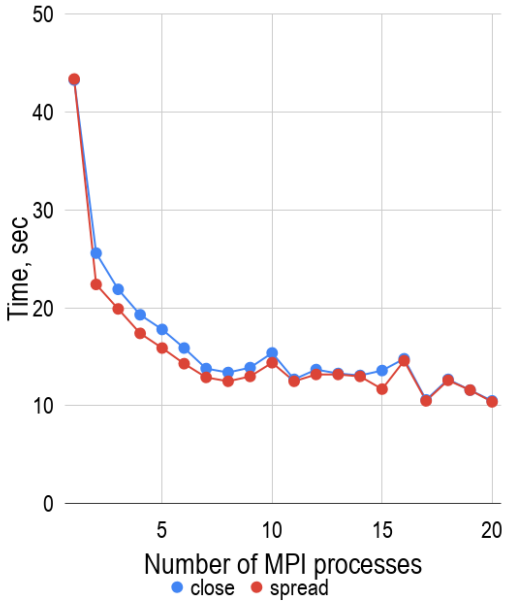
Figure 5.14.: Comparison of *close* and *spread* pinning strategies



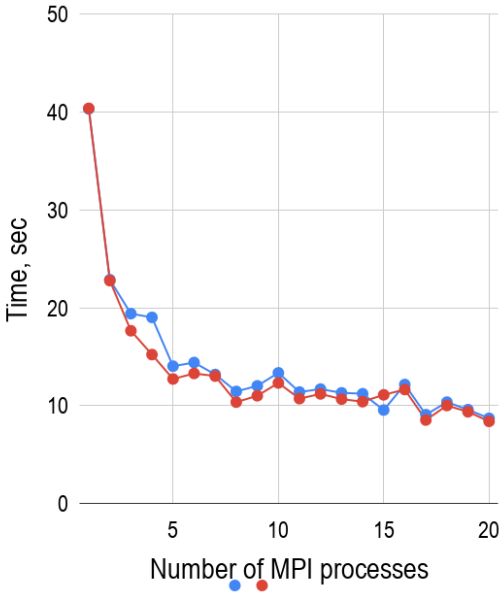
(a) HW1 - cube-645



(b) HW2 - cube-645

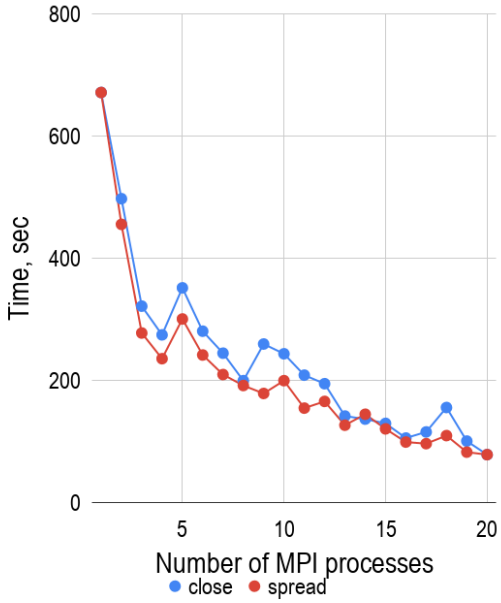


(c) HW1 - k3-18

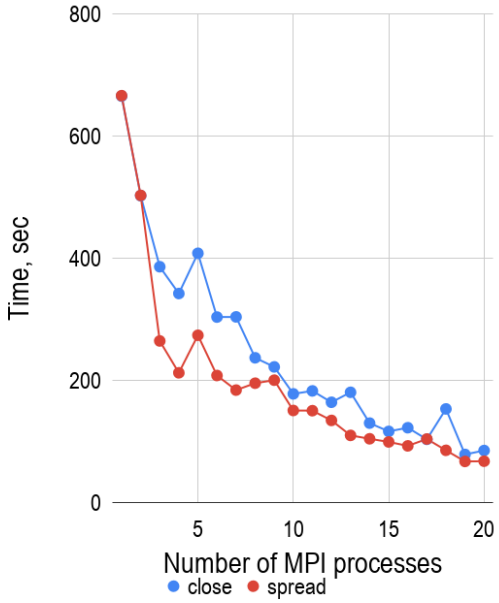


(d) HW2 - k3-18

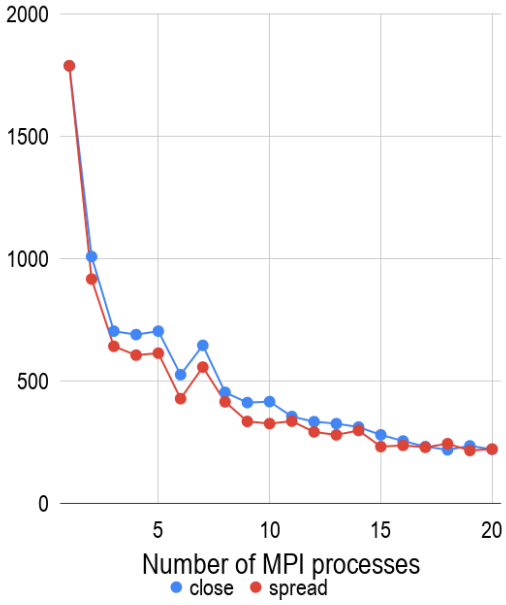
Figure 5.15.: Comparison of *close* and *spread* pinning strategies



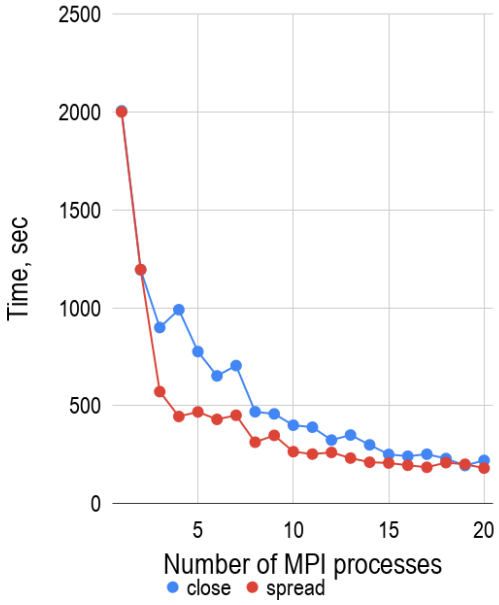
(a) HW1 - PFlow_742



(b) HW2 - PFlow_742



(c) HW1 - CurlCurl_3



(d) HW2 - CurlCurl_3

Figure 5.16.: Comparison of *close* and *spread* pinning strategies

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 posses only 20 cores in a compute-node, because the points basically represent exactly the same process distributions.

It is also important to investigate performance gain around the saturation point i.e. the point after which a further increase of the process count results in either stagnation or drop of computer program speed-up. It is worth pointing out that from time to time it becomes very difficult to decide where the saturation point locates 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.

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

A study of tables 5.9 and 5.9 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 of efficiency and speed-up of MUMPS.

In this subsection, we have shown 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

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

Table 5.10.: Analysis and comparison of MUMPS performance at the saturation point between HW1 and HW2 for SuiteSparse matrix set.
*ROM - run out of memory

increase of the number of NUMA domains.

The result of this study can be relevant for energy-efficient parallel computing where a strong requirements to program efficiency are applied. This fact usually forces the user to 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 the tests, *spread*-pinning has been chosen for the rest of the study. This process distribution can be easily achieved by means of some advanced OpenMPI command line options, for example *–rank-by* and *–bind-to*, as following:.

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

Listing 5.3: An example of *spread*-pinning with using advanced OpenMPI command line options

5.3.4. Choice of BLAS Library

To perform column eliminations of a fully summed block of a frontal matrix, MUMPS intensively calls GEMM, TRSM and GETRF subroutines which are parts of BLAS and

LAPACK libraries. Figures 5.17 and 5.18 illustrate application of the BLAS subroutines to factorization of a type 2 node.

place for figure 5.17

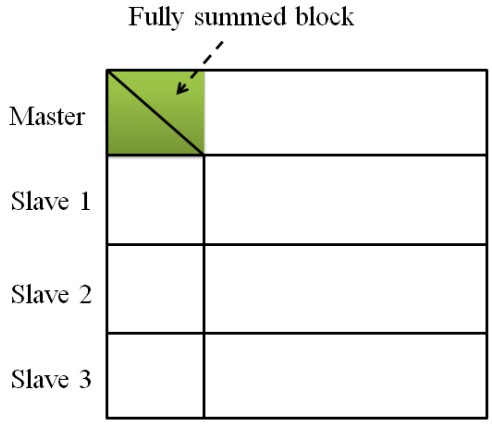


Figure 5.17.: MUMPS: 1D block column distribution of a type 2 node

place for figure 5.18

Both BLAS and LAPACK 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 [**netlib-overview**].

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.

In its turn, LAPACK is designed in such a way so that as much as possible computations are performed by calling BLAS subroutines. This allows to achieve high efficiency for operations such as LU, QR, SVD decompositions, triangular solve, etc. on modern computers. However, the Netlib BLAS implementation is written for an abstract general-purpose central processing unit where hardware parameters are based on market statistics. Hence, it is not possible to achieve the maximum possible performance on a specific hardware.

Therefore, there exist special-purpose, hardware-specific implementations of the library developed by hardware vendors i.e. IBM, Cray, Intel, AMD, etc., as well as open-source tuned implementations such as ATLAS, OpenBLAS, etc. To achieve full compatibility, the developers consider the Netlib implementation of BLAS library as

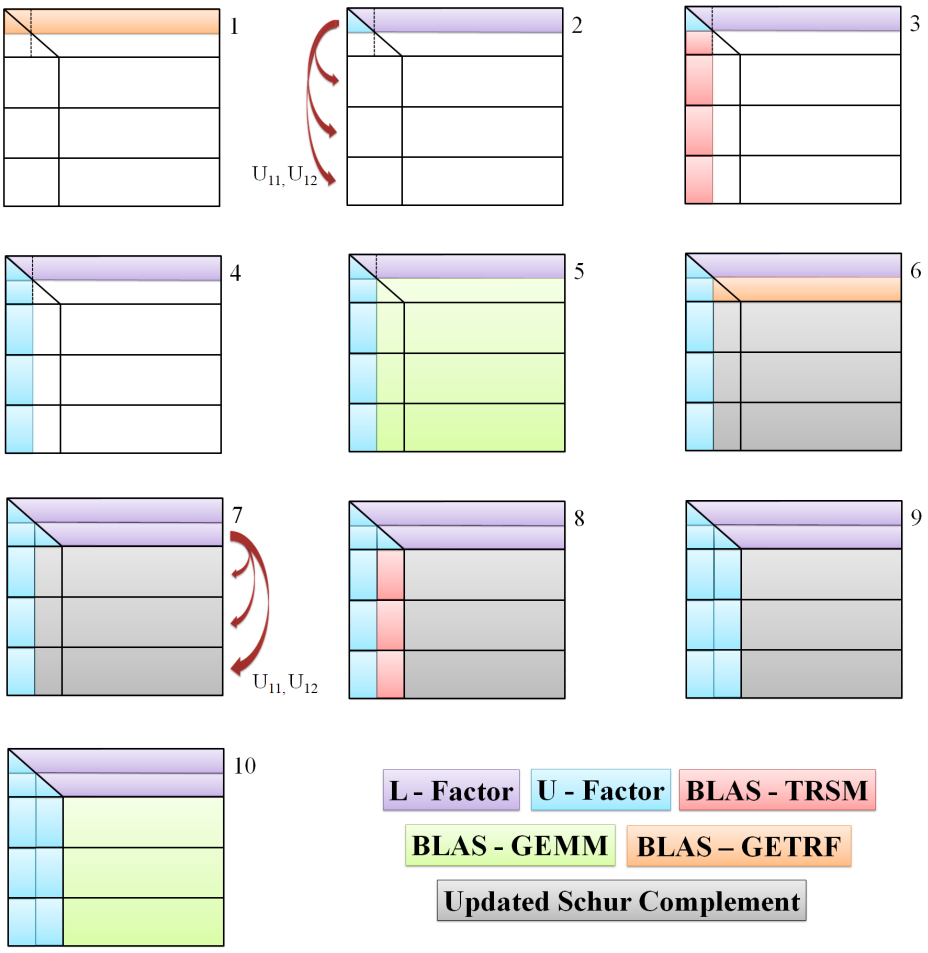


Figure 5.18.: MUMPS: An example of a type 2 node factorization

the standard, or 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 or LAPACK 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.

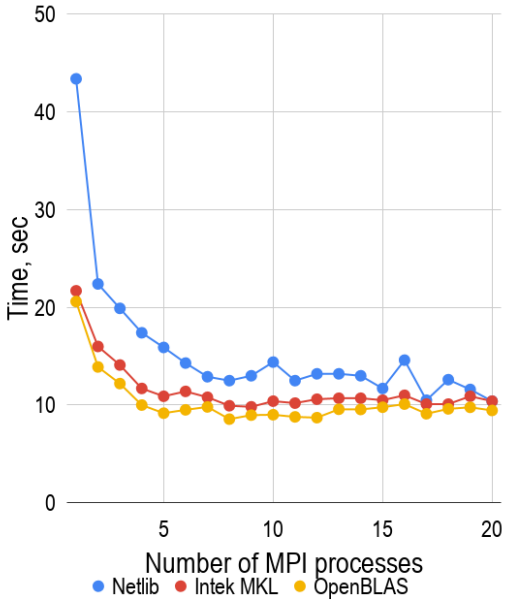
Among all libraries listed in table 5.11 there were only four available in HW1 machine environment, namely: Netlib BLAS, Intel MKL, OpenBLAS and ATLAS. However, installation of ATLAS requires to switch off dynamic frequency scaling, also called CPU throttling, to allow ATLAS configuration routines to 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. The test results were obtained on HW1 machine and are represented in figures 5.19, 5.20 and appendix D.

place for figure 5.19

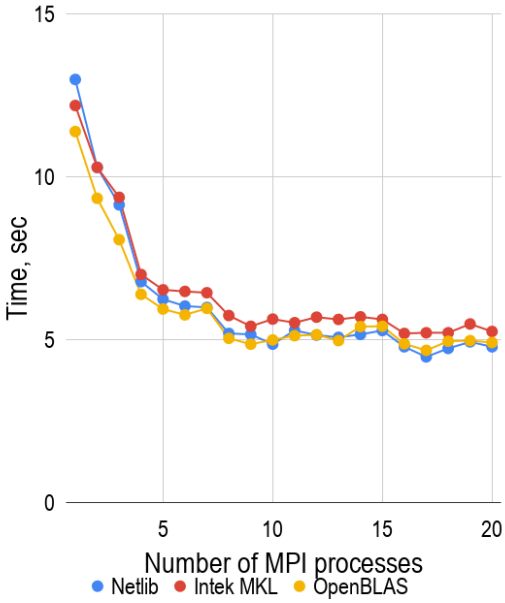
place for figure 5.20

The tests show that OpenBLAS outperforms both Netlib and Intel MKL libraries in case of GRS matrix set. In average, OpenBLAS is about **13%** faster than the default Netlib implementation and approximately **21%** faster than Intel MKL library. It is interesting to notice Intel MKL 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 comparison are summarized in tables 5.12 and 5.13.

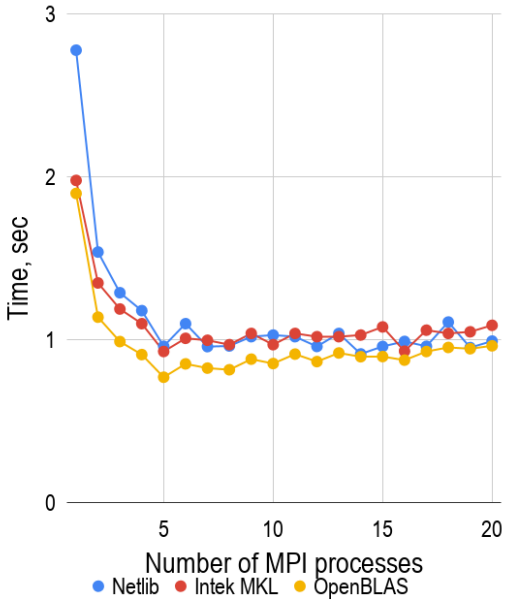
It can be clearly observed from the tables that test-cases derived from GRS matrix



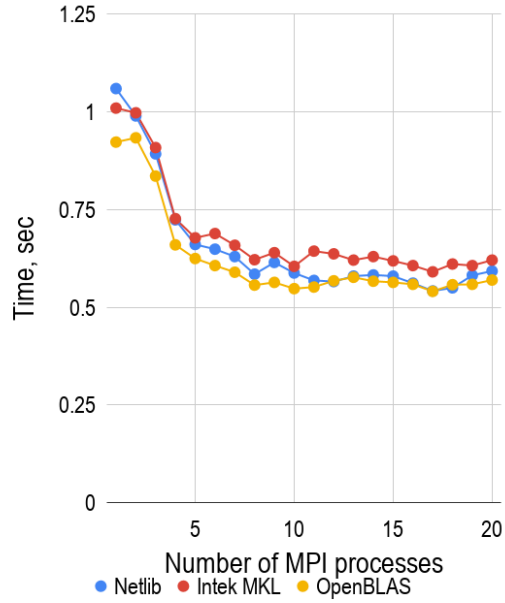
(a) k3-18



(b) cube-645

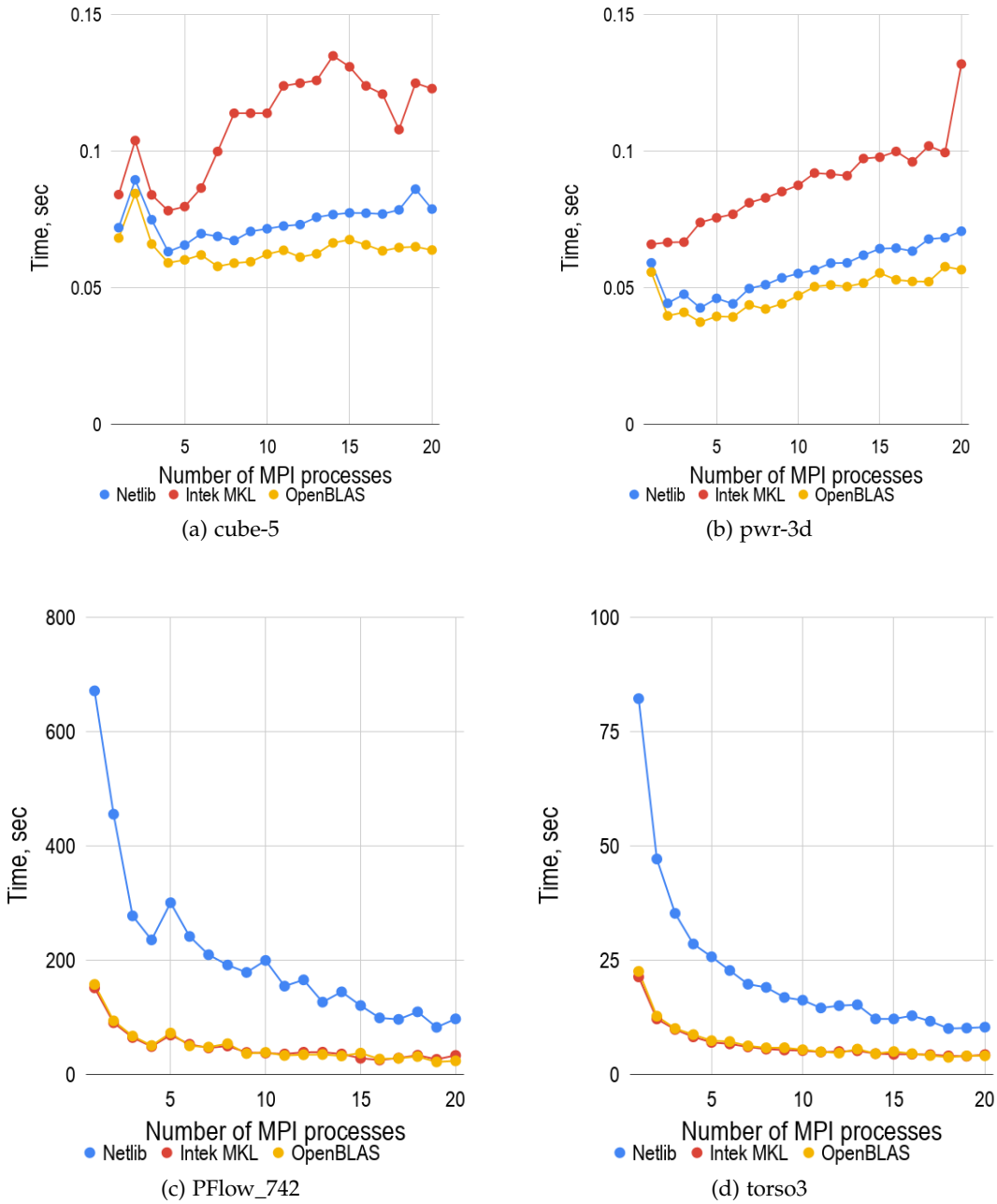


(c) k3-2



(d) cube-64

Figure 5.19.: MUMPS: comparison of different BLAS libraries with using GRS matrix set on HW1 machine



(a) cube-5

(b) pwr-3d

(c) PFlow_742

(d) torso3

Figure 5.20.: MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine

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

Table 5.11.: Commercial and open source BLAS implementations [**wiki:blas-implementations**]

set demonstrate insignificant improvements of execution time in constant 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 by the root and type 1 nodes. As it was mentioned in subsection 5.3.1, 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 or not. Even if it is a case performance of BLAS can be limited because of small frontal matrix sizes of such nodes.

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

Table 5.12.: Comparison of different MUMPS-BLAS configurations applied to GRS matrix set

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

Table 5.13.: Comparison 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 inefficient amalgamation process of the corresponding elimination tree resulted from the matrix sparsity pattern.

Based on the obtained results, comparison 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 shown where and how MUMPS utilizes BLAS and LAPACK libraries. We have compared two tuned BLAS implementations with a 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 of some test-cases as well as we have talked about probable specifics of linear systems generated by ATHLET.

5.3.5. MPI-OpenMP Tuning of MUMPS Library

As it was mentioned in subsection 5.3.1, the development of MUMPS began in 1996 when message-passing programming paradigm dominated in parallel computing. Therefore, the library originally was designed only for distributed-memory machines.

In 2010, **chowdhury2010some** published their first experiments and some issues, in [**chowdhury2010some**], of exploiting shared memory parallelism in MUMPS. The authors showed that it was possible to achieve some improvements in multicore systems using multi-threading, given a purely MPI application. However, later **l2013introduction** mentioned, in [**l2013introduction**], 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 of 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 done by **chowdhury2010some**, 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 [**l2013introduction**] and [**chowdhury2010some**] 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 the data provided in the papers. However, at the same time, the results also show that sometimes 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. **l2013introduction** 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 leaded to better parallel performance when the ratio was high. On the other hand, they observed the absolutely opposite result with relatively small ratios. Unfortunately, **l2013introduction** did not provide any quantitative measure for the notion of small and large ratios in their work [**l2013introduction**].

It is also interesting to notice that parallelization of region 1 using a multi-threaded BLAS library gives the most of parallel performance improvement for mixed or pure OpenMP strategies, according to the results from [**l2013introduction**]. 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, data movement and execution of *if-statements* which always result in low computational intensity.

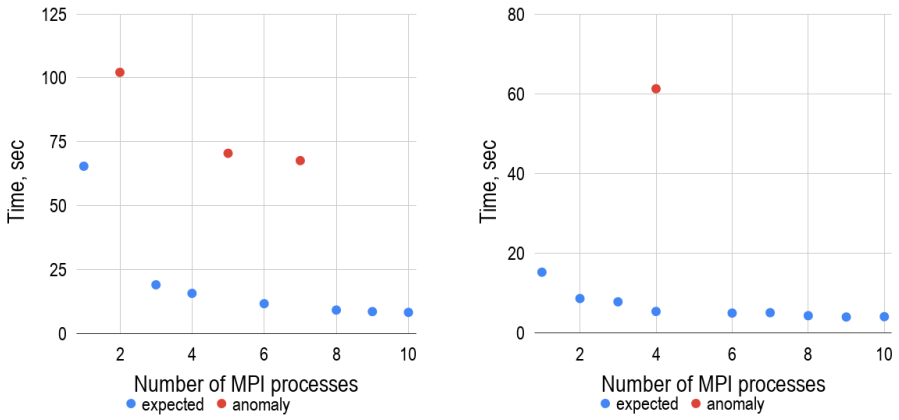
We have to admit that both works, [**chowdhury2010some**] and [**l2013introduction**], are relatively old and the analysis above may be not complete and full. Because MUMPS is a dynamic developing project, we can expect that adaptation of shared-memory parallelization in MUMPS has been significantly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

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

- 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 factorization of matrix *cube-645*, the increase reached almost **450%** in contrast to the pure sequential execution.

place for figure 5.21



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

Figure 5.21.: Anomalies of MUMPS-OpenBLAS configuration running with 2 OpenMP threads per MPI process

Multiple conflicts between application and system threads were observed using *htop* software as an interactive process viewer. Figure 5.22 shows a snapshot taken during

factorization of matrix *k3-18* running with 1 MPI process and 20 threads.

place for figure 5.22

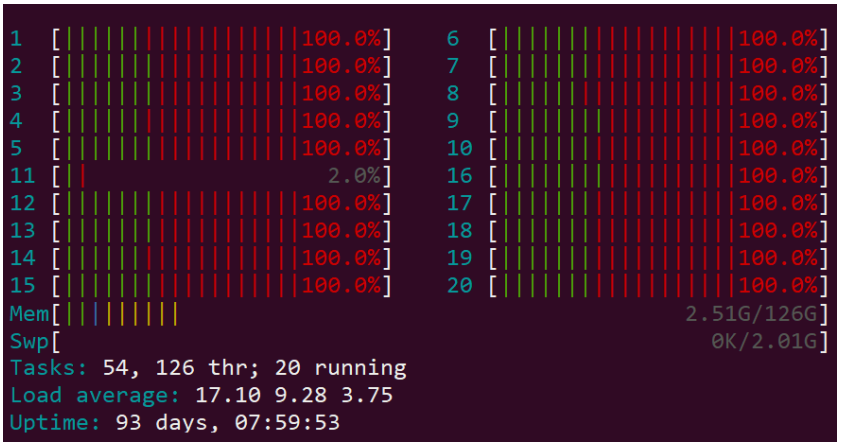


Figure 5.22.: A MUMPS-OpenBLAS thread conflict in case of *k3-18* matrix factorization (green - application threads, red - system threads)

It is difficult to state what exactly caused such behavior. However, **chowdhury2010some** 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 [**chowdhury2010some**]. 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 modes were tested in order to check influence of shared-memory parallelism on parallel performance of MUMPS as well as to limit the amount of testing. The following strategies were chosen: 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 with the aim of checking whether results would be consistent between different hardware running under different operating and environment settings. The test results are represented in tables 5.14 5.15, 5.16 and 5.17 where numerical values are given in seconds.

According to the results, 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, the

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.: Compassion of different hybrid MPI/OpenMP modes used for GRS matrix set on HW1

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

Table 5.15.: Compassion of different hybrid MPI/OpenMP modes used for GRS matrix set on HW2

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*. 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 that improvement came only from reducing 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

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.: Compassion of different hybrid MPI/OpenMP modes used for SuiteSparse matrix set on HW1

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

Table 5.17.: Compassion of different hybrid MPI/OpenMP modes used for SuiteSparse matrix set on HW2
*ROM - run out of memory

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 a high ratio of large and small fronts of this particular test-case.

According to the test outcomes, we have observed a negligible improvement in MUMPS parallel performance from 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 5.3.4 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 from SuiteSparse matrix set have shown an advantage of hybrid parallel computing, especially in case of *Geo_1438* matrix factorization.

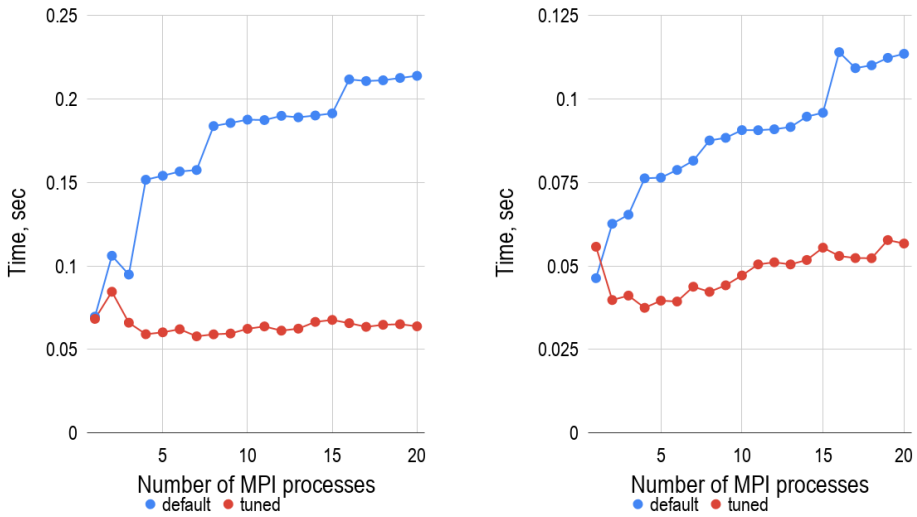
These contradictory results obtained from two different matrix sets second our reasoning of specifics of linear systems generated by ATHLET software. Again, we presume that assembly trees resulted from GRS matrices are mostly formed with subtrees filled with type 1 nodes where each subtree is processed by 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 programming. As it is in case of fill reducing reordering algorithm selection, subsection 5.3.2, it is not possible to find an optimal mixed MPI/OpenMP strategy in advance without performance testing. We have come to the concluded that flat-MPI mode is the best one for GRS matrix set and provided our reasoning for that. Generally speaking, there are 3 reason to use this mode in our case. Firstly, the mode always resulted in more efficient hardware utilization. Secondly, the improvements obtained with MUMPS-Intel MKL configuration running with optimal hybrid MPI/OpenMP modes can deteriorate performance gain obtained with MUMPS-OpenBLAS flat-MPI configuration, shown in subsection 5.3.4. 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 significant reduction of amount of testing.

5.4. Results

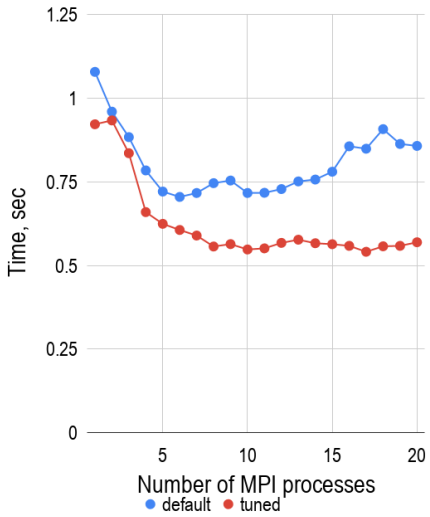
Figures 5.23 and 5.24 show comparisons of MUMPS parallel performance before and after application of the optimal MUMPS settings, found in subsections 5.3.2, 5.3.3, 5.3.4, and 5.3.5, to GRS matrix set. Results labeled as *default* were obtained using the fill

reducing reordering algorithm provided by ParMetis library because it had been used by ATHLET users before the current study.

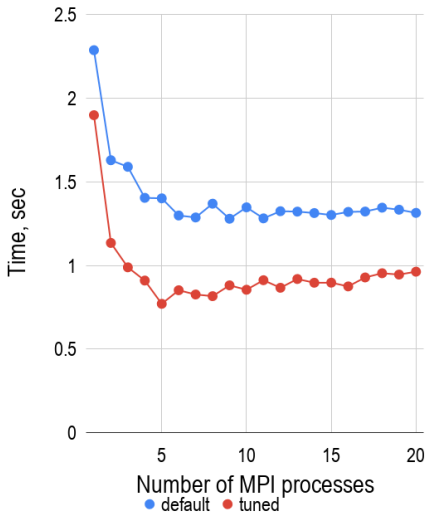


(a) small system: cube-5

(b) small system: pwr-3d

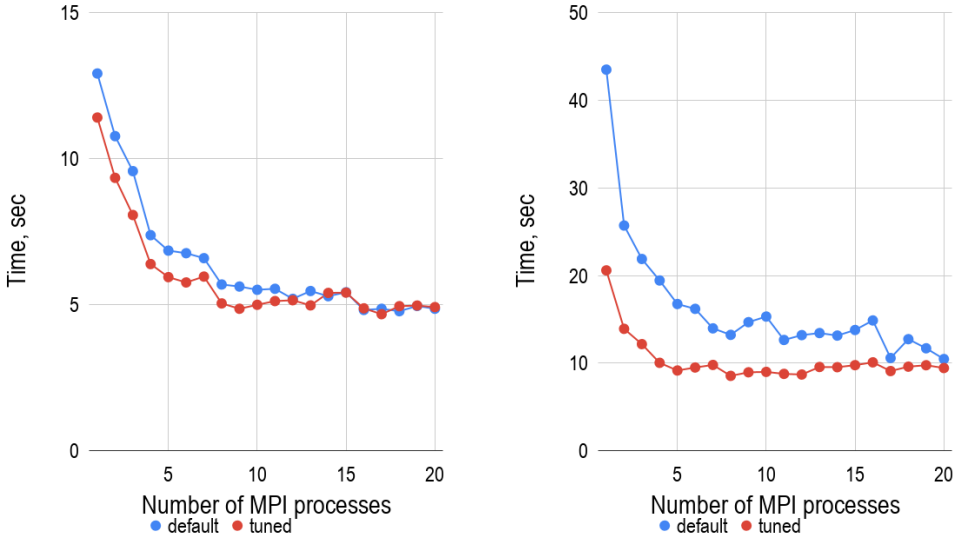


(c) medium system: cube-64



(d) medium system: k3-2

Figure 5.23.: Comparison of MUMPS parallel performance before and after application of optimal settings to small- and middle-sized GRS problems



(a) large system: cube-645

(b) large system: k3-18

Figure 5.24.: Comparison of MUMPS parallel performance before and after application of optimal settings to large-sized GRS problems

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, application of PT-Scotch for these systems of equations results in a drastic change of strong scaling behavior, see figures 5.23a and 5.23b, 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 factorization 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* test-case considerably improves. Additionally, application of PT-Scotch to *cube-64* matrix results in shifting of the optimal MPI process count, the saturation point, from 5 to 10 and, as a result, improves efficiency of the solver. Applied settings reduce execution time around the corresponding saturation points by almost **31%** on average for these type of GRS matrices.

Performance improvements in parallel factorization of large-sized GRS systems comes only from optimal processes pinning and application of OpenBLAS library because of

usage of the same optimal fill reducing reordering algorithm, ParMetis. On average, performance increased almost by **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 lacks type 2 nodes. However, the saturation points of both test-cases are shifted towards lower values of MPI process count which result in a considerably improvement of hardware utilization. For example, a detailed study of *k3-18* performance graph, figure 5.24b, shows the optimal MPI process count value 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.

By and large, in this subsection we have shown that application of the optimal settings leads to the total accumulative improvements of factorization time and hardware utilization.

5.5. 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 much suitable for this purpose because of its robustness.

In subsection 5.2, we have tested different direct sparse solvers, namely: SuperLU_DIST, PasTiX and MUMPS. MUMPS showed better parallel performance among the other solvers according to 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 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 an optimal hybrid MPI/OpenMP processes/threads distribution

The 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 subsection, 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 which are presented at the end of each subsection that we are going to reference to bellow.

1. In subsection 5.3.2, it has been shown that parallel performance of MUMPS is quite sensitive to an applied fill-in reducing reordering algorithm. 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 have assumed that the obtained conclusion can be not accurate due to small size of GRS matrix set. At the moment of writing, we have not found any indirect method to predict a correct algorithm beforehand. Thus, 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 5.3.3, influence of different process pinning strategies on MUMPS parallel performance has been investigated. The tests have shown that equal distribution of MPI process among all available NUMA domains always results in additional performance gain.

3. In subsection 5.3.4, we have tested MUMPS configured with 3 different implementations of BLAS library, namely: Netlib, OpenBLAS and Intel MKL. The results have shown that application of OpenBLAS library always results in better parallel performance.

4. In subsection 5.3.5, we have investigated impact of various MPI/OpenMP processes/threads distributions 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 subsection 5.4, we have studied the overall impact of introduced configuration changes found in subsections 5.3.2, 5.3.3, 5.3.4 and 5.3.5. Testing shows the changes results in a positive accumulative effect leading to considerable improvements of both factorization time and hardware utilization.

During the study, we have noticed that the optimal value of MPI process count lays within the range between 1 and 4 in case of small-sized GRS matrices and 4 and 8 for middle- and large-sized problems. The 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 the number of non-linear function evaluations which is quite a computationally intensive operation. Minimization is performed by means of efficient treatment of non-zero entries of a sparse matrix. The problem is also known as matrix partitioning.

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

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

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 **??** does not exploit Jacobian matrix sparsity and, therefore, estimation of a Jacobian matrix requires N function evaluations.

place for figure **??**

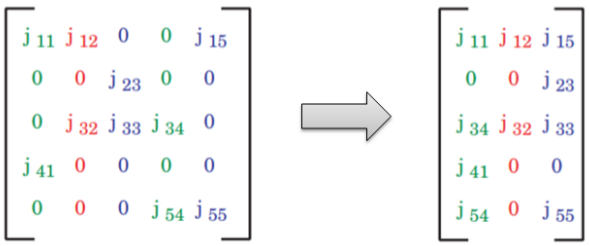


Figure 6.1.: An example of matrix coloring and compression [**gebremedhin2005color**]

The compression algorithm is based on the notion of *structurally orthogonal* columns

i.e. columns which do not share any non-zero entry in a common row. Figure **??** shows an example of a matrix compression where each color denotes independent *structurally orthogonal* columns.

Having obtained a compressed form of Jacobian, another set of vectors $d \in \mathbb{R}^N$, also known as seed vectors, can be used to perform function perturbation instead of unit vectors e_k. A seed vector d has 1's in components corresponding to the indices of columns in a structurally orthogonal group of columns, and zeros in all other components [**gebremedhin2005color**]. 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 $F(y + d)$ [**gebremedhin2005color**].

It is obvious the algorithm requires to partition a matrix into the fewest amount of groups, colors, in order to achieve the most of efficiency. There exist various methods and huristics dedicated to that particular problem. **gebremedhin2005color**, in work [**gebremedhin2005color**], conducted one of the most recent comprehensive studies in this field and summarized different matrix partitioning algorithms proposed over the last 20 years. Currently, Jacobian matrix compression has been successfully implemented in ATHLET by means of the corresponding built-in PETSc subroutines through NuT interface.

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

place for figure **??**

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

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

All **code listings**, presented in this part of the study, are written in **pseudo-code** and intended for for convenience of reading. The aim is to show and display the main

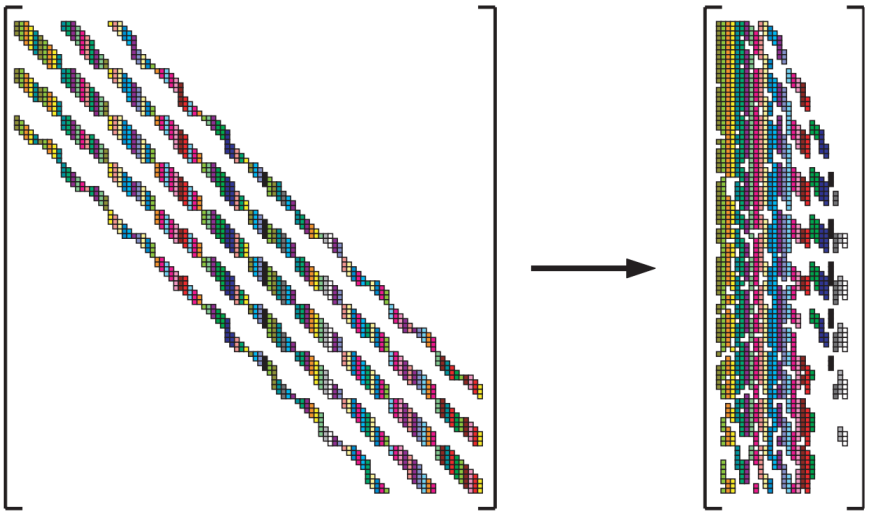


Figure 6.2.: An example of an efficient Jacobian matrix partitioning **[gebremedhin2005color]**

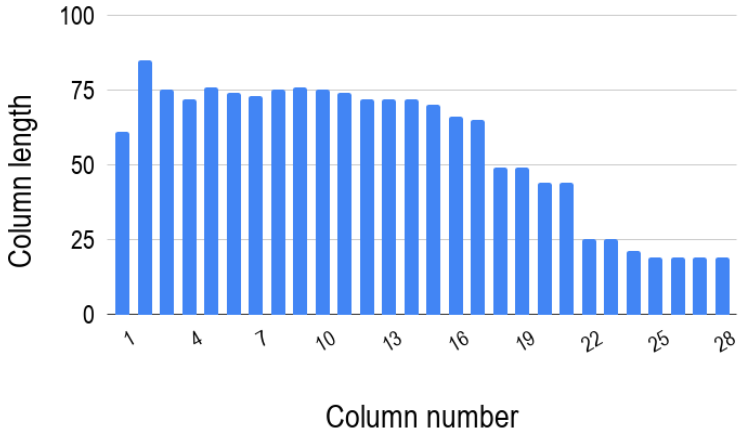


Figure 6.3.: Column length distribution of the example of figure **??**

ideas skipping non-relevant parts of the source code. The **pseudo-code** is a mixture of several programming languages, namely: **Python**, **C/C++**, **Fortran**, **MPI**.

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

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

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

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

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

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

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

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

```

Listing 6.1: Pseudocode of the default ATHLET-NuT coupling: ATHLET part

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

while nut_running:
  if rank in heads:

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

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

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

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

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

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

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


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

Listing 6.2: Pseudocode of the default ATHLET-NuT coupling: NuT part

6.2. Accumulator Concept

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

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

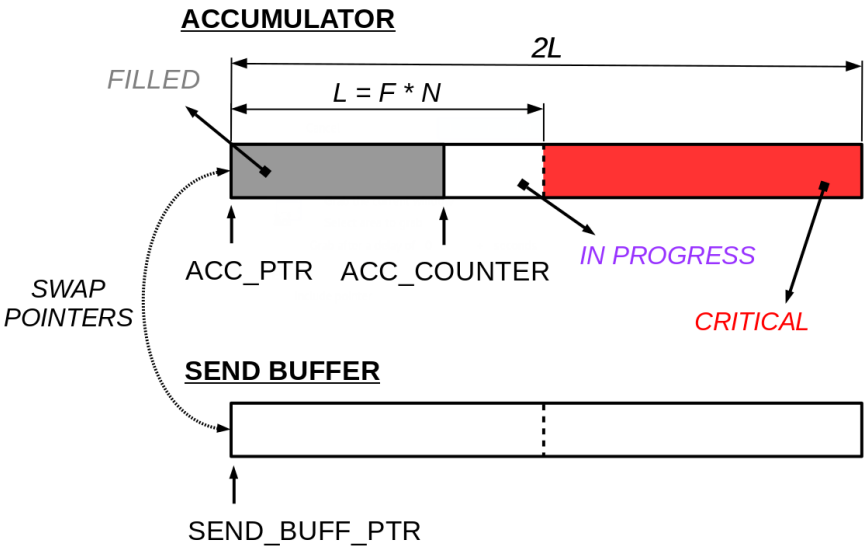


Figure 6.4.: Accumulator concept

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, using an assumption that a column vector cannot exceed the Jacobian matrix size. The assumption is based on long-term and first-hand experience of ATHLET-NuT users. On another hand, the event, described above, triggers a signal for a regular pointer

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

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

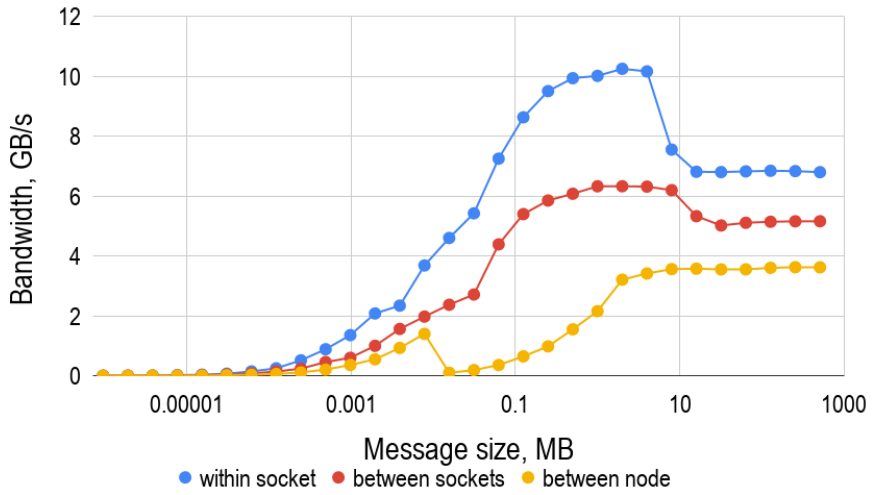
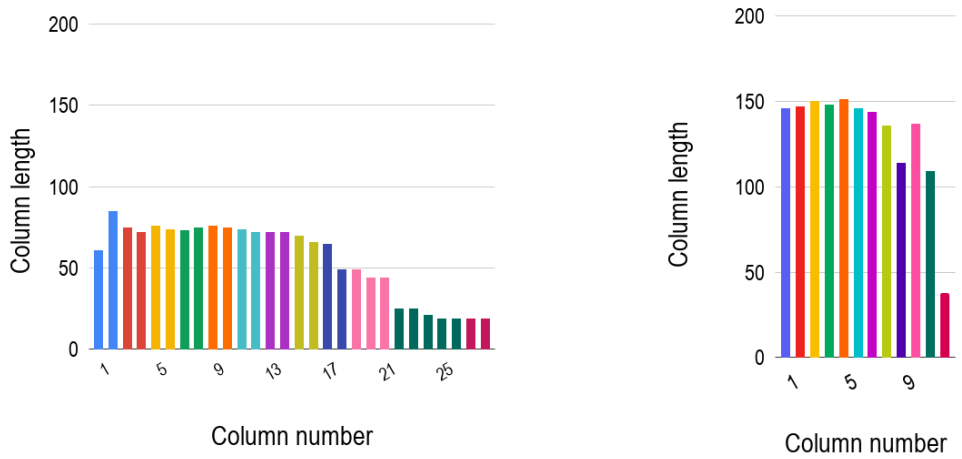


Figure 6.5.: Technical characteristics of HW1 interconnection

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

place for figure **??**

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 operations. It means ATHLET gets blocked only during the last column transfer



(a) Before

(b) After

Figure 6.6.: Application of *accumulator* concept to the example of figure **??**, with $N = 100$ and $F = 1$

and MPI gives the execution control back only when the last piece of data has been successfully distributed among NuT processes.

6.3. Benchmark and Test Data

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

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

The benchmark fully replicates all basic ideas of the default ATHLET implementation and the *accumulator* concept. It focuses only on the compressed Jacobian matrix transfer and, therefore, does not include any expensive compute-operations such as function perturbations with seed vectors. The approach allowed to sufficiently speed

up time of development, comparison and testing which, in turn, helped to design to the final concept, described in section **??**, excluding mistakes made at earlier steps of development after several iterations of testing.

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

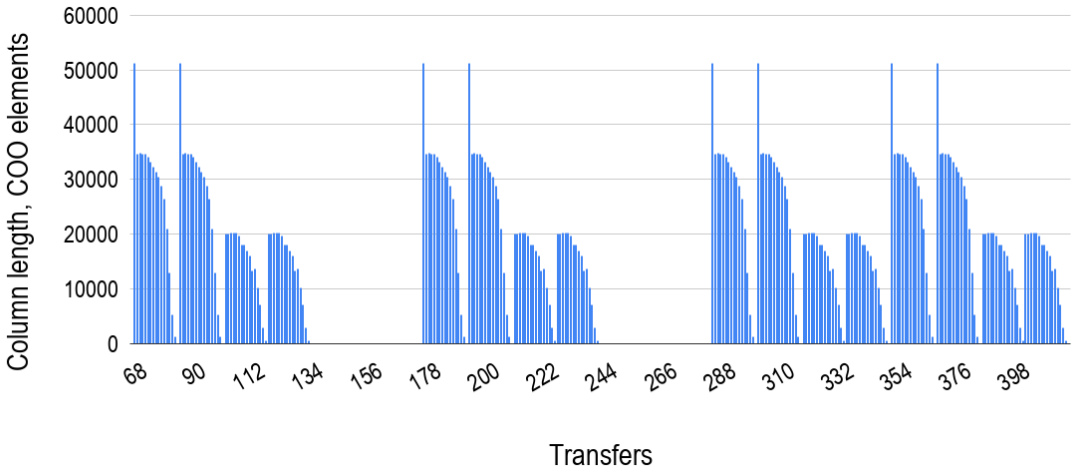


Figure 6.7.: A part of **cube-64** communication pattern

According to the *accumulator* concept, described in section **??**, 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 **??** represents an additional auxiliary class used for data accumulation. The pseudo-code of the benchmark client side is in listing **??**.

```
# 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 the auxiliary class

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

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

for column in recording:

  time_start = MPI_Wtime()

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

  # instantiate non-blocking data broadcast

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

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

Listing 6.4: A pseudocode of the benchmark: modified ATHLET part

6.4. Results

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

accumulation and non-blocking data transfer.

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

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

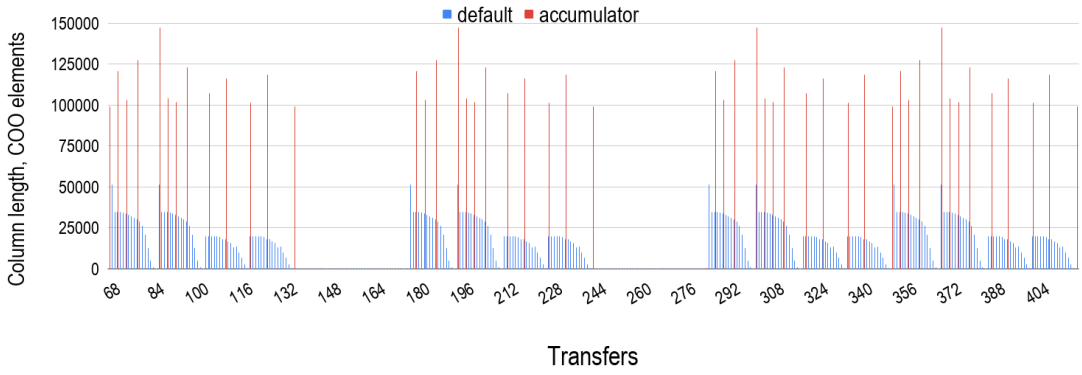
Benchmark name	BM2, %	BM1, %
within a socket	7.61	13.84
between sockets	9.04	26.26
between nodes	-2.06	3.20

Table 6.1.: Time reduction in contrast to the default approach in case of **cube-64** communication pattern

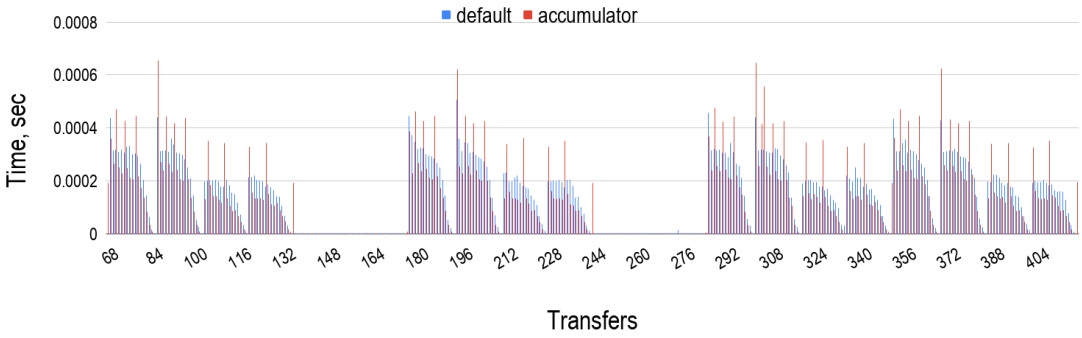
It turned out that **BM2** benchmark was slower than the default ATHLET approach in approximately **2%** in case of the inter-node communication. However, non-blocking data broadcast, according to **BM1** benchmark, helped to alleviate the slow-down and achieve almost **3%** of improvement.

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

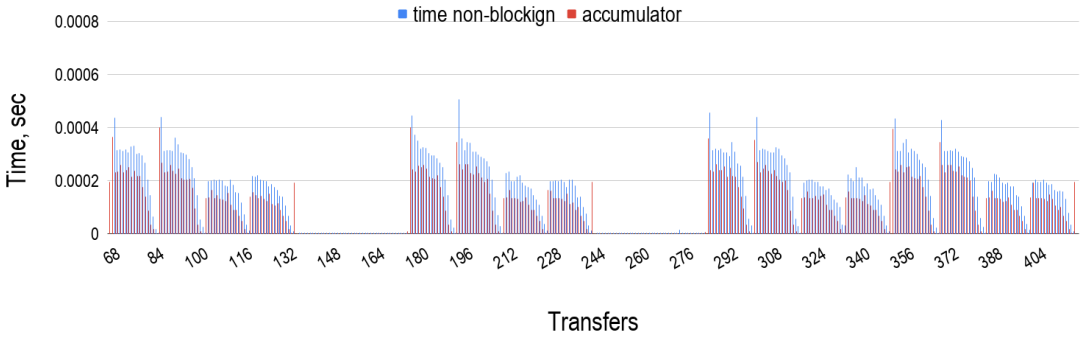
The slow-down resulting from pure data accumulation could be explained by automatic MPI protocol switching, namely: *Eager* and *Rendezvous* [**mpi:protocols-explanation**].



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



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



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

Figure 6.8.: Comparison of benchmarks running recorded **cube-64** communication pattern between two sockets within a node

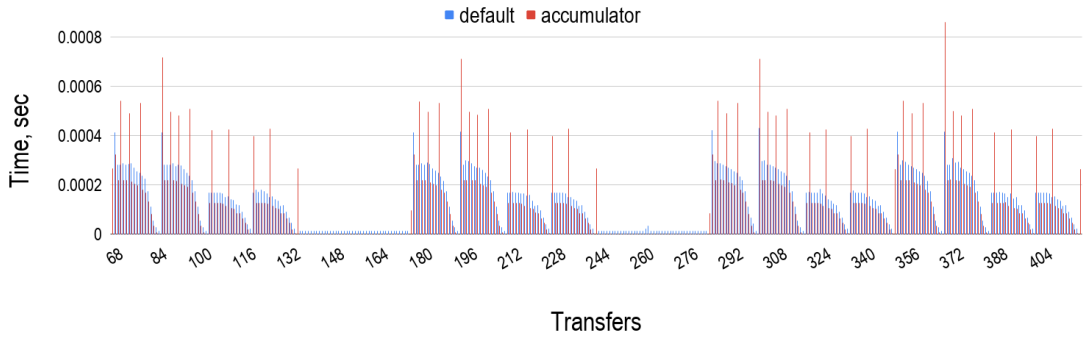


Figure 6.9.: Comparison of **BM1** benchmark with the default approach running recorded **cube-64** communication pattern between two nodes

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

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

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

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

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

6.5. Conclusion

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

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

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

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

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

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

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

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

Appendices

A. Choice of a Sparse Direct Solver Library

MPI	MUMPS	PaStiX	SuperLU
1	4.58E-02	5.60E-02	4.64E+00
2	4.31E-02	5.14E-02	1.89E+00
3	4.51E-02	5.28E-02	1.22E+00
4	4.61E-02	5.64E-02	9.13E-01
5	4.92E-02	5.97E-02	7.70E-01
6	5.37E-02	6.14E-02	6.04E-01
7	5.42E-02	6.51E-02	crashed
8	5.41E-02	6.60E-02	4.81E-01
9	5.69E-02	6.84E-02	4.35E-01
10	5.86E-02	7.22E-02	4.08E-01

MPI	MUMPS	PaStiX	SuperLU
11	5.93E-02	8.97E-02	crashed
12	6.07E-02	9.20E-02	3.61E-01
13	6.26E-02	8.25E-02	crashed
14	6.28E-02	9.75E-02	crashed
15	6.43E-02	1.03E-01	3.05E-01
16	6.55E-02	1.05E-01	2.99E-01
17	6.61E-02	9.46E-02	crashed
18	6.73E-02	1.24E-01	2.65E-01
19	6.84E-02	1.14E-01	crashed
20	7.02E-02	1.32E-01	2.60E-01

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

MPI	MUMPS	PaStiX	SuperLU
1	1.55E+02	6.44E+01	time-out
2	6.28E+01	4.84E+01	time-out
3	5.06E+01	5.02E+01	time-out
4	4.17E+01	4.50E+01	time-out
5	2.52E+01	3.98E+01	time-out
6	2.58E+01	4.29E+01	time-out
7	2.65E+01	4.30E+01	time-out
8	2.59E+01	3.73E+01	time-out
9	1.95E+01	4.08E+01	time-out
10	1.91E+01	3.81E+01	time-out

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

MPI	MUMPS	PaStiX	SuperLU
1	1.52E+01	1.61E+01	crashed
2	1.13E+01	1.13E+01	crashed
3	1.00E+01	1.03E+01	crashed
4	9.29E+00	1.05E+01	crashed
5	8.85E+00	9.84E+00	crashed
6	8.43E+00	8.99E+00	crashed
7	8.64E+00	9.69E+00	crashed
8	8.70E+00	9.12E+00	crashed
9	8.91E+00	8.94E+00	crashed
10	8.76E+00	9.26E+00	crashed

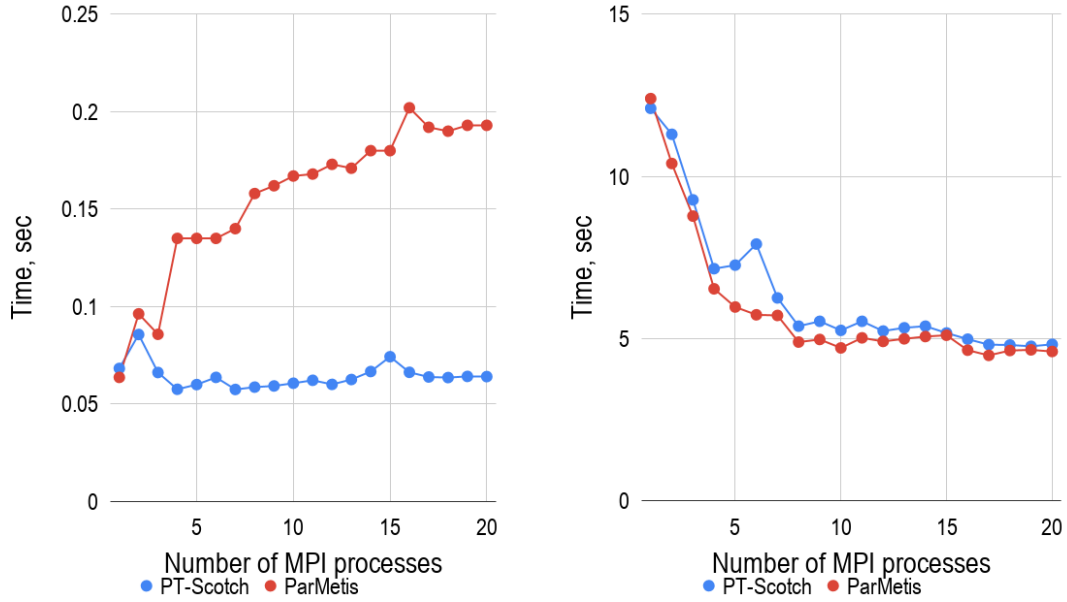
MPI	MUMPS	PaStiX	SuperLU
11	8.62E+00	9.09E+00	crashed
12	8.53E+00	8.92E+00	crashed
13	8.44E+00	9.13E+00	crashed
14	8.52E+00	9.00E+00	crashed
15	8.54E+00	9.19E+00	crashed
16	8.56E+00	9.05E+00	crashed
17	8.65E+00	9.12E+00	crashed
18	8.62E+00	8.96E+00	crashed
19	8.66E+00	9.30E+00	crashed
20	8.66E+00	9.16E+00	crashed

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

B. Choice of Fill Reducing Reordering

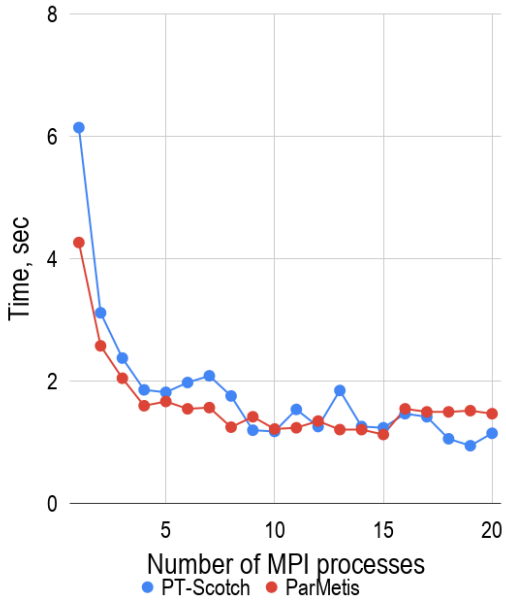
place for figure B.1

place for figure B.2

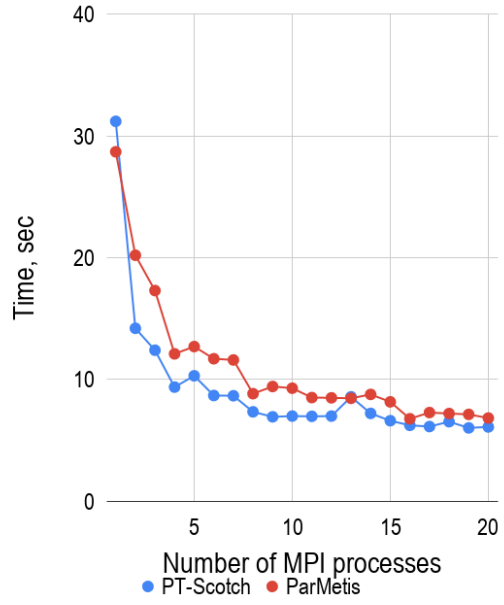


(a) cube-5

(b) cube-645

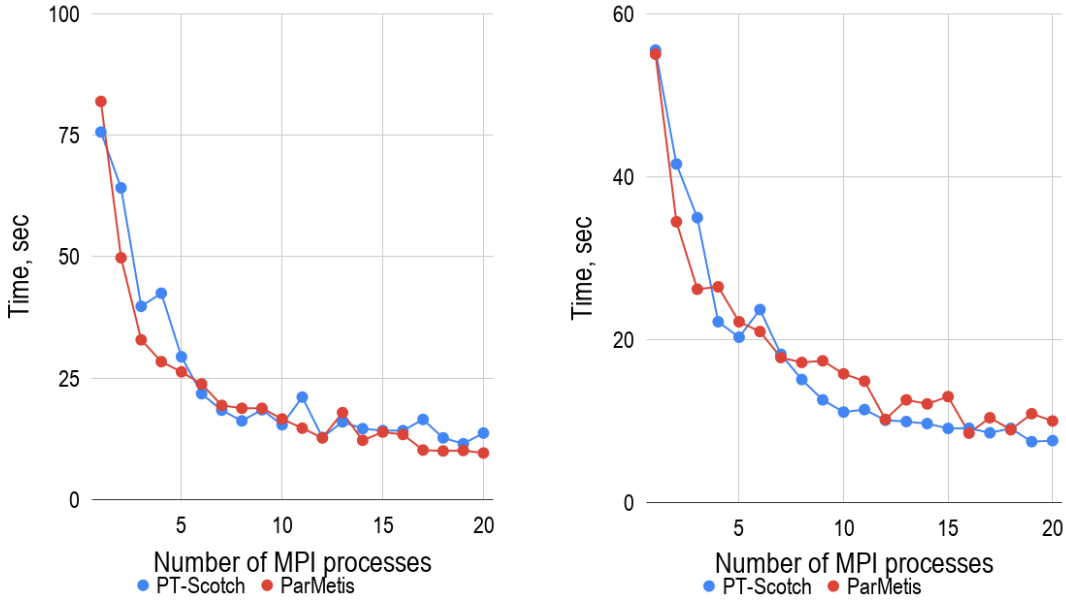


(c) cant



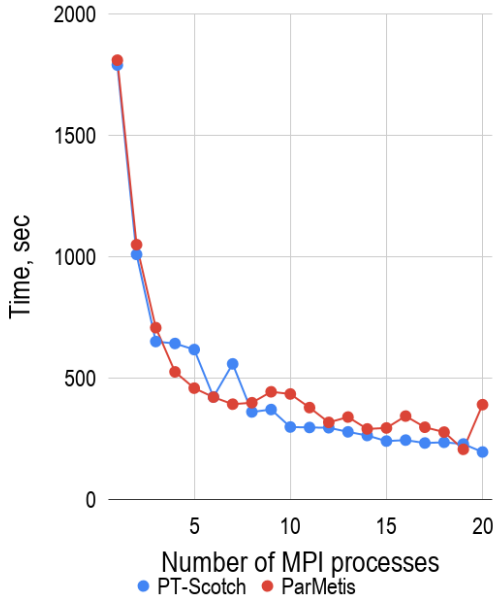
(d) memchip

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

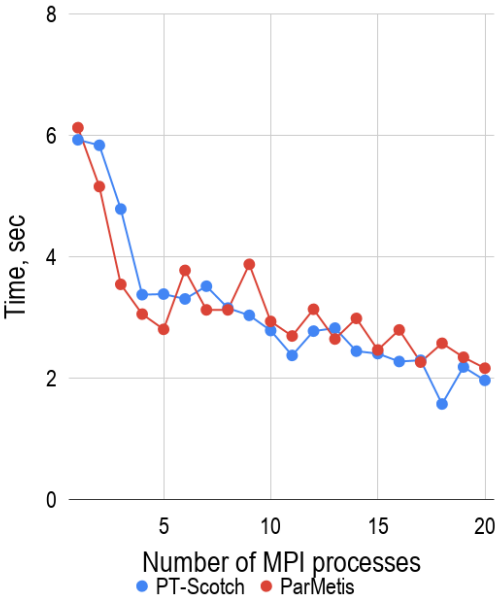


(a) torso3

(b) consph



(c) CurlCurl_3



(d) x104

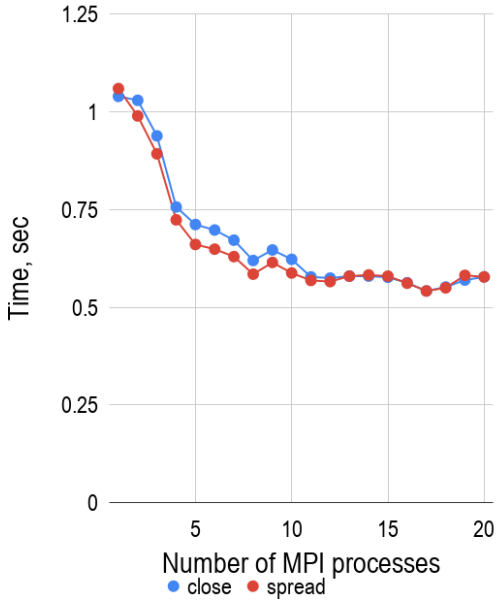
Figure B.2.: Comparison of different fill-reducing algorithms

C. MUMPS: Process Pinning

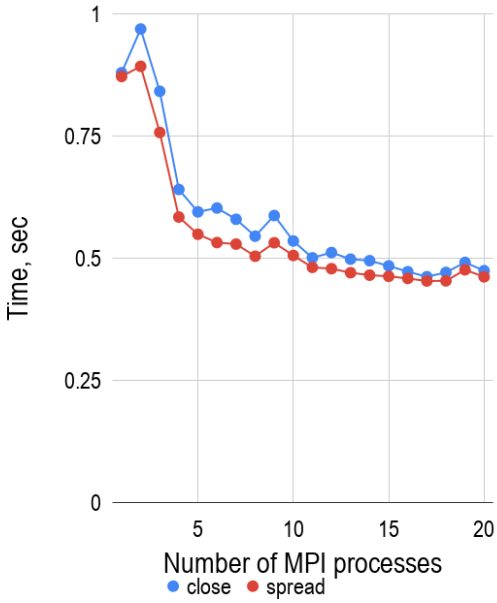
replace cube-64 by Geo matrix

place for figure C.1

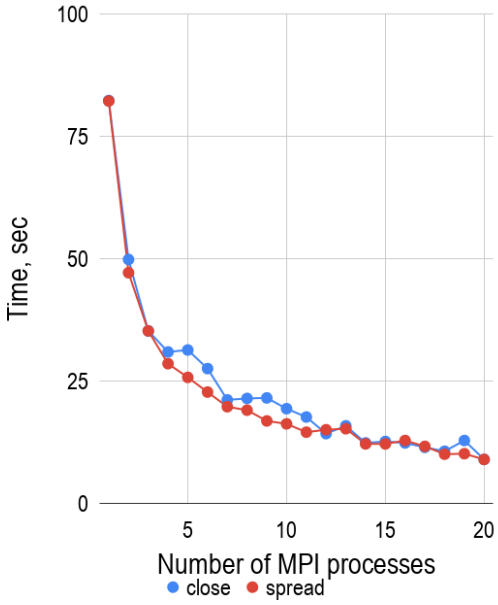
place for figure C.2



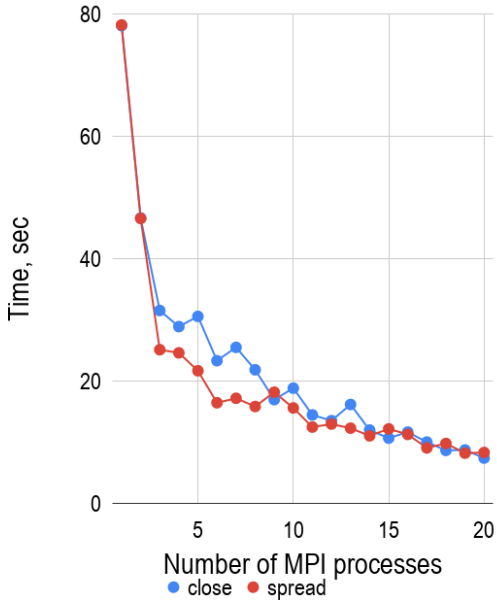
(a) HW1 - cube-64



(b) HW2 - cube-64

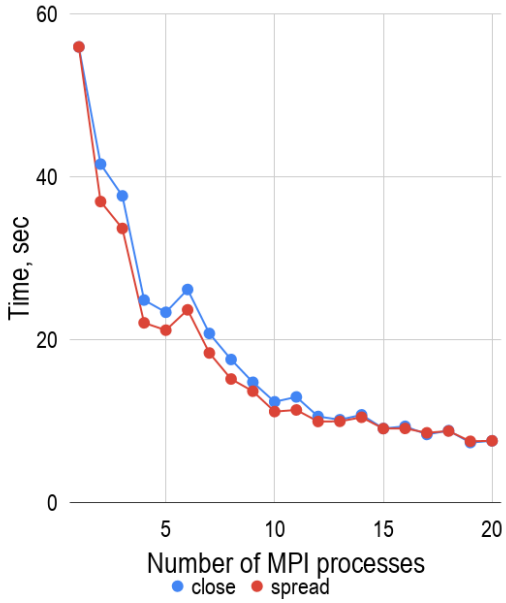


(c) HW1 - torso3

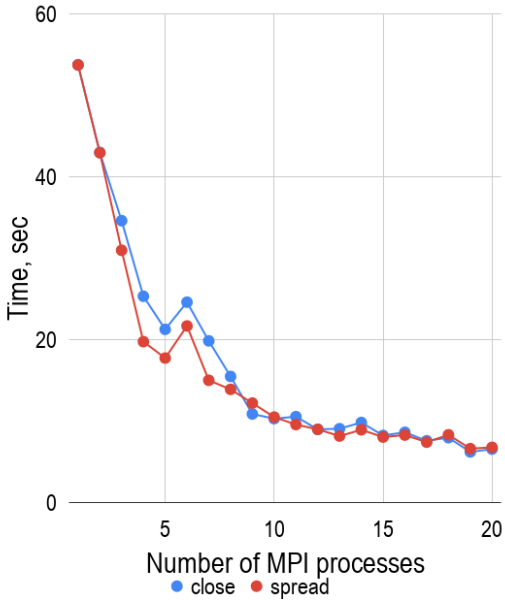


(d) HW2 - torso3

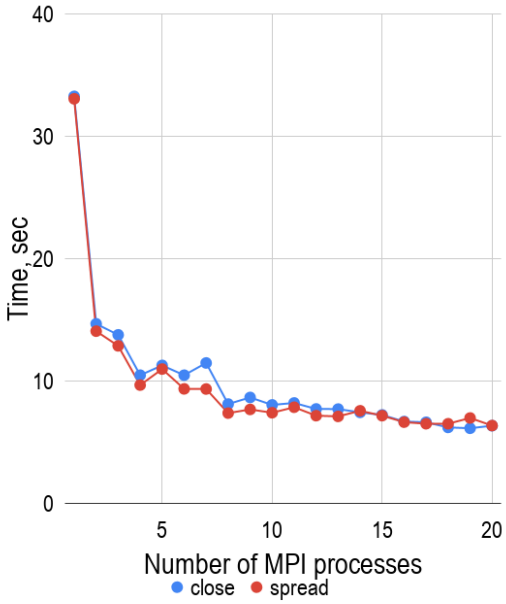
Figure C.1.: Comparison of *close* and *spread* pinning strategies



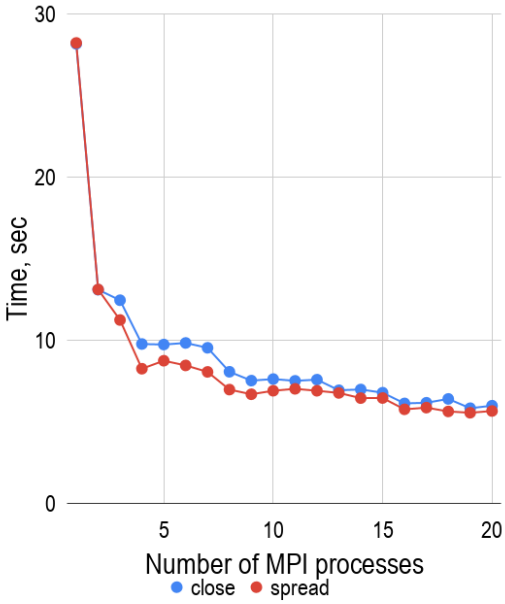
(a) HW1 - consph



(b) HW2 - consph



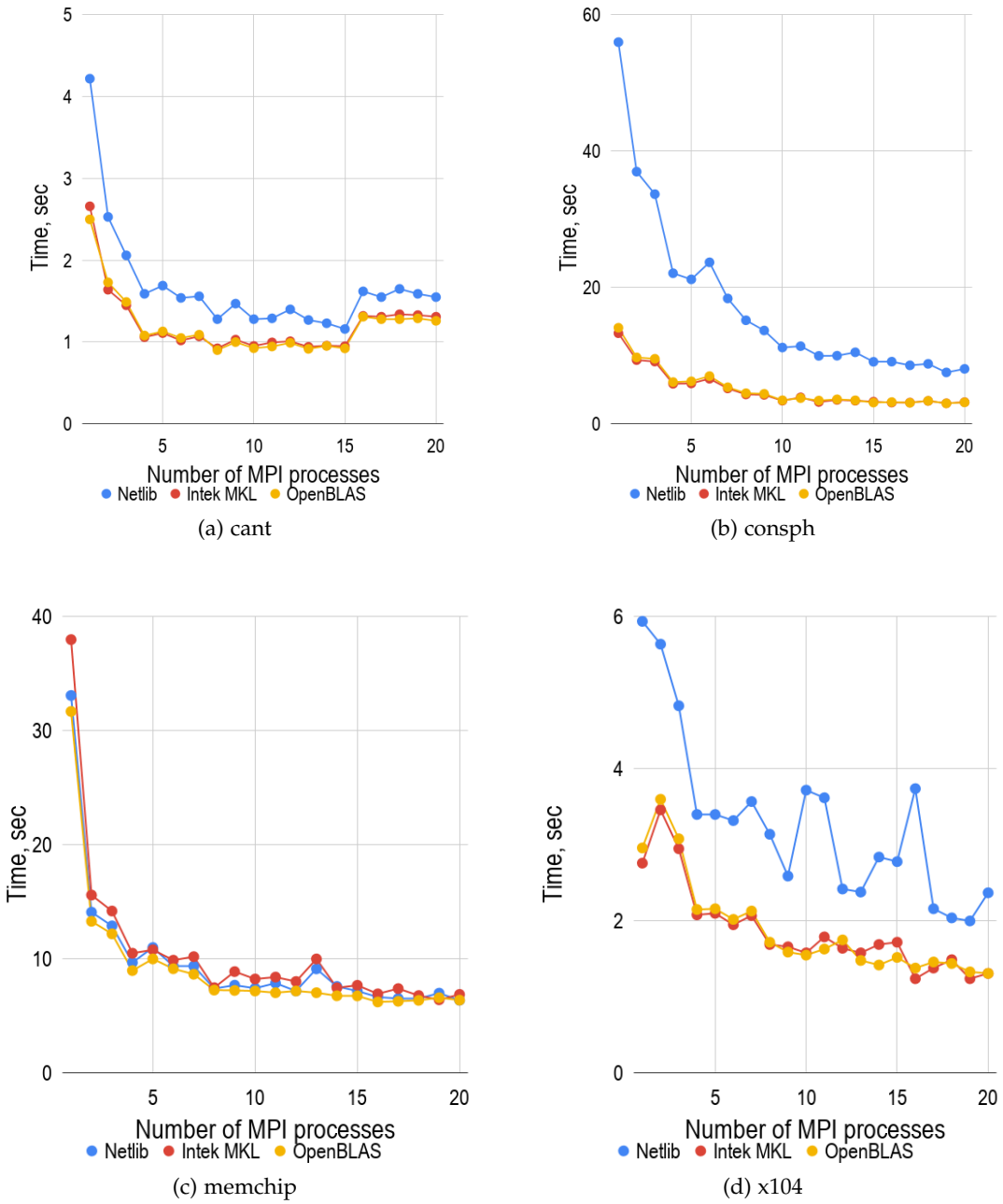
(c) HW1 - memchip_3



(d) HW2 - memchip

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

D. Choice of BLAS Library



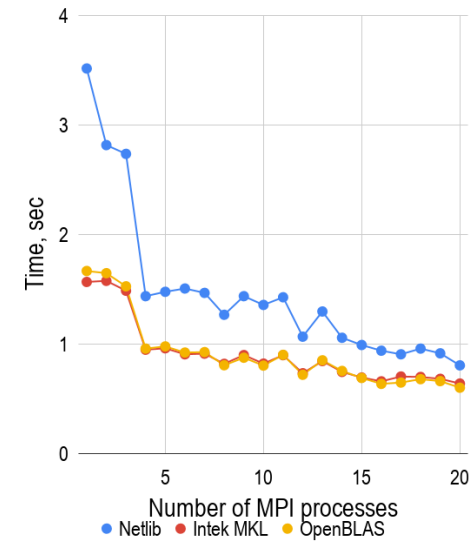
(a) cant

(b) consph

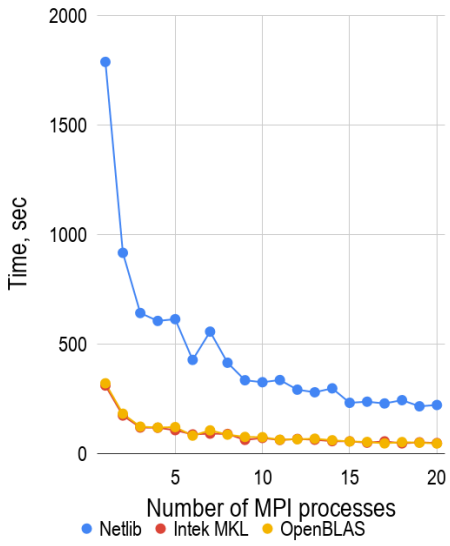
(c) memchip

(d) x104

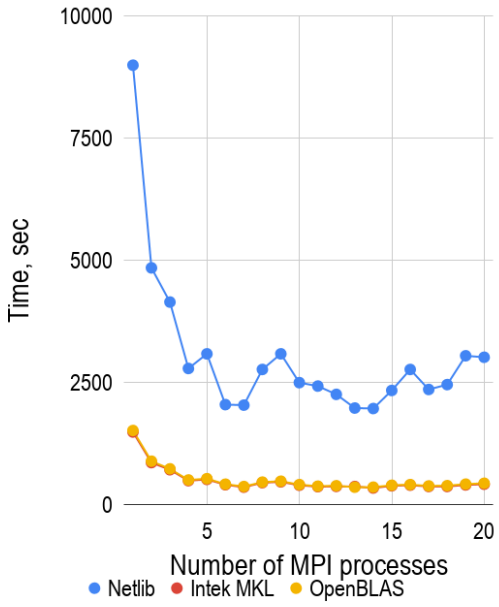
Figure D.1.: MUMPS: comparison of different BLAS libraries with using both GRS and SuiteSparse matrix sets on HW1 machine



(a) pkustk10



(b) CurlCurl_3



(c) Geo_1438

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

List of Figures

List of Tables