Scuola Internazionale Superiore di Studi Avanzati

Mathematics Area

Master in High Performance Computing

Towards Exascale BEM simulations: Hybrid Parallelisation Strategies for Boundary Element Methods

Advisor:

Dott. Luca Heltai

Co-Advisor:

Dott. Andrea Mola

Author:

Nicola Giuliani

Were it so easy.

Contents

Chapter 1

Introduction

Many fields of engineering benefit from an accurate and reliable solver for the Laplace equation. Such an equation is able to model many different phenomena, and is at the base of several multi-physics solvers.
For example, in nautical engineering, since the Navier–Stokes system has an extremely high computational cost, many reduced order models are often used to predict ship performance. Under the assumption of incompressible fluid and irrotational flow it is possible to recover a flow field by simply imposing mass conservation, which simplifies to a Laplace equation.
Morevore, the deep theoretical background that surrounds this equation, makes it ideal as a benchmark to test new numerical softwares.
Over the last decades such equation has often been solved through its Boundary Integral formulation, leading to Boundary Element Methods. What makes such methods appealing with respect to a classical Finite Element Method is the fact that they only require discretisation of the boundary.
The purpose of the present work is to develop an efficient and optimize BEM for the Laplace equation, designed around the architecture of modern CPUs.
In 1975 Gordon Moore predicted that the number of transistor, and consequently the computational possibilities, in a dense integrated circuit would double every two years. In 2015 Intel CEO has confirmed that this tendency has only slowed down sligthly, by making the doubling period closer to two and a half year. Originally processors makers kept developing architectures with higher clock rates, but to manage CPU-Power dissipation they started favouring multicore chip designs.
If we consider a modern High Performance computing facility we need to face different level of parallelism. Firstly we may use different computational nodes that does not share any logical memory, then on a single node we usually have a multicore design, and finally we may need to use an accellerator, as a Graphic Power Unit. Therefore, it is becoming more and more important to include the newest parallel programming technique to develop an efficient code and fully exploits the

possibilities of the newest architectures.
Boundary Element Method can be quite optimally parallelised using the classical Message Passing Interface. The first goal of the present work is to develop an efficient parallel BEM for the Laplace equation. We believe that such an implementation may lead to huge time reductions of a single computation. In order to develop a reliable, efficient and general code we make good use of existing efficient library: we couple METIS [6], `deal.II` [2] and Trilinos[11] to split the computational effort over different MPI processors. We use instead Intel Threading Building Block to manage multicore architectures. Similar couplings have already been successfully applied to achieve high computational efficiency in fluid dynamics, as demonstrated in the state of the art ASPECT code [12].
An inevitable disadvantage of BEMs is that their computational cost depends quadratically on the number of unknowns. Thus, in the last decades a lot of methods have been developed to reduce such a dependency. In particular, we consider the Fast Multiple Method. This algorithm was originally thought for particle dynamics, but it has been successfully applied to BEMs by Greengard [9]. The parallelisation of the FMM is non-trivial. Greengard himself proposed a pure multithreaded version in 1990 [8]. In more recent years, Yokota *et alt.* developed a parallel version of the algorithm using hybrid coding techniques [20]. In particular, it has been assessed that FMMs can behave well in the new Exascale era [3].
Therefore, the second goal of the present work is the coding of a parallel Fast Multipole Method. Given the relatively small sizes of the problems, and the particular industrial constraints we usually address, existing libraries are not flexible enough, and we provide a new version of the Fast Multiple Method through the usage of the same libraries we successfully employed in our parallel Boundary Element Method: METIS, `deal.II` and Trilinos.
We highlight that the increased algorithm complexity, with respect to a classical BEM, makes it necessary to use hybrid multicore multiprocessor parallelisation techniques. In particular, we have chosen to use MPI to handle different processors and Intel Threading Building Blocks to take care of different threads.
We make extensive use of parameter files in both the Boundary Element and Fast Multipole Methods, through the newborn `deal2lkit` library, [17], to properly handle the parameters.
We benchmark our application considering mixed boundary conditions, and we analyse both strong and weak scalability performances. In particular we compare standard Boundary Element Method and our Fast Multipole implementation, and we discuss optimality conditions for the hybrid BEM-FMA algorithm.

Chapter 2

BEM for the Laplace Equation

This chapter focuses on the Laplace equation and its discretization via the Boundary Element Method. The Laplace equation is first introduced along with a set of boundary conditions which ensure the well posedness of the partial differential problem in 3D domains. The Laplace equation is then reformulated into its boundary integral form, and finally we present the discretization strategy. We follow the formalism of [7].

2.1 Basic notation and governing equations

The Laplace equation can be solved in a closed bounded domain called Ω. Dirichlet and Neumann boundary conditions are imposed on the portions Γ_D, and Γ_N of $\partial\Omega$, such that $\Gamma_D \bigcup \Gamma_N = \partial\Omega$, $\Gamma_D \bigcap \Gamma_N = \oslash$, and $|\Gamma_D| \neq \oslash$.

$$\Delta\phi = 0 \qquad \text{in } \Omega \tag{2.1a}$$

$$\frac{\partial\phi}{\partial n} = h(\mathbf{x}) \qquad \text{on } \Gamma_N \tag{2.1b}$$

$$\phi = g(\mathbf{x}) \qquad \text{on } \Gamma_D. \tag{2.1c}$$

Another possibility is to considered the solution over an unbounded region, namely the space surrounding a region Ω. Thus the Laplace equation is solved on $\mathbb{R}^n\backslash\Omega$, introducing the same decomposition of $\partial\Omega$ seen before

$$\Delta\phi = 0 \qquad \text{in } \mathbb{R}^n\backslash\Omega \tag{2.2a}$$

$$\frac{\partial\phi}{\partial n} = h(\mathbf{x}) \qquad \text{on } \Gamma_N \tag{2.2b}$$

$$\phi = g(\mathbf{x}) \qquad \text{on } \Gamma_D \tag{2.2c}$$

$$\lim_{|r|\to\infty} \phi(r) = \phi_\infty. \tag{2.2d}$$

Where $h(\mathbf{x})$, and $g(\mathbf{x})$ represent the boundary values for ϕ, and $\partial\phi/\partial n$ respectively. The presence of Dirichlet boundary conditions ensures uniqueness to the solution.

2.2 Boundary Integral Formulation of the governing equations and BEM

Following [4, 7], we exploit the second Green identity and reformulate Laplace equation (2.1a) as

$$\int_\Omega (-\Delta\phi) G \,\mathrm{d}x = \int_\Omega (-\Delta G)\phi \,\mathrm{d}x + \int_\Gamma \frac{\partial \phi}{\partial \mathbf{n}} G \,\mathrm{d}s - \int_\Gamma \phi \frac{\partial G}{\partial \mathbf{n}} \,\mathrm{d}s = 0, \tag{2.3}$$

where $\mathbf{n}$ is the outward normal to Γ while G is the so called free-space Green function, or fundamental solution of the Laplace equation, also known as the Rankine source, namely:

$$G(\mathbf{x}-\mathbf{y}) = \frac{1}{4\pi}\frac{1}{|\mathbf{x}-\mathbf{y}|}. \tag{2.4}$$

The Rankine source satisfies, in a distributional sense, the equation

$$-\Delta G(\mathbf{x}-\mathbf{y}) = \delta(\mathbf{x}-\mathbf{y}). \tag{2.5}$$

Exploiting this property, equation (2.3) can be rewritten as

$$\phi(\mathbf{x}) = \int_\Gamma G(\mathbf{x}-\mathbf{y})\frac{\partial \phi}{\partial n}(\mathbf{x}) \,\mathrm{d}s_y - \int_\Gamma \phi(\mathbf{x})\frac{\partial G}{\partial n}(\mathbf{x}-\mathbf{y}) \,\mathrm{d}s_y \qquad \forall \mathbf{x}\in\Omega. \tag{2.6}$$

We point out that equation (2.6) allows for the computation of the potential ϕ in any point $\mathbf{x}$ in the domain Ω if $\phi(\mathbf{x})$ and its normal derivative $\frac{\partial\phi}{\partial n}(\mathbf{x})$ are known on the boundary Γ. If we move the point $\mathbf{x}$ towards the boundary Γ, the kernels $G(\mathbf{x}-\mathbf{y})$ and $\frac{\partial G}{\partial n}(\mathbf{x}-\mathbf{y})$ become weakly singular (but integrable) and singular respectively. Considering the Cauchy Principal Value (CPV) of the singular integral, we can write the boundary integral form of the original problem as

$$\begin{aligned}
\alpha(\mathbf{x})\phi(\mathbf{x}) &= \int_\Gamma G(\mathbf{x}-\mathbf{y})\tfrac{\partial\phi}{\partial n}(\mathbf{x})\,\mathrm{d}s_y - \int_\Gamma^{PV} \phi(\mathbf{x})\tfrac{\partial G}{\partial n}(\mathbf{x}-\mathbf{y})\,\mathrm{d}s_y && \text{on } \Gamma && \text{(2.7a)}\\
\frac{\partial\phi}{\partial n} &= -\mathbf{v}_\infty\cdot\mathbf{n} && \text{on } \Gamma_{body} && \text{(2.7b)}\\
\frac{\partial\phi}{\partial n} &= -\frac{U_\infty^2}{g}\frac{\partial^2\phi}{\partial x^2} && \text{on } \Gamma_{fs} && \text{(2.7c)}\\
\frac{\partial\phi}{\partial n} &= 0 && \text{on } \Gamma_{out}\cup\Gamma_{tank} && \text{(2.7d)}\\
\phi &= 0 && \text{on } \Gamma_{in}, && \text{(2.7e)}
\end{aligned}$$

where the coefficient $\alpha(\mathbf{x})$ appearing in the left hand side of equation (2.7) is obtained from the CPV evaluation of the singular integral on the right hand side, and represents the fraction of solid angle with which the domain Ω is seen from the boundary point $\mathbf{x}$. Equation (2.7a) is also known as Boundary Integral Equation (BIE).

2.3 Discretization procedure for BIE

A discretization of the Laplace problem based on the BIE obtained in the previous chapter is here discussed. It leads to a linear system whose solution is an approximated solution of the original Laplace problem.
Boundary integral formulations only involve functions defined on the boundary Γ of the computational domain Ω. In order to solve numerically such a problem, it suffices to provide an approximation of the surfaces making Γ and to define finite dimensional functional spaces *on the boundary only.*

We use standard Lagrangian finite element spaces on Γ to define both the geometry and the basis functions for ϕ and $\frac{\partial\phi}{\partial n}$. These basis functions are of interpolatory type, in the sense that they are defined through a set of *support points* $\mathbf{x}_j$ where they may only be zero or one, and each basis function has value one in a unique support point.

The two unknowns of our problem, namely ϕ and $\frac{\partial\phi}{\partial n}$, have different mathematical characteristics. While the potential ϕ is a continuos function on Γ, $\frac{\partial\phi}{\partial n}$ depends on the normal vector $\mathbf{n}$ which is discontinuos across the edges of Γ. In order to provide an accurate numerical solution, it is crucial to choose a different numerical representation for ϕ and $\frac{\partial\phi}{\partial n}$. The approximation of the boundary Γ of the domain should be continuos, and we represent it through the same basis functions that we use for the potential ϕ. Our approximation is isoparametric in both the unknowns ϕ, $\frac{\partial\phi}{\partial n}$. We refer to this type of approximation as Isoparametric BEM. The unknowns of the problem are the values of ϕ and $\frac{\partial\phi}{\partial n}$ on the respective set of support points.

The approximation of the geometry of Γ and the choice of the proper spaces are exploited at the discrete level by an approximation procedure divided in five main steps:

Computational mesh creation: introduce a computational mesh which is a regular decomposition Γ_h of the boundary Γ made of quadrilateral cells (here *regular* means that any two cells K, K' only intersects on common faces, edges or vertices);

Definition of the discrete spaces: introduce two (*a priori* independent) finite dimensional spaces V_h and Q_h, [1] defined on Γ_h, such that

$$V_h := \left\{\phi_h \in C^0(\Gamma_h) : \phi_{h|K} \in \mathbb{Q}^r(K),\ K \in \Gamma_h\right\} \equiv \operatorname{span}\{\psi_i\}_{i=1}^{N_V} \tag{2.9a}$$

$$Q_h := \left\{\gamma_h \in L^2(\Gamma_h) : \gamma_{h|K} \in \mathbb{Q}^s(K),\ K \in \Gamma_h\right\} \equiv \operatorname{span}\{\omega_i\}_{i=1}^{N_Q}, \tag{2.9b}$$

where on each cell K, located on the boundary, $\phi_{h|K}, \gamma_{h|K}$ are polynomial functions of degree r and s respectively, in each coordinate direction. The corresponding Lagrangian basis functions of the spaces V_h and Q_h are denoted ψ_i and ω_i respectively;

Collocation of the Boundary Integral Equations: replace the continuous functions ϕ and $\frac{\partial\phi}{\partial n}$ by their numerical approximations ϕ_h and γ_h, which represent the discretised potential and potential normal derivative respectively in V_h and Q_h, and collocate the BIE on the correct support ponts on the boundary Γ_h;

Imposition of the boundary conditions: compute the boundary condition for ϕ_h and γ_h, and impose them int the system;

Computation of a proper preconditioner: compute the preconditioner taking into account the boundary conditions;

Solution of the linear system: the procedure above leads to a (dense) linear system which is solved iteratively making use of the preconditioner.

[1] For the integrals in equation (2.7a) to be bounded, ϕ and $\frac{\partial\phi}{\partial n}$ must lie in the spaces V and Q, defined as

$$V := \left\{\phi \in H^{\frac{1}{2}}(\Gamma)\right\} \tag{2.8a}$$

$$Q := \left\{\gamma \in H^{-\frac{1}{2}}(\Gamma)\right\}, \tag{2.8b}$$

where $\Gamma = \partial\Omega$. We recall that $H^{\frac{1}{2}}(\Gamma)$ can be defined as the space of traces on Γ of functions in $H^1(\Omega)$, while $H^{-\frac{1}{2}}(\Gamma)$ is its dual space. The spaces V_h and Q_h are constructed as conforming finite dimensional subspaces of V and Q respectively.

2.3.1 Geometry and variable representations

When the Lagrangian basis functions for V_h and Q_h are given by ψ_i and ω_i respectively, a finite dimensional approximation of the unknowns ϕ and $\frac{\partial\phi}{\partial n}$ reads

$$\phi\Big|_{\Gamma}(\mathbf{x}) \sim \phi_h(\mathbf{x}_h) = \sum_{i=1}^{N_V} \psi_i(\mathbf{x}_h)\phi_i \qquad \mathbf{x}\in\Gamma, \quad \mathbf{x}_h \in \Gamma_h \tag{2.10}$$

$$\frac{\partial\phi}{\partial n}\Big|_{\Gamma}(\mathbf{x}) \sim \gamma_h(\mathbf{x}_h) = \sum_{i=1}^{N_Q} \omega_i(\mathbf{x}_h)\gamma_i \qquad \mathbf{x}\in\Gamma, \quad \mathbf{x}_h \in \Gamma_h. \tag{2.11}$$

Here $\{\psi_i\}$ and $\{\omega_i\}$ have the cardinality N_V and N_Q of the corresponding finite element spaces, and are defined *only* on the approximated boundary Γ_h. In particular, they can be conveniently expressed in terms of a *local coordinate system* (u, v) on each element K, by introducing for both V_h and Q_h, the set of *local basis functions*, a *mapping* from a reference element to the real geometric element K and a *local to global* numbering map k_i (see Figure 2.1).

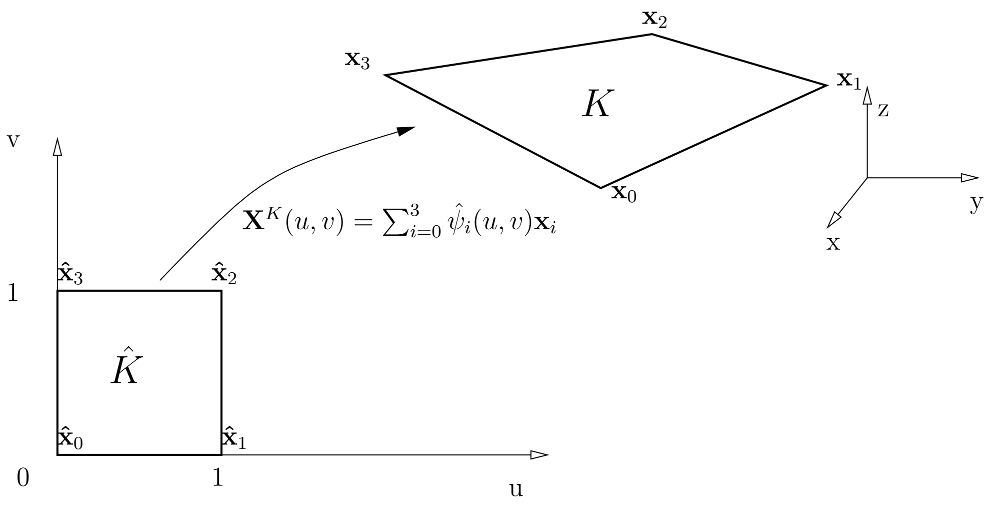

Figure 2.1: Transformation from reference to real cell. In this example we have considered a linear continuous approximation for the geometry, with support points on the vertices of the quadrilateral. The BEM is isoparametric because the geometry is described by the same finite element approximation of the potential ϕ.

In particular, we can define the approximation of the global basis functions restricted on $K \subset \Gamma_h$ as functions of the reference variables u and v on K:

$$\mathbf{X}^K(u,v) = \sum_{m=0}^{3} \hat{\psi}_m(u,v)\mathbf{x}_{i_m} \qquad i_m \in \{1,\dots,N_V\} \tag{2.12a}$$

$$\hat{\psi}_m(u,v) = \psi_{i_m^K}(\mathbf{X}^K(u,v)), \qquad i_m^K \in \{1,\dots,N_V\}, \quad m \in \{0,1,2,3\} \tag{2.12b}$$

$$\hat{\omega}_m(u,v) = \omega_{a_m^K}(\mathbf{X}^K(u,v)), \qquad a_m^K \in \{1,\dots,N_Q\}, \quad m \in \{0,1,2,3\}. \tag{2.12c}$$

2.3.2 BEM: collocation technique

A discrete form of the BIE (2.7a) is obtained by replacing the continuous solutions ϕ and $\frac{\partial \phi}{\partial n}$ by their finite dimensional approximations ϕ_h and γ_h, and imposing the original boundary integral equation at a sufficient number of *collocation points*. Such collocation points are placed in correspondance with the N_V support points of the V_h space and, with the N_Q support points of the Q_h space. Thus we obtain a system of $N_V + N_Q$ algebraic equations which reads respectively,

$$\begin{bmatrix} \mathbf{H^V} & \mathbf{D^V} \\ \mathbf{H^Q} & \mathbf{D^Q} \end{bmatrix} \begin{Bmatrix} \boldsymbol{\phi} \\ \boldsymbol{\gamma} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}, \tag{2.13}$$

where

$$\boldsymbol{\phi} = \{\phi_1,\dots,\phi_{N_V}\}, \boldsymbol{\gamma} = \{\gamma_1,\dots,\gamma_{N_Q}\}.$$

are the vectors containing the unknown values of the the approximated functions ϕ_i and γ_i at each collocation point. In the block system (2.13), the matrix rows in the top blocks are the ones obtained collocating the BIE on the N_V support points corresponding to the potential degrees of freedom, while the matrix rows in the bottom blocks are obtained using the N_Q support points of the normal derivative as collocation points.

All integrations are performed on a planar reference domain, i.e. we assume that each element K_i of Γ_h as a transformation of the reference boundary element $\hat{K} := [0,1]^2$, as depicted in Figure 2.1.The integrations are performed after a change of variables from the real element K_i to the reference element $\hat{K}$.
Given a collocation point $\mathbf{P}_i$ of the potential ϕ, the block matrix entries read

$$H_{ij}^V = \alpha(\mathbf{P}_i) + \sum_{K=1}^{M} \int_0^1 \int_0^1 \frac{\partial G}{\partial n}(\mathbf{P}_i - \mathbf{X}^K(u,v))\hat{\psi}_j(u,v)J^K(u,v)\,\mathrm{d}u\,\mathrm{d}v, \tag{2.14a}$$

$$D_{ib}^V = \sum_{K=1}^{M} \int_0^1 \int_0^1 G(\mathbf{P}_i - \mathbf{X}^K(u,v))\hat{\omega}_b(u,v)J^K(u,v)\,\mathrm{d}u\,\mathrm{d}v, \tag{2.14b}$$

while if $\mathbf{Q}_a$ is a γ collocation point, we have

$$H^Q_{aj} = \alpha(\mathbf{Q}_a) + \sum_{K=1}^{M} \int_0^1 \int_0^1 \frac{\partial G}{\partial n}(\mathbf{Q}_a - \mathbf{X}^K(u,v))\hat{\psi}_j(u,v) J^K(u,v)\,\mathrm{d}u\,\mathrm{d}v, \tag{2.14c}$$

$$D^Q_{ab} = \sum_{K=1}^{M} \int_0^1 \int_0^1 G(\mathbf{Q}_a - \mathbf{X}^K(u,v))\hat{\omega}_b(u,v) J^K(u,v)\,\mathrm{d}u\,\mathrm{d}v, \tag{2.14d}$$

where M is the total number of elements K of the triangulation Γ_h, J^K is the Jacobian of the mapping $\mathbf{X}^K$ in the K-th element and the indices i, j run from one to N_V, while the indices a, b run from one to N_Q.

Due to the possible presence of singular kernels, the integrals in equations (2.14) require special treatment. In this work, standard Gauss quadrature rules are used for panels not containing singularities while for panels containing singular kernel values, we use Telles' quadrature rule [18].

Block system (2.13) is ill posed and cannot be used to obtain a numerical solution of problem (2.7), since i) it does not contain any information about boundary conditions and ii), it might have identical rows if some support points of ϕ and γ coincide. Notice that usually, on any given region of Γ, either ϕ is known and $\gamma = \frac{\partial \phi}{\partial n}$ is unknown, or the opposite.

2.3.3 Imposition of the Boundary Condition

If a Neumann problem is considered, the normal derivative is known so the system can be rewritten as follows

$$(\mathbf{H^V})\boldsymbol{\phi} = -\mathbf{D^V}\boldsymbol{\gamma} = \mathbf{b}, \tag{2.15}$$

where the i-th element of vector $\boldsymbol{\gamma}$ is $\overline{\gamma}_i = h(\mathbf{x}_i)$. In case of Dirichlet problem instead, the value of $\boldsymbol{\phi}$ is known so the system is rewritten as follows

$$\mathbf{D^Q}\boldsymbol{\gamma} = -\mathbf{H^Q}\boldsymbol{\phi} = \mathbf{c}, \tag{2.16}$$

where the i-th element of $\boldsymbol{\phi}$ is $\overline{\phi_i} = g(\mathbf{x}_i)$.
When the more general case, with mixed boundary condition, is considered, a typical approach is, see [14],

$$\boldsymbol{\phi} = \overline{\boldsymbol{\phi}}_D + \boldsymbol{\phi}_N \qquad \boldsymbol{\gamma} = \boldsymbol{\gamma}_D + \overline{\boldsymbol{\gamma}}_N,$$

where

$$\overline{\phi}_{iD} = \begin{cases} \overline{\phi}_i & \mathbf{x}_i \in \Gamma_D \\ 0 & \mathbf{x}_i \in \Gamma_N \end{cases} \qquad \phi_{iN} = \begin{cases} 0 & \mathbf{x}_i \in \Gamma_D \\ \phi_i & \mathbf{x}_i \in \Gamma_N \end{cases} \tag{2.17a}$$

and

$$\gamma_{iD} = \begin{cases} \gamma_i & \mathbf{x}_i \in \Gamma_D \\ \\ 0 & \mathbf{x}_i \in \Gamma_N \end{cases} \qquad \overline{\gamma}_{iN} = \begin{cases} 0 & \mathbf{x}_i \in \Gamma_D \\ \\ \overline{\gamma}_i & \mathbf{x}_i \in \Gamma_N . \end{cases} \tag{2.17b}$$

As seen before, the overlined symbol indicates known values, prescribed by imposing the boundary conditions. The final form of the linear problem to be solved reads

$$(\mathbf{H^V})\boldsymbol{\phi}_N - \mathbf{D^Q}\boldsymbol{\gamma}_D = \mathbf{D^V}\overline{\boldsymbol{\gamma}}_N - (\mathbf{H^Q})\overline{\boldsymbol{\phi}}_D, \tag{2.18}$$

which can be recast in the more familiar form

$$\mathbf{Mu} = \mathbf{b} \tag{2.19}$$

where

$$\mathbf{u} = \boldsymbol{\phi}_N + \boldsymbol{\gamma}_D, \qquad \mathbf{b} = \mathbf{D^V}\overline{\boldsymbol{\gamma}}_N - (\mathbf{H^Q})\overline{\boldsymbol{\phi}}_D,$$

and where $\mathbf{M}$ is defined by

$$M_{ij} = \begin{cases} H^V_{ij} & \mathbf{x}_j \in \Gamma_N \\ D^Q_{ij} & \mathbf{x}_j \in \Gamma_D . \end{cases} \tag{2.20}$$

2.3.4 Preconditioner for the Linear System

We have chosen to use an iterative solver for the problem (2.19). Once we have defined the system matrix as in (2.20) we can build a proper preconditioner. The preconditioner should be an approximation, in spectral sense, of the inverse of the system matix, M^{-1}. In this way we can reduce the number of iterations needed to the solver to converge. This is due to the fact that Krylov method iterations are strictly linked to the bandwith of the matrix spectrum. Ideally if we use as preconditioner P the matrix inverse M^{-1} the preconditioend matrix would have all its eigenvalues equal 1 and we would need only a step to obtain the solution. Clearly we can't afford the computation of an inverse of a dense matrix, thus we have chosen to compute an Icomplete Gauss LU factorisation of the system matrix. In this way we seek for

$$LU \approx M \tag{2.21}$$

In this way we can use its inverse to precondition our system as

$$P = (LU)^{-1}. \tag{2.22}$$

Such an approximation is obtained computing a setting a bandwith and extracting a diagonal approximation from the system matrix M. Clearly P spectrum is closer to the ideal preconditioner as we increase the bandwith but we increase also the computational cost of such a matrix. In the present work we have chosen to keep fix the bandwith, in this way we have a coarser representation of the preconditioner as the system size increases.

Chapter 3

Parallel BEM

In this chapter we describe the parallelisation of the Boundary Element Method. Given the particular resolution strategy we have depicted in Section 2.3 we have chosen to follow the MPI paradigm.

3.1 Implementation Principles

The existing code implements a Boundary Element Method to solve the Laplace equation though its Boundary Integral representation. The code is primarily developed by mathematicians and engineers at SISSA. We want it to be based on the principles that guides a very well developed scientific code, ASPECT, [12]. These basic principles are:

- *Usability and extensibility*: At the time being the code only solves a BEM for the Laplace equation. We would like to improve it in order to make easier some extensions in the future as:
  - New boundary conditions for the Laplace equations that may arise from industrial requirements.
  - Possibility to use the code to solve more complex problems (Stokes system, Helmholtz equation).

  Such an extendibility would be a keystep to reach a bigger user community.

- *Modern numerical methods*: We want to develop a code which makes good use of the most advanced numerical schemes and techniques. This choice implies complexity in our algorithms but ensure efficient computations (minimisation of unknowns and good memory access).

- *Parallelism*: In BEM problems we need to assemble matrices where every line is independent from the others. For this kind of massively parallel algorithm we can achieve a very good level of parallelisation. We need to modify the code in order to fully support a parallel computation

- *Building on others' work*: Develop from scratch a code with those characteristics would be almost impossible. Therefore the code has been built on existing libraries. For what concerns the implementation of advanced numerical algorithms we have chosen the `deal.II` library, [2]. In the `deal.II` library the most advanced numerical scheme are implemented and constantly developed. For what concerns the parallelisation we use, again through `deal.II` library, Trilinos, [11], for parallel linear algebra and METIS, [6], for parallel mesh handling.

- *User community*: We believe that a code as ours can only be succesful as a community project. At the time being the code is developed only by the three authors. Our goal is to develop a well written and documented code in order to reach a bigger user community.

3.2 Code Description

We have chosen to divide the code in main objects that handle its different aspects. The four main classes are:

Driver: this class controls the execution of the overall algorithm;

ComputationalDomain: this class handles only the geometry. In particular we don't want it to handle things regarding the unknown and Finite Element Spaces;

BEMProblem: this class is the core of the algorithm. In particular it handles the Finite Element Spaces, it assembles the matrices of the problem and solve the linear system;

BoundaryCondition: this class is responsible for the boundary condition handling. In particular she relies on BEMProblem and ComputationalDomain to know the necessary informations about unknonws and geometry;

We need all these classes to perform a complete simulation. The real application runs typically as follow:

- Mesh creation and refinement

- Functional Spaces definitions
- Boundary Condition Set up
- Matrix Assembling
- System solving

3.3 Code parallelisation

The main goal of the present section is to obtain an efficient parallelisation exploiting the Open MPI paradigm, letting the `deal.II` library handle internally, when possible, the shared memory parallelisation. In the following we report the details of the implementation of the standard BEM. The code provides some very particular boundary conditions that are defined by industrial requirement in ship wave interaction problems.

3.3.1 METIS Grid Partitioning

Many of the parallel applications that use the `deal.II` library make use of a domain which is distributed among all the processors. This is a key ingredient for Finite Element Methods where the interactions are limited to the support of the basis function of the FiniteElement space. In Boundary Element Methods we have a particular situation where the system matrices depend on the position of all the unknowns. This is the main difference between FEMs that uses big sparse matrices and BEMs which exploits little but full matrices. Therefore the computational domain needs to be shared among all processors. We stress that since that kind of domain is only the boundary of a standard FEM grid, we can afford all the processors to have a copy of it. Then we can split only the unknowns, and the matrices, among the processor exploiting the massively parallel environment of BEMs.

We have chosen to use METIS, [6], as a graph partitioning tool to split the domain into subdomains among processors. METIS takes also care of the load balancing among the processors but it needs the triangulation to be shared which is exactly what we need too. METIS has been developed to minimise the number of overlaps between different processors. This ensures that the communication is minimised.

The METIS partitioning is stored in the Triangulation object as a *subdomain id* that memorise which part of the domain is active on each processor. METIS is a geometrical tool that only splits the cell, thus if we are using standard Lagrangian Finite Elements we have some unknowns shared between processors. Given the *subdomain id* we can retrieve two different subsets of indices representing our unknowns:

- IndexSet with shared elements: It reports all the unknowns connected to a given *subdomain id*, therefore it comprehends the overlapping elements.
- IndexSet without shared elements: It reports only the elements actually active on each processor, in particular we have that the sum of the unknowns of these IndexSets equals the total number of unknowns.

In particular we need both these subsets in different part of the parallelisation strategy. This is need to retrieve an efficient way to split and export distributed vectors accordingly with these two sets.

```
// Grid reading from input file
GridIn<dim-1, dim> gi;
gi.attach_triangulation (tria);
gi.read_ucd (in);
// Grid refining as requested
tria.refine_global(refinement_level);
// METIS partitioning
GridTools::partition_triangulation(n_mpi_processes, tria);
```

Code Listing 3.1: Usage of METIS grid partitioning.

In Figure 3.1 we can see a grid partitioned between 7 processors.

3.3.2 IndexSet Creation

We are using a codimension one environment and we need to use both scalar and vectorial unknown. To make the vectors as contiguos as possible we have chosen to renumber our degrees of freedom subdomain wise. In this way we ensure that the scalar unknowns are contiguos processor by processor. In order to implement the boundary conditions we need a combination of scalar and vectorial unknowns. We need great care in the creation of these index sets and we must assure consistency among them in order to combine properly gradients of scalar unknowns with vector unknowns. In particular we have noted some problems in creation of the unknown-subdomain associations when we have shared unknowns between different subdomains. Therefore we force the creation of a consistent vectorial index set given the scalar one. Furthermore we have chosen to renumber our vectorial unknowns in order to guarantee a stronger consistency with the scalar representation.

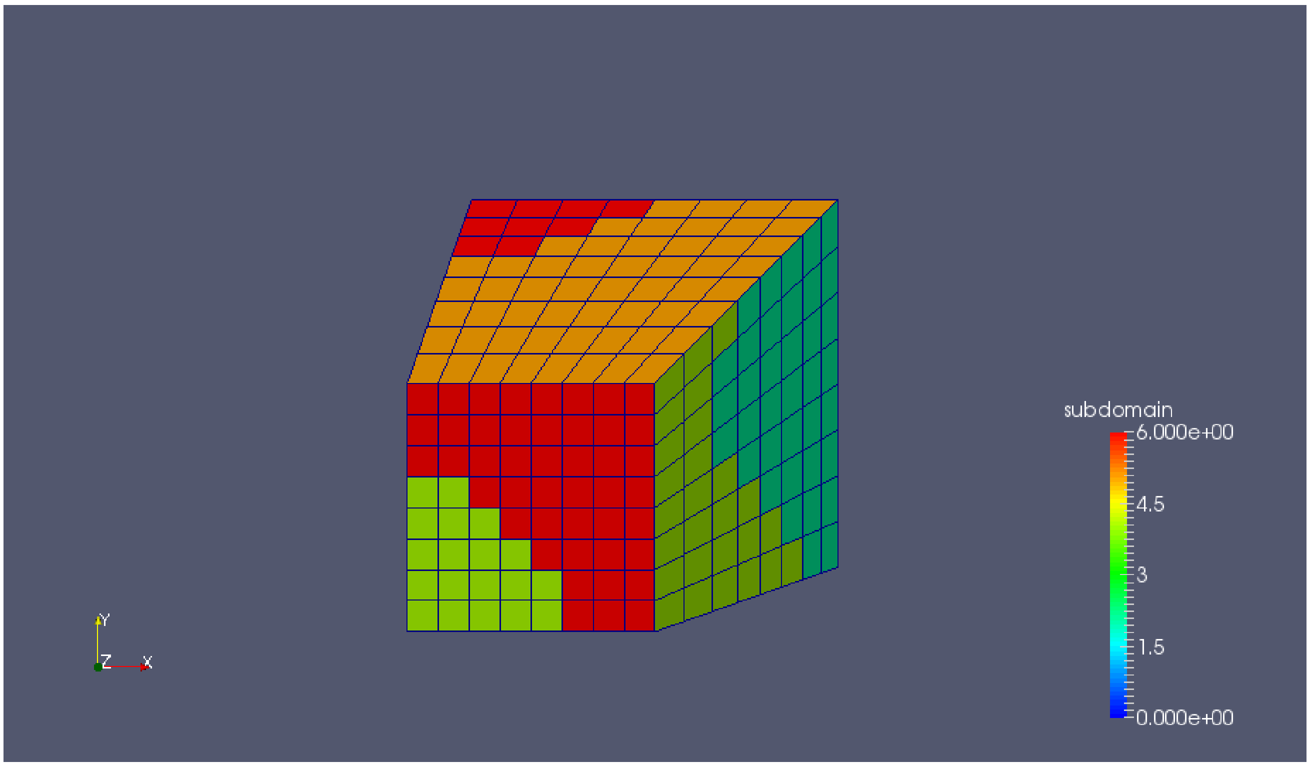

Figure 3.1: METIS non trivial partitioning

```
for (types::global_dof_index i=0; i<n_dofs; ++i)
  if (dofs_domain_association[i] == this_mpi_process)
    {
      this_cpu_set.add_index(i);
      types::global_dof_index dummy=sub_wise_to_original[i];
      for (unsigned int idim=0; idim<dim; ++idim)
        {
          vector_this_cpu_set.add_index(...);
        }
    }
```

3.3.3 Trilinos Parallelisation

We use the wrappers of the Trilinos library already provided by `deal.II` . The Trilinos library provides, see [12, 11], a high level interface that we can use to handle almost any kind of parallelisation problem. In particular we can exploit the subsets obtained with METIS to set up the distribution of our vectors among processors. We stress that since the decomposition provided by METIS is not triv-

ial it would be almost impossible to manually treat the communication pattern. We highlight that once we have set up the vector partitioning we can access to its value simply using the global indices of such a vector. This make the code extremely similar to the serial one. We must only take care that the specific index is stored on the processor but we can do it checking the METIS partitioning through the usage of the index sets. Trilinos allows for a very straightforward interface with the linear system solver already available in `deal.II` .
Almost all the algorithm of `deal.II` have been wrapped to use also Trilinos vectors. In particular this is a major advantage in the resolution of the linear system once we have assembled the distributed matrices. We have chosen to use a Generalised Minimal RESiduum linear solver, see [16]. In particular GMRES is a Krylov method that minimise the residual of a linear system. It has been designed to work with non symmetric matrices. Therefore we can apply it to the full non symmetric matrices of BEM. We can use it directly with Trilinos distributed sparse matrices and vectors.

3.3.4 BEM Algorithm

In this section we describe the actual implementation of the BEM algorithm. First of all we provide a profiling of the code and then analyse the implementation of what we believe to be the bottlenecks of the algorithm.

Serial Code Profiling

We consider a test case scenario:

- Laplace Boundary Element Method with 5 global refinements.
- Double Nodes on every sharp edge, [10], this is a request of industrial interest. We need to maintain these characteristics.
- Mixed Neumann Dirichlet boundary condition.
- Mixed vectorial scalar constraints.

We use the Trilinos utility TeuchosTimeMonitor to take note of the number of time a specific method is called and the time taken for each call. We consider a simulation with 6534 unknowns to identify the bottlenecks of the code. We can see in Table 3.1 that the assembly of the two full matrices needed for the BEM resolution is the major part of the overall program. We should focus our efforts on its parallelisation. We stress that our first goal is to enhance parallelization through Trilinos. In order to do so we also need to take care of the Gradient recovery method. This is essential to make the overall code run fully in parallel.

Assemble Time	68.44
Solve Time	5.741
Normal and Gradient Recovery Time	0.1645
Total Time	78.6

Table 3.1: First profiling of the execution time in a serial environment

Matrix Assembling

Since this function is extremely time consuming we need to parallelise it very efficiently. As explained in Chapter 2, we use a collocation scheme to solve the boundary integral formulation represented by equation (2.7). Every line of the matrix can be assembled at the same time, making this an embarrassingly parallel BEM formulation. A good way to parallelise the assembling is applying the MPI paradigm and we expect a theoretically linear scalability. The assembling can be sketched as follows:

- Loop over all the degrees of freedom.
- For any unknown we need a loop over all the cell in the domain in order to assemble a single line of the system (2.14).

We need to split the first loop among all the MPI processors. We use the index set for the scalar unknowns. In this way a processor performs the loop over the cells only if the degree of freedom is between the ones it owns. Thus every processor assemble a different slice of the overall matrix of (2.14). Since this parallelisation is very straightforward we expect almost a linear scalability.

```
for(types::global_dof_index i=0; i<n_dofs; ++i)
if(this_cpu_set.is_element(i))
{
...BEM assembling...
}
```

Normal Vector and Gradient Recovery

This part of the code is needed to approximate the normal vector and the solution gradient on the edges of the domain. In particular we implement an L_2 projection that relies on an efficient implementation of Mass Matrices and right hand sides given a known partitioned vector.
As in the previous computations we have used the Trilinos wrappers for both SparsityPatterns and SparseMatrices. We need to provide the appropriate map (EPetraMap in Trilinos language) to all the Trilinos classes. To do so we need a

vector divided considering the non ghosted vectorial IndexSet. Then we can easily reinit the sparse matrix and proceed to the assemblage. This is very similar to what can be done in a scalar code, see [7], once we take care of ensuring that the cell we are considering in our grid is in the subdomain owned by the specific processor.
Problems arise when we try to assemble the right hand side using a given unknown split using the non ghosted index set. In this way we get an error when the cell has ghost elements because `deal.II` tries to read elements that are not owned by the specific processors. This is clearly a forbidden operation. We can solve this issue building a new unknown using the ghosted index set and the existing solution. Trilinos can efficiently map two vector with different maps internally handling all the needed communications.
Once again we stress that a manual communication pattern would be almost impossible to determine, therefore it is essential the usage of a high level library as `deal.II` and Trilinos. Once we have the ghosted elements we can straightforwardly proceed to the normal and gradient recovery.

Linear Solver

We have chosen to use an iterative linear solver. In Section 2.3.4 we have stressed the need for a preconditioner. We need to parallelise its assemblage. Once we have achieved that we can rely on Trilinos for the parallel implementation of the incomplete ILU and on `deal.II` for the resolution via the GMRES algorithm. The parallelisation of such a preconditioner follows the same strategy of the assembling of the BEM matrices.

3.4 Strong Scaling Analysis

3.4.1 Strong Scalability up to 40 processors

We try to execute a scaling analysis up to 2 nodes (40 processors) on the SISSA cluster Ulysses. We perform our first analysis on the same case of Section 3.3.4, thus we consider 6534 degrees of freedom. We report the result in Figure 3.2.

We can see that the code has almost a linear behaviour up to 4 processor and then we report some non optimal behaviour. The only method that maintains a linear, nevertheless suboptimal, behaviour is the function that takes care of the assemblage of the matrix. This is due to the fact that in this function no communication is involved since it is massively parallel for the well known mathematical peculiarities of the system. For instance we can see that the normal and gradient recovery function has clearly a suboptimal behaviour and this can be due

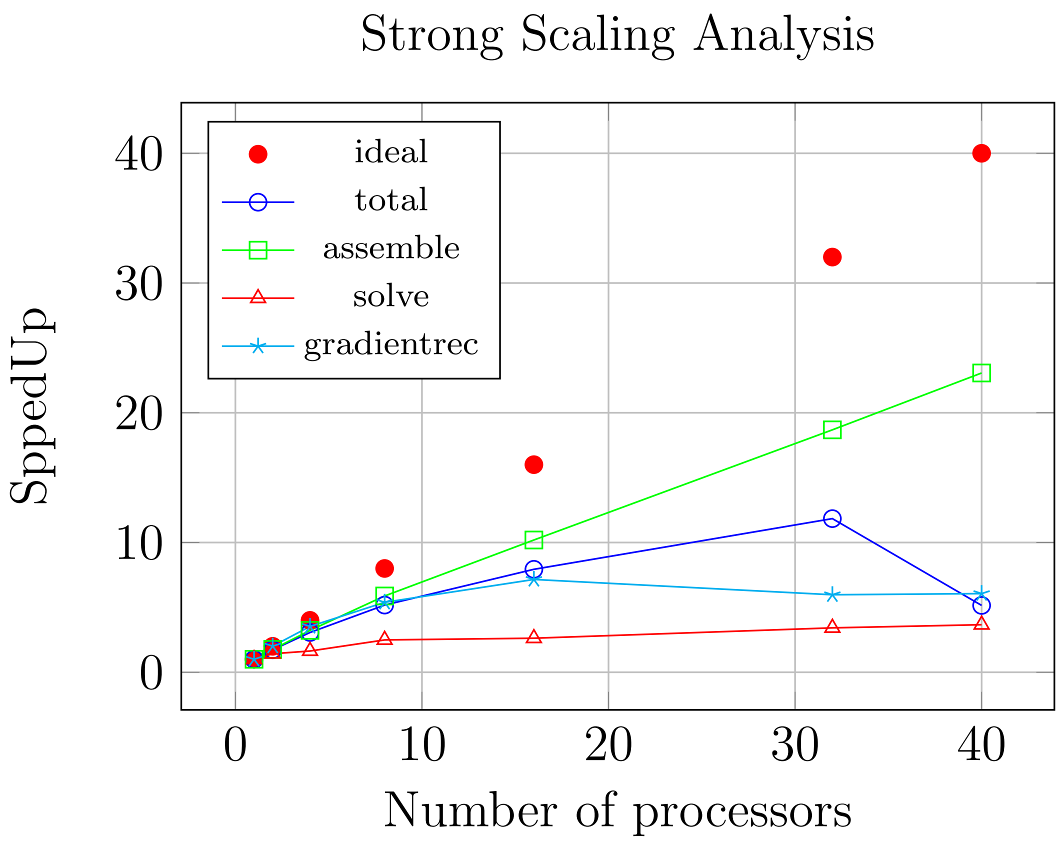

Figure 3.2: Strong Scalability test using 6534 degrees of freedom. We test up to 40 processors. We report the scaling considering the worst timing on all the used processors. In blu with circles we plot the total scalability, in green with squares the performances of the full matrix assembling, in cyan with stars the gradient recovery scalability, and in red with triangles the performance of the linear solver. The red dots represents the ideal speedup.

both communication overheads. These issues could be the reason for the loss of performance between 32 and 40 processors. To test out hypothesis we execute a global mesh refining, thus considering 25350 degrees of freedom, and we repeat the analysis in Figure 3.3

We can easily see that the code has almost an optimal scalability up to 8 processor. This improvement is due to the fact that the communication overhead is not enough to induce a performance loss. We can also see that the less optimal function is the one that takes care of the parallel resolution of the system. This is quite expected since it is mainly based on many matrix vector multiplication that introduce a communication overhead at each step of the computation.

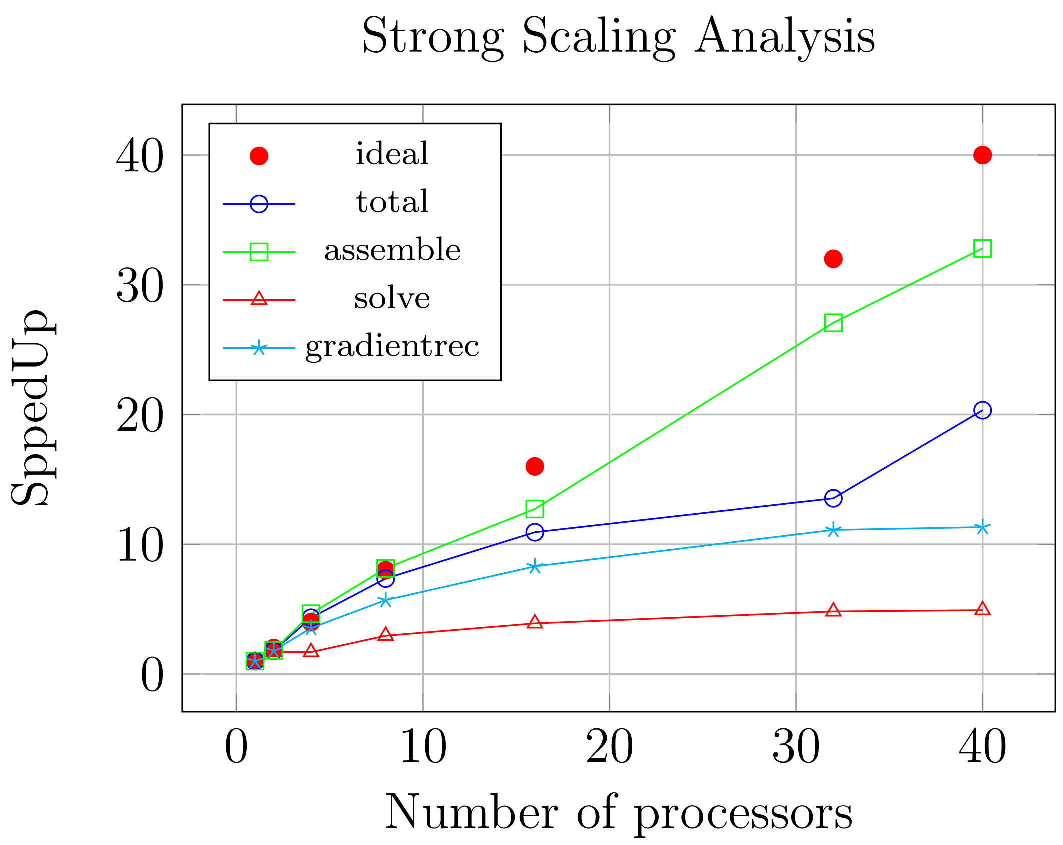

Figure 3.3: Strong Scalability test using 25350 degrees of freedom. We test up to 40 processors. We report the scaling considering the worst timing on all the used processors. In blu with circles we plot the total scalability, in green with squares the performances of the full matrix assembling, in cyan with stars the gradient recovery scalability, and in red with triangles the performance of the linear solver. The red dots represents the ideal speedup.

Now we analyse the gradient and normal recovery function which is the routine we parallelised in order to make the overall code parallel. We can see that it is an optimal behaviour due to the communication needed to the assemblage of the mass matrix. Nevertheless we want to stress that this is not a major issue since this method requires only the 0.06% of the overall computational time. In Figure 3.4 we can see the relative importance of all the main functions of the BEM code.

We can see that the two most importance functions are the one which takes care of the assemblage of the matrices which already has almost an optimal behaviour and the one which solves the linear system. For an increasing number of processors we may expect the assembling function to behave almost optimally

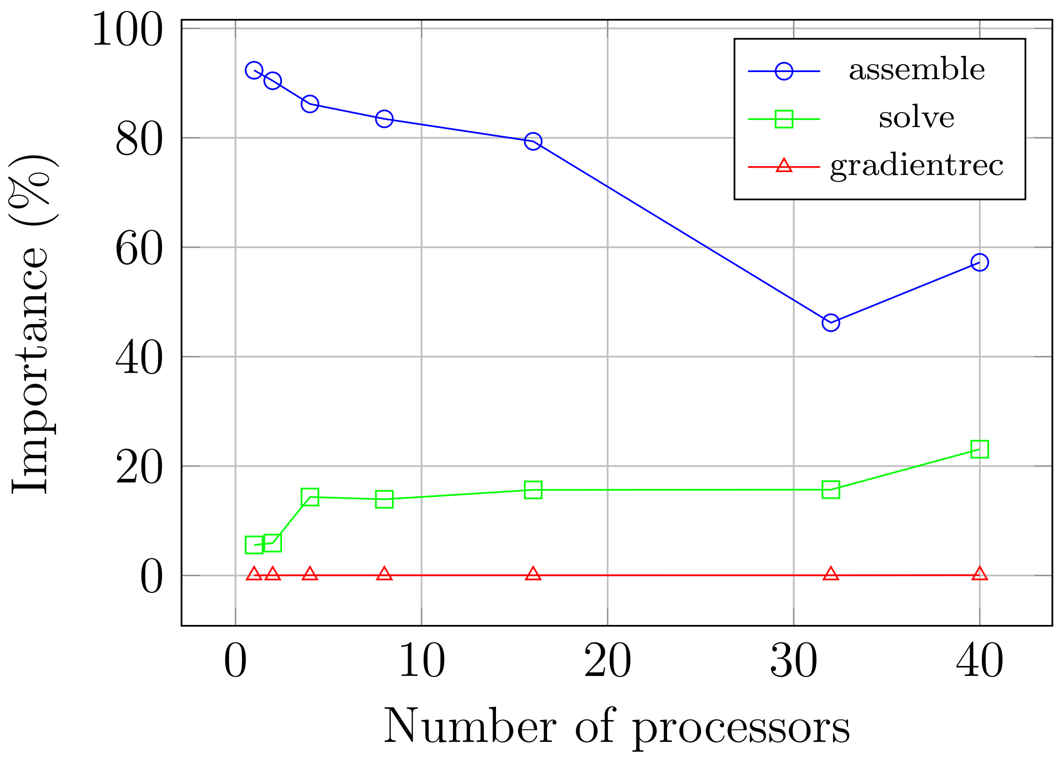

Figure 3.4: The relative importance, in terms of computational time, of the main methods of the BEM code. Simulation up to 25350 unknowns. The blue circled lines represents the assembling of the full matrices, the green squared the importance of the linear solver, and the red with triangles the gradient recovery routine.

while the increasing number of iteration for the linear solver would introduce some new communication overhead.

3.5 Weak Scaling Analysis

In this section we analyse the performances of our code from the point of view of weak scalability.

3.5.1 Computational Cost $O(N^2)$

Since the computational cost of a Boundary Element Method is of order $O(N^2)$, we consider a number of processor that goes as the square of the number of unknowns. In this way we keep constant the computational effort per each processor. We have performed our analysis up to 256 MPI processors and 25350 degrees of freedom. We report our weak analysis in Figure 3.5 We can see that the assem-

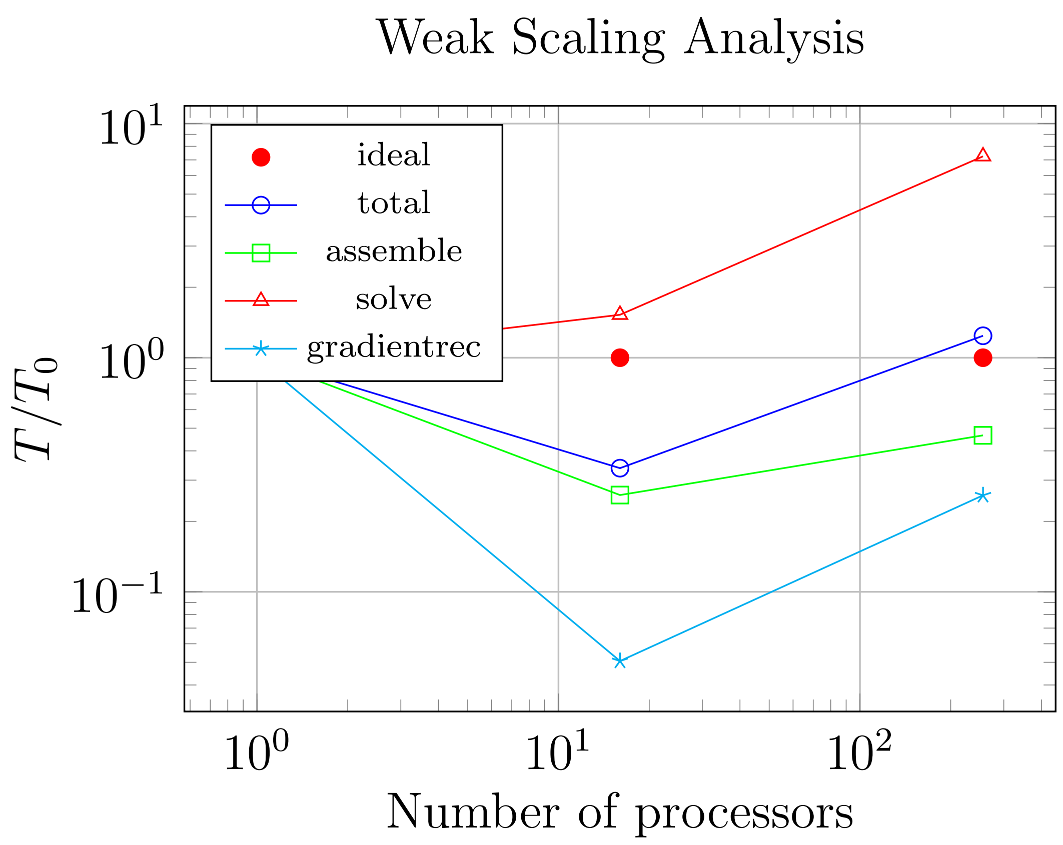

Figure 3.5: Weak Scalability analysis up to 25350 degrees of freedom and 256 processors. We report the scaling considering the worst timing on all the used processors. In blu with circles we plot the total scalability, in green with squares the performances of the full matrix assembling, in cyan with stars the gradient recovery scalability, and in red with triangles the performance of the linear solver. The red dots represents the ideal speedup.

bling routine has a superoptimal behaviour. This is probably due to the peculiar structure of the assembling cycle

```
for(index in processor set)
  for(cell in grid)
    assemble the matrices
```

While we guarantee that the number of matrix elements are the same on each processor, each processor has to loop over all the cells N times less. This should explain the superoptimal behaviour. The matrix assembling is the leading term for the computational cost and induces the superoptimality up to 16 processors. If we increase the number of MPI processors to 256 we increase dramatically the computational costs for the linear solver. This explains the suboptimal behaviour of this part of the code that is becoming very important in terms of computational cost as we can see in Figure 3.4.

3.5.2 Computational Cost $O(N)$

We repeat the Weak Scaling Analysis considering a computational cost of $O(N)$. We don't expect the code to behave optimally since we know that its computational costs scales as $O(N^2)$, however we would like to further analyse the behaviour of the assembling function, which is the most computationally demanding one. We can see clearly that if we consider a low number of degrees of freedom

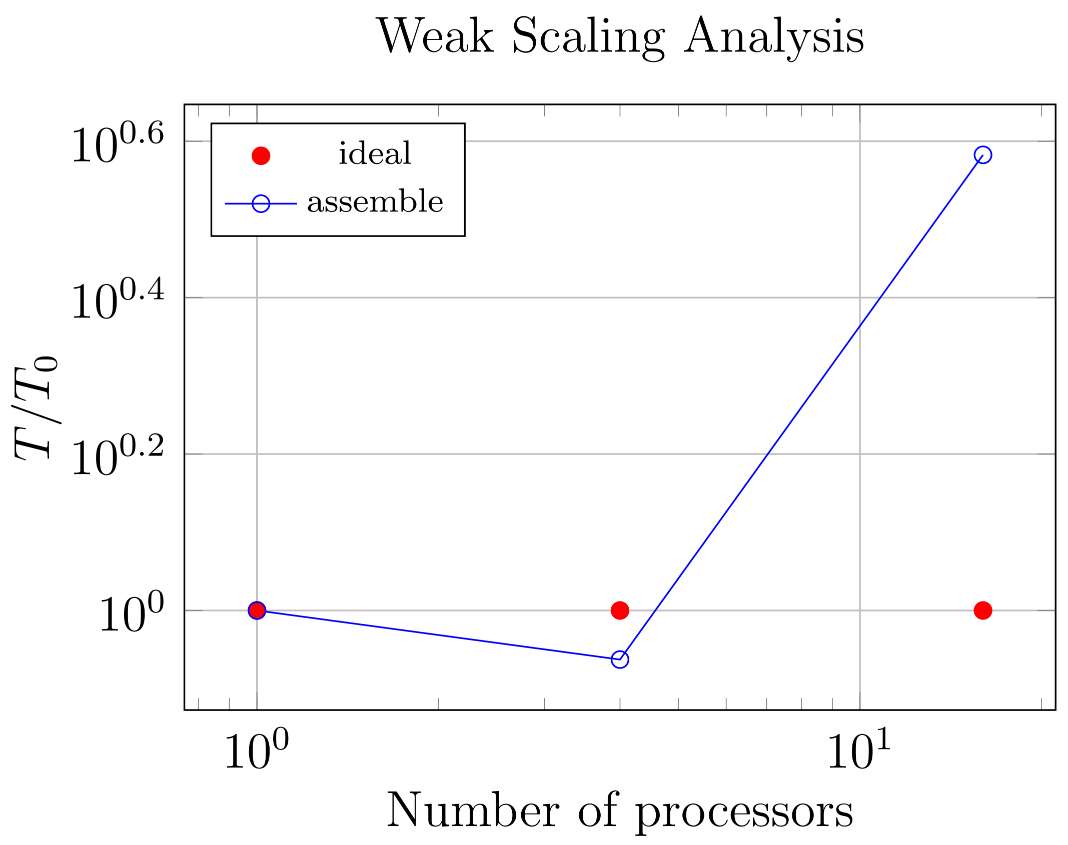

Figure 3.6: Weak Scalability analysis up to 25350 degrees of freedom and 128 processors. We report the scaling considering the worst timing on all the used processors. In blu with circles we plot the scalability for the matrix assembling. The red dots represents the ideal speedup.

we have a computational cost of $O(N)$, while for a higher number of unknowns we recover the expected quadratic behaviour.

3.6 PETSc

In the previous sections we have stressed the use of external parallel library in order to achieve a good scalability. Up to now we have used Trilinos which is already wrapped in the library `deal.II` . We have another option, Portable Extensible Toolkit for Scientific computation (PETSc) [1]. The PETSc library is constantly developed, as Trilinos, to provide high-quality parallel numerical libraries using modern computational strategies in order to reduce the difficulties of writing a parallel code. We need some change inside our code in order to run using PETSc. We define a generic space LA::MPI and then we set it to PETSc ot

Trilinos depending on the case. This is quite straightforward since `deal.II` has already defined almost equal interfaces to both the libraries.

3.6.1 Sparsity Patterns

We need to use a sparsity pattern that is usable for both the libraries. We have chosen the DynamicSparsityPattern from the `deal.II` library. This is due to the fact that we need to specify the IndexSet representing the partion among our MPI processes. We can use the same sets we have used in the precious sections to provide subdivisions for scalar and vectorial unknowns. We stress that our choice may induce a loss of performance for the Trilinos code since we are not using anymore a TrilinosSparsityPattern that we expect to be the best choice for initialising a TrilinosMatrix. However we highlight that the DynamicSparsityPattern is recognizable from the wrappers of both PETSc and Trilinos.

Trilinos Strong Scalability Comparison

We want to compare the two different implementation of the sparsity pattern with the Trilinos library. In Figure 3.7 we see the comparison. We can clearly see that the Trilinos SparsityPattern has better performances both for the reinitialisation time and the matrix assembling. This is probably due to the fact that the native sparsity pattern allows Trilinos for a faster access to the matrix members.

3.6.2 PETSc Trilinos Comparison on a single processor

In this section we want to profile the code on a single MPI processor. We have considered in both cases the implementation using DynamicSparsityPattern in order to highlight the performance differences between the libraries in our code. This is a preliminar analysis. In Section 3.3.4 we have seen that the most expensive section of the BEM is the assembling procedure for the full matrices. We only compare these timings between the two libraries. We have considered a simulation with 25350 degrees of freedom From Table 3.2 we can see the following things:

Function	Trilinos	PETSc
Assemble Time	2494	3187
Total Time	2682	3349

Table 3.2: First comparison between Trilinos and PETSc

we can see how the assembling time represents the major part of the overall com-

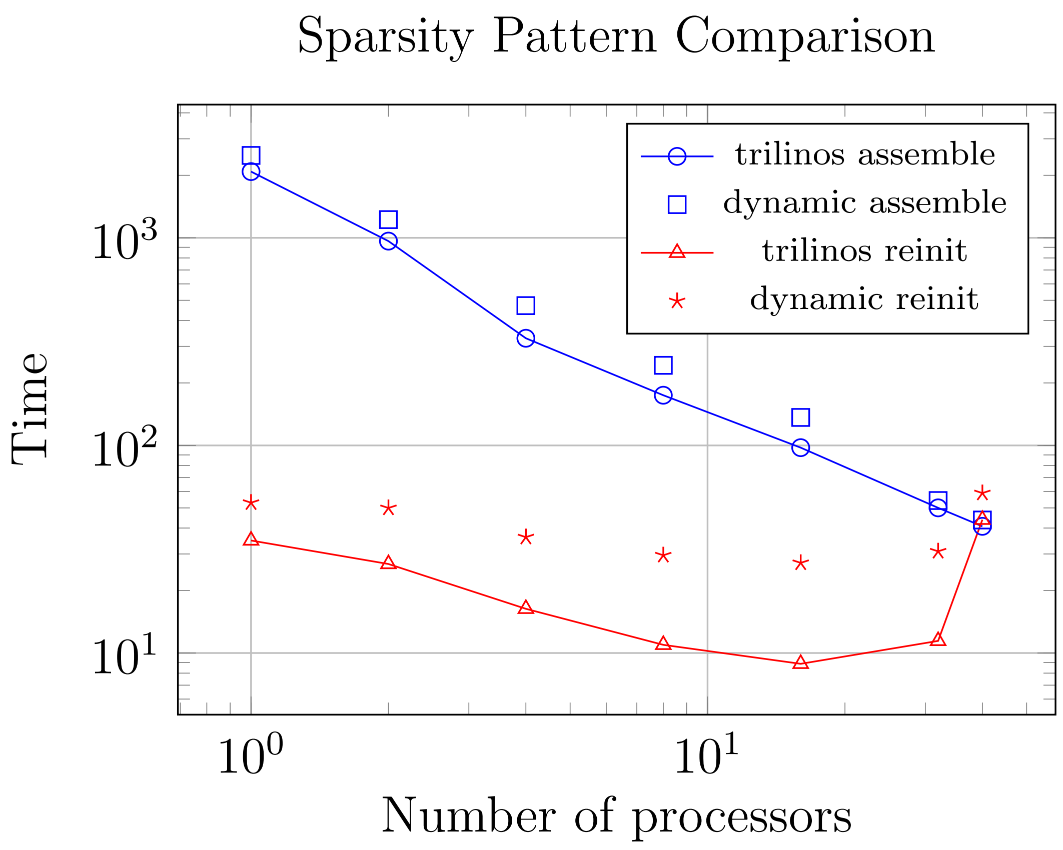

Figure 3.7: The comparison with the two different sparsity pattern implementations. In blue with circles we see the assemble time with Trilinos, the squares represent the time needed by the DynamicSparsityPattern. The red line with triangles represents the reinitialisation time with Trilinos while the stars with DynamicSparsityPatter.

puations, and we can see how the usage of PETSc library has induced a loss of performance of 27.8%. We expect Trilinos to have better perfomance also in a parallel run with more than one MPI process.

3.6.3 PETSc Scalability

In this section we want to see if we are able to gain some scalability advantages using PETSc instead of Trilinos as parallel library. We consider the same problem as in Section 3.4.1. We note that if we use more than one MPI processor our code crashes. We have analysed our code and we have found the problem in the initialisation of PETSc parallel vectors. The `deal.II` manual is very clear about the fact that PETSc supports only contiguos IndexSets. We chose to use a renumbering of the degrees of freedom subdomain wise. In this way we can guarantee

that the IndexSet are contiguos. From a further analysis we have discovered that the problem depends on the vectorial IndexSet.
In section 3.3.2 we have stressed our need to have consistent IndexSets for scalar and vectorial unknowns since we need to combine scalar and vectorial unknowns. This choice has a major drawback since we can no longer guarantee the vectorial IndexSet to be contiguos in memory. In conclusion we can assess that we can't use PETSc in our parallel application. This is due to the very particular constraints we need to impose on the edges of the domain and that have arisen from industrial requirements.

Chapter 4

Parallel FMA with MPI paradigm

In Chapter 3 we have described the implementation of a fully parallel BEM by means of MPI paradigm. In the present chapter we present the coupling of such a parallel Boundary Element Method with a Fast Multipole Algorithm. We believe that such a coupling would allows us to: increase the size of the problem and reduce the overall computational complexity. Firstly we have decided to use a pure MPI parallelisation strategy. For the sake of clarity we have organised the chapter as follows:

- A description of a generic Fast Multipole Algorithm.
- An explanation of the parallelisation strategy we have adopted.
- The analysis of the results we have obtained.

4.1 The Fast Multipole Method

The Fast Multipole Method has been originally presented in 1987 by Greengard and Rokhlin in [9]. It presents a highly efficient implementation of the classical Multipole Method. Its natural implementation is in the field of particle dynamics or, more generally, N-body simulations. We consider a simulation regarding N particles. In order to completely solve the problem we need to evaluate all the pairwise interactions. This leads to an overall computational complexity of $O(N^2)$. We can straightforwardly imply that simulations involving a big number of particles are unbearable. Making good use of the mathematical properties of the kernel involved in the computation we can approximate the interactions between "far" couples. This is the key point of the method in fact we have to consider a certain accuracy in the computation in order to use the multipole expansions. In the framework of Multipole methods it is possible to know *a priori* the error we are introducing approximating the long distance interactions. We can see the general aspects of a Multipole Method in [15]. Following a *divide et impera* approach based on a hierarchical space distribution we can see that the computational costs are $O(NlogN)$. Using the Fast Multiple Method, [9] we can drop the computational effort to $O(N)$ and this results in a great advantage for simulation involving a high number of particles. Our goal is to develop a parallel implementation of an existing Fast Multiple Methos, [14], and couple it with the existing BEM code. The Fast Multiple method isn't embarrassingly parallel, however there are some existing parallel implementation, [13, 3], that show that the FMM is able to deal billions of unknowns on heterogeneous architectures. However we stress a big difference between the classical parallel framework of such codes and our implementation. We will not deal with billions of unknowns but hundred of thousands. This is a crucial different since the communication has a greater impact in the computations. We need to focus on an implementation for such a number of degrees of freedom. Moreover we don't want to introduce any further loss of accuracy. We are implementing and parallelising Greengard original algorithm maintaining all of its characteristics, [9].

4.1.1 FMM in a standard N-Body problem

In this section we want to introduce the mathematical description of a Fast Multipole Method in the standard context of a N-Body problem, we follow the procedure explained in [14]. We suppose to have M of charges $\mathbf{Q}_i$ (called sources) and we want to compute the electrical potential in N points $\mathbf{P}_j$(called nodes). We can straightforwardly compute the unknown potential by means of

$$\Phi(\mathbf{P}_j) = \sum_{i=0}^{M} q_i G(\mathbf{Q}_i, \mathbf{P}_j) = \sum_{i=0}^{M} \frac{q_i}{4\pi r(\mathbf{Q}_i, \mathbf{P}_j)}. \tag{4.1}$$

From equation (4.1) we can see that such an algorithm needs NM computation to get the potential on all the requested nodes. If we assume that M is $O(N)$ we see that this is a $O(N^2)$ algorithm. The so-called Multipole algorithm are based on the expansion by means of harmonic series of the potential field due to the presence of the charge q_i located at $\mathbf{Q}_i$. Such an expansion decouples the contribution of $\mathbf{Q}_i$ and $\mathbf{P}_j$, making therefore possible to group the contribution of different source points into a single *multipole expansion* to be valued just once.

Multipole Expansion

We put ourselves in a polar spherical frame of reference with centre at the origin. We suppose to have k charges $\mathbf{Q}_i = (\rho_i, \alpha_i, \beta_i)$ contained in a sphere with centre at the origin and radius a, so we have $\rho_i < a, \forall i = 1, \dots, k$. If we suppose that the evaluation point $\mathbf{P} = (\tau, \theta, \gamma)$ is located outside the sphere with radius a we can expand the contribution of the charges $\mathbf{Q}_i$ by means of harmonic series. First of all we need to introduce the so-called spherical harmonics defined as

$$Y_n^{-m}(\alpha, \beta) = \sqrt{\frac{(n-|m|)!}{(n+|m|)!}} P_n^{|m|}(cos(\alpha)) e^{-im\beta}, \tag{4.2}$$

where $P_n^{|m|}$ is the associated Legendre function defined as the solution of the following differential equation

$$P_n^{|m|}(x) = \frac{-1^{|m|}}{2^n n!}(1-x^2)^{|m|/2} \frac{d^{|m|+n}(1-x^2)^n}{dx^{|m|+x}} \quad \text{with} \quad n \geq 0, |m| \leq n. \tag{4.3}$$

Using the definition (4.2) we can introduce the expansion by means of harmonic series as

$$\Phi_i(\mathbf{P}) = \frac{q_i}{4\pi r(\mathbf{Q}_i, \mathbf{P})} = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} q_i \rho_i^n Y_n^{-m}(\alpha_i, \beta_i) \frac{Y_n^m(\theta, \gamma)}{\tau}. \tag{4.4}$$

We can see that the contribution of $\mathbf{Q}_i = (\rho, \alpha, \beta)$ and $\mathbf{P} = (\tau, \theta, \gamma)$ has decoupled . Thus we can sum all the contribution of the k sources introducing the multipole expansion of all the set of charges as

$$M_n^m = \sum_{i=1}^{k} q_i \rho_i^n Y_n^{-m}. \tag{4.5}$$

Using (4.5) we get straightforwardly the potential at point $\mathbf{P}$ as

$$\Phi(\mathbf{P}) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} M_n^m \frac{Y_n^m(\theta, \gamma)}{\tau^{n+1}}. \tag{4.6}$$

We stress that (4.6) offers an exact representation but it is unusable since it implies an infinite summation. We can truncate such a sum introducing an error.

$$\Phi(\mathbf{P}) = \sum_{n=0}^{p} \sum_{m=-n}^{n} M_n^m \frac{Y_n^m(\theta, \gamma)}{\tau^{n+1}} + \varepsilon \approx \sum_{n=0}^{p} \sum_{m=-n}^{n} M_n^m \frac{Y_n^m(\theta, \gamma)}{\tau^{n+1}}. \tag{4.7}$$

It essential to be able to value exactly such an approximation error ε. We can do that analytically as

$$\varepsilon = \left| \Phi(\mathbf{P}) - \sum_{n=0}^{p} \sum_{m=-n}^{n} M_n^m \frac{Y_n^m(\theta, \gamma)}{\tau^{n+1}} \right| \leq \frac{\sum_{i=0}^{k} |q_i|}{\tau - a} \left(\frac{a}{\tau}\right)^{p+1}. \tag{4.8}$$

We stress that any multiple method is based on the estimation of such an error in order to set up the entire algorithm. If we don't accept such an error we have no choice than to go for the exact $O(N^2)$ algorithm presented in (4.1). From equation (4.5) we can see that the multipole expansion is a complex number associated with two indices, namely n, m, and it contains information about all the sources located inside the sphere. We can apply (4.6) to any point outside the sphere of radius a to get the potential due to the k charges. We stress again that M_n^m needs to be computed only once and this is major advantage of the multipole method. The introduction of the finite multipole can lead to a reduction of the computational costs, in fact if we assume to have k nodes outside the sphere and the cost of the multiple method would be $O(p^2 k)$ while the exact method costs $O(k^2)$ operations. If we assume $p \ll k$ we see that we have gained an advantage. The linear dependence on k for the multipole method is achieved introducing $O(p^2)$ complex computations and these operation are considerably more difficult that the simple real evaluation of a distance between two points of (4.1). We need to stress that whenever a point is located inside the sphere of radius a we need to perform the exact computation using (4.1).

Hierarchical Space Subdivision

In order to efficiently apply the multipole method we need to hierarchically subdivide the space in order to know for which nodes we can apply (4.7) and where we need (4.1) instead. We choose to use an octree partitioning of the space. This is a tree data structure in which each space element (block) can be recursively divided into 8 octants, thus octree. This kind of structures are often used to partition three dimensional domains exploiting their recursive divisions. We start with a first block that contains all the sources and the nodes and then we recursively subdivide it into children blocks until we have reached a specified number of levels (or a specified number of nodes-sources in the last level). If a children block does not hold any source and node it is purged from the tree. With such a structure we can easily determine a list of nearest neighbours in which we need (4.1), while we can safely use (4.7) in the other blocks, since (4.8) holds. Using such a *divide et impera* strategy implies a computational of the order $O(NlogN)$ as we can see in [5]. The Fast Multipole Method makes good use of a series of mathematical tools in order to get a computational cost of the order of $O(N)$.

Multipole Expansion Translation M2M

Up to now we have seen that we need to evaluate multipole expansion for any many different regions, blocks. We would like to aggregate these expansions in order to quicker compute the long-range interactions. In order to do so we need to be able to translate the multipole expansion, which are centred at the centre of each different block, to the centre of the parent block and then sum them. If we call M_n^m the multipole to be translated with spherical coordinates, with respect to the parent block centre, (ρ, α, β) we have that the translate multipole is given by

$$R_j^s = \sum_{n=0}^{p} \sum_{m=-n}^{n} \frac{M_{j-n}^{s-m} i^{|s|-|m|-|s-m|} A_n^m A_{j-n}^{s-m} \rho^n Y_n^{-m}(\alpha, \beta)}{A_j^s}, \tag{4.9}$$

where

$$A_n^m = \frac{-1^n}{\sqrt{(n-m)!(n+m)!}}. \tag{4.10}$$

We get a new error bound for any evaluation of the new multipole R_j^k outside a sphere of radius 2ρ

$$\left| \Phi(\mathbf{P}) - \sum_{j=0}^{p} \sum_{s=-j}^{j} R_j^s \frac{Y_s^j(\theta, \gamma)}{\tau^{j+1}} \right| \leq \frac{\sum_{i=0}^{k} |q_i|}{\tau - 2\rho} \left(\frac{2\rho}{\tau} \right)^{p+1}. \tag{4.11}$$

We stress that the evaluation of all the approximation errors is essential for a proper implementation of the Fast Multipole Method.

Multipole to Local Expansion Conversion M2L

We would like to be able to evaluate the effect of a far away multipole expansion just once and then values its effect on all the nodes. In order to do so we need to introduce the so-called Local Expansions. A local expansion L_n^m is a mathematical tool that allows us to sum the effect of a multiple on *all* the blocks that are well separated from it. We consider a set of sources located in a sphere of radius a centred in $\mathbf{Q}(\rho, \alpha, \beta)$ with respect to a polar spherical frame of reference centred in the well separated block. We call R_j^s the corresponding multipole expansion and we get

$$L_n^m = \sum_{j=0}^{\infty} \sum_{s=-j}^{j} \frac{R_j^s i^{|n-s|-|n|-|s|} A_j^s A_n^m Y_{j+n}^{s-n}(\alpha, \beta)}{-1^n A_{s-m}^{j+n} \rho^{j+n+1}}. \tag{4.12}$$

We see that we are not introducing any new approximation error but we are considering an infinite summation. We can truncate such a sum and compute the potential at a point $(P) = (\tau, \theta, \gamma)$ contained in a sphere of radius a through the evaluation of the local expansion as

$$\Phi = \sum_{n=0}^{p} \sum_{m=-n}^{n} L_n^m Y_n^m(\theta, \gamma) + \delta \approx \sum_{n=0}^{p} \sum_{m=-n}^{n} L_n^m Y_n^m(\theta, \gamma). \tag{4.13}$$

If we suppose $\rho > 2a$ we can value the new approximation error analytically as

$$\delta = \left| \Phi(\mathbf{P}) - \sum_{n=0}^{p} \sum_{m=-n}^{n} L_n^m Y_n^m(\theta, \gamma) \right| \leq \frac{\sum_{i=0}^{k} |q_i|}{\rho - 2a} \left(\frac{a}{\rho - a} \right)^{p+1}. \tag{4.14}$$

Local Expansion Translation L2L

Local expansions account for the contributions of all the charges that are in well separated blocks. If we are considering the effects of the sources located in block B_1 on the nodes of the well separated block B_2 we know that B_1 will be well separated from all B_2 children. Therefore we can make B_2 children inherit the local expansion of B_2, by means of a translation of local expansions. If we call R_j^s the expansion centred in the centre of one child and L_n^m the parent expansion we have the exact formula

$$R_j^s = \sum_{n=0}^{p} \sum_{m=-n}^{n} \frac{L_n^m i^{|m|-|m-s|-|s|} A_{n-j}^{m-s} A_j^k Y_{n-j}^{m-s}(\alpha,\beta) \rho^{n-j}}{-1^{n+j} A_m^n}. \tag{4.15}$$

Where we have assumed the parent block centre located at (ρ, α, β) with respect to the child centre.

4.1.2 FMM in Boundary Element Method

In this section we want to describe the application of the N-Body FMM algorithm to a Boundary Element Method like the one we are developing. In Section 4.1.1 we have seen how the FMM can be developed and applied in a standard N-Body problem, we now want to apply it in a very different settings, the Boundary Element Method. We recall that the kind of operation needed in this framework is

$$\int_{\Gamma} G(\mathbf{P}, \mathbf{Q}(y)) \phi(\mathbf{Q}(y)) ds_y, \tag{4.16}$$

where G is the fundamental solution of the Laplace equation that in three dimension is

$$G(\mathbf{P}, \mathbf{Q}(y)) = \frac{1}{4\pi r(\mathbf{P}, \mathbf{Q}(y))}. \tag{4.17}$$

We can see that we have the same kind of function that we can approximate by means of harmonic series to form multipole and local expansions. We assume $\mathbf{P} = (\tau, \theta, \gamma)$ and $\mathbf{Q}(y) = (\rho(y), \alpha(y), \beta(y))$ and we can get, dropping the explicit dependence on y

$$G(\mathbf{P}, \mathbf{Q}) = \frac{1}{4\pi} \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \rho^n T_n^{-n}(\alpha, \beta) \frac{Y_n^m(\theta, \gamma)}{\tau^{n+1}}. \tag{4.18}$$

If we introduce the discretisation of the domain in the usual triangulation and the unknown ϕ we get on each cell

$$\frac{1}{4\pi}\sum_{n=0}^{\infty}\sum_{m=-n}^{n}\frac{Y_n^m(\theta,\gamma)}{\tau^{n+1}}\int_{\hat{K}}\rho^n Y_n^- m(\alpha,\theta)\omega_i^\phi(\mathbf{Q}(\rho,\alpha,\beta))J(\rho,\alpha,\beta)ds. \tag{4.19}$$

We stress that ρ, α, β depend on the surface integration variable since $\mathbf{Q}$ is taken on that surface, with $J(\rho,\alpha,\beta)$ we indicate the determinant of the Jacobian relative to the affine transformation that maps the reference cell to each real cell and with ω_i^ϕ the i-th base function for the variable ϕ. We can see that we have connected the BIE formulation to what we have introduced in Section 4.1.1. We introduce the k-th panel contribute to the BEM multipole expansion around a block centre $\mathbf{O}$ as

$$M_n^m(\mathbf{O}) = \int_{\hat{K}}\rho^n Y_n^- m(\alpha,\theta)\omega_i^\phi(\mathbf{Q}(\rho,\alpha,\beta))J^k(\rho,\alpha,\beta)ds. \tag{4.20}$$

In the complete boundary integral equation (2.7) we have another surface integral related with the normal derivative of the fundamental solution. We can derive another harmonic series expansion for such a function obtaining another multipole expansion for the k-th panel

$$N_n^m(\mathbf{O}) = \int_{\hat{K}}\nabla(\rho^n Y_n^- m(\alpha,\theta))\cdot\mathbf{n}\omega_i^{\frac{\partial\phi}{\partial n}}(\mathbf{Q}(\rho,\alpha,\beta))J^k(\rho,\alpha,\beta)ds. \tag{4.21}$$

We can, recalling (4.2), straightforwardly compute the requested gradient as

$$\begin{aligned}\nabla(\rho^n P_n^{|m|}(cos(\alpha))e^{-im\beta}) &= n\rho^{(n-1)}\rho^n Y_n^- m(\alpha,\theta)\nabla\rho\\ &+ \rho^n\frac{d}{d\alpha}(P_n^{|m|}(cos(\alpha))e^{-im\beta}\nabla\alpha\\ &- imP_n^{|m|}(cos(\alpha))e^{-im\beta}\nabla\beta.\end{aligned} \tag{4.22}$$

In our algorithm al these integrals are computed by means of numerical integration techniques, using quadrature formulas. In detail we use standard Gauss quadrature integration formula and Telles' singular integration strategy. Once we have computed the two kinds of multipole we can apply all the passages of section 4.1.1.

4.1.3 Description of the Fast Multipole Method

In this section we want to give quite an informal description of the serial FMM, we follow the explanation provided by Greengard and Gropp in [8]. We must divide the sources (quadrature points), and the nodes (degrees of freedom), in near range and far range interactions. We need to value the effect of the sources on the nodes. The FMM uses a *divide et conquera* strategy combined with multipole and local

(Taylor-like) expansion to value the long range interactions. Once these ones have been approximated we value the short range interaction directly using the exact pairwise formulation. The algorithm takes into account N nodes and M sources

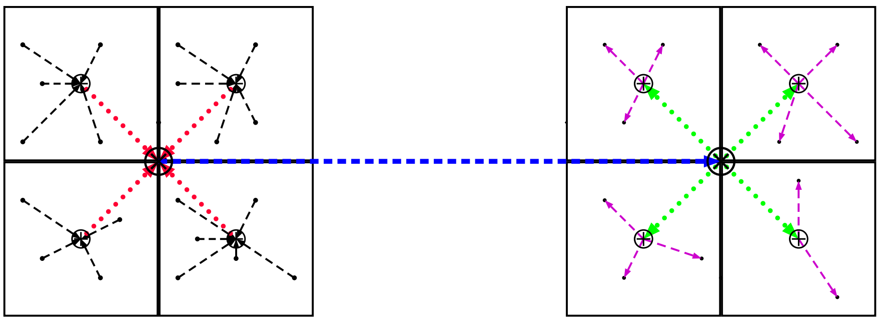

Figure 4.1: The sketch of a complete FMM. On the left we have M sources, on the right N nodes. Black dashed lines represents the multipole expansions. Red dotted are the multipole to multipole translations. The blue line represents the multipole to local conversions. The green lines are the local to local translations and finally in magenta we have depicted the evaluations of the local expansions.

that are distinct in space. First of all we need to introduce a hierarchical space subdivision. We assume that at the first level the space is enclosed in a cube which represents the seed of our octree. We recursively subdivide the tree dividing each box into 8 children. We stop the subdivision when we have a certain number of nodes B in the last level of the tree, the box becomes a leaf of the tree (childless block). If after a subdivision a block is empty, it does not contain any source or node, we purge it from the tree. In general we can assume the level of refinements l to be on the order of $log_8(N)$. For every block at every level of the tree we define its nearest neighbours as the box itself and any box at the same level with which it shares at least one boundary point. In three dimensional simulation we can have 27 nearest neighbours at most. At a given level we call two box, with side length D, well separated if they are separated by at least D. Then we need to associate both multipole and local expansions to each box of the tree. We call χ_i^l the multipole expansion around the centre of the box i at level l. This expansion describes the far-field contribution of the sources that are inside the box i. We denote ψ_i^l as the local expansion of the block i at level l. This expansion describes the contribution of the far-field sources on the nodes that are inside the box i and its nearest neighbours. Then we introduce as $\tilde{\psi}_i^l$ the local expansion describing the contribute of the sources located outside the parent of the box i at level l and all of its nearest neighbours. Finally we need to introduce an interaction list that is associated at any box at any level of the tree. This set of boxes depicts the blocks that are children of the nearest neighbours of i's parent but that are well separated from the box i at level l. We can now draw a scheme of the overall FMM

- Octree Generation. We set the level of refinement (the number of particles in the finest level), and a precision ε for the multipole approximation.
- Direct Interactions. We compute the direct interaction between particles in nearest neighbours box and in all the blocks that does not satisfy the hypothesis for the multipole and local approximations.
- Ascending Phase.
  - Multipole Expansions Generation. We form the multipole expansions χ around the centre of any box at the finest tree level, see (4.20) and (4.21).
  - Multipole to Multipole operation. We recursively form the multipole expansion for any box in coarser levels using (4.9), we stress that each expansion represent the field due to the particles inside the box and that can be applied for long-range interactions.
- Descending Phase. We start from the second level and proceed to finer levels.
  - Multipole to Local conversion.: we convert the multipole expansion χ_j^l into a local expansion ψ_i^l using equation (4.13) for any box j that is located in the interaction list of the box i. The local expansion is centred at the centre of the box i. We add all the different local expansions together.
  - Local to Local operation. If we are at the finest level the process is over, otherwise we can use equation (4.15) and form the expansion ψ_i^{l+1} for any box i children of the box for which we have complete the M2L operations.
  - Local Expansions valuation. We evaluate the local expansions to get the contribution of all the far-field source on the node of the box i
- Sum of multipole and direct contribution to get the overall pairwise interactions

4.2 Existing Code

We have chosen to use an existing Fast Multiple Method currently developed at SISSA, [14], by Dott. Andrea Mola. The code is very accurate from a mathematical point of view, since it uses Greengard original algorithm, but it is completely serial. Our first goal is to achieve a parallelisation of such a code in terms of the MPI paradigm. The existing code is already written in C++ using the `deal.II` library.

4.2.1 Existing Code Description

In this section we want to give a brief description of the existing implementation of the Fast Multiple Method. All the code has been written in C++ making use of the `deal.II` library.

Octree Block

This represents the building block of the octree data structure that is in the code. Basically it is a cube that contains degrees of freedom (nodes) and quadrature points (sources). In the implementation this is done through an *ad hoc* class. The main characteristic of such a class are

- Geometrical attributes: a position vector that represents locates the block in the space, a quantity that specifies the dimension of such a block to determine what elements are inside the block

- Logical attributes: list of children IDs, parent ID, list of nearest neighbours and lists of interacting blocks

The overall octree is represented through a standard STL vector of pointer to blocks.

Proximity Lists

Every blocks needs two different lists that describe how to deal properly with multipole and local expansions.

- Nearest Neighbours List: it contains the IDs of the blocks that share at least one point with the current block.

- Interaction List: it groups the nearest neighbours of the block and its parent

- Non Interaction List: it contains the blocks that were in the interaction list of the parent of the considered block but that are not in the interaction list of the current block. This means that the blocks in this list are well separated from the current block.

Multipole Expansions

In section 4.1.1 we have stressed that the multipole expansion is composed by complex number and that they can be evaluated, translated and added. The complex number are represented though the STL library. The two main methods of such a class are the followings.

- Add: this method can be use, thanking to polymorphism, both to add a new charge to the expansion and to add an expansion coming from a child block

- Evaluate: this function is used to value the multipole expansion, we stress that this approximation is reliable only outside a sphere of radius δ that is stored in the block class.

In order to build the class we must establish the order of the multiple expansion, see (4.7).

Local Expansions

Another C++ class that implements all that is need for the usage of the Local Expansions. Once again we must provide at construction time the order of the approximation. The class stores all the members of the block expansion and has two polymorphic method.

- Add: this method can be use, thanking to polymorphism, both to add a new multipole conversion to the expansion and to add an expansion coming from a parent block

- Evaluate: this function is used to value the local expansion, this is the last step of the descending phase..

We maintain the same order set for the Multipole expansions even for the Local ones.

4.2.2 Serial Code Profiling

First of all we want to profile the existing code to understand which parts are the bottlenecks of the computations and understand how to parallelise them. In this first computations we have considered a single node in the final level of the octree, *i.e.* $B = 1$. We have executed a computations with 6144 cells and 6534 degrees of freedom. We report the overall time and the timings of the three main part of the algorithms: tree generation, ascending phase and descending phase.

Method	Time	Repetitions
Tree Generation	1.798	1
Ascending Phase Time	0.53285714	42
Descending Phase time	22.91380952	42
Total Time	940	1.

Table 4.1: First profiling of the execution time in a serial environment

We stress that the tree generation is needed only once while the other two phases are needed any time a BEM matrix vector multiplication is invoked, thus they are needed at any iteration of our solver. However we can see that the major time needed of the algorithm is due to the descending phase and by the conversion of Multipole Expansions into Local Expansions. We need to focus our attention on the descending phase and we can let the tree generation and ascending one remain serial on all the processors. Once again we stress that every degrees of freedom depends on all the others unknowns so it is essential to have a localised complete tree, which is the FMM representation of the geometry, on every processor.

4.3 Code Parallelisation Strategy

We have decided to parallelise the code maintaining the existing tree generation procedure, we will focus just on the parallelisation of the descending phase. For consistency with Chapter 3 we have taken the Trilinos wrappers of the `deal.II` library as main tool to achieve an efficient parallelisation. We will describe the steps needed by the Trilinos structures to be run in a parallel MPI environment through their `deal.II` interfaces. The tree generation is completely uncorrelated from the geometrical splitting done by the BEM code through METIS. Therefore we can't assume the block to be splittable with the same philosophy. Two main list are generated, one groups the quadrature points in each block, the quadrature points are the charges of a classical particle dynamics FMM, and one groups the degrees of freedom that are the evaluation points of such charges.

4.3.1 Close range interactions

Since for close range interaction the bounds that allow the usage of multipole and local expansions don't hold, we need to use the so called Direct Method, *i.e.* we discretise directly the Boundary Integral Equation to compute these effects. The code, in its method "direct_integrals" works as follows.

- For any childless block we check if its nodes belong to the current processor, using the usual METIS partitioning. Then if the node is on the processor we compute, the ids of the sources in its nearest neighbours block list. Once we have all the ids we can compute the sparse matrix that takes care of these short range interactions.

- Unfortunately we need to compute another class of interactions: if a block is in the non interaction list of another smaller block, the bound for the multipole expansion application does not hold, and so we must compute direct integrals. Here we scan the non interaction lists of each block at each level to look for bigger blocks and we compute these interactions checking that every nodes belong to the current processor.

This procedure is quite hard to implement but it can be parallelised quite straightforwardly with MPI. We only need to pass the IndexSet of the METIS partitioning to this function to let it know if a node belongs to the current processor.

4.3.2 Ascending Phase

In this first BEM-FMA implementation we focus our attention on reducing the time needed without introducing a communication overhead. Moreover we have seen with our first code profiling that the ascending phase, composed by the Mulitpole Expansion Generation in the childless blocks and their translation to their parent blocks in tree isn't particularly time depending. We have decided to proceed as follows

- We execute a AlltoAllv communication to get the complete settings of informations on every processor. This is an expansive communication of two vectors. We stress that we don't deal with very long arrays and that at this point we need to communicate standard doubles so we don't need any particular communication strategy.

- Every processors performs the complete Multipole Generation.

We don't expect any improvement in the performances of the method since we are only introducing a communication overhead. However we know that this function isn't the real bottleneck. In principle we could introduce a parallelisation of the method based on the very same METIS subdivision of our unknowns. We stress that we need a strategy to split quadrature points as well and this would result in a particular communication strategy between processors to get the complete knowledge of nodes and sources. Moreover we would need to perform a AlltoAllv communication for any level of the octree and this would introduce a much greater communication overhead.

4.3.3 Descending Phase

Basically the descending phase cycle on all the blocks that contain a degree of freedom of our BEM system, the DofsFilledBlocks, and: performs the Multipole to Local conversion, the Local expansion translation to the childless blocks and then the Local evaluations. We can insert a check in this cycle so that each processor performs the computations if and only if a owned degree of freedom is present. This is enough to break down the overall cycle, and we can ensure that the every processor compute correctly its owned degrees of freedom since it has the complete Multipole Expansion at its disposal. However this strategy has two major drawbacks.

- The code bottleneck is due by the number of Multipole to Local (M2L) operations needed, with this strategy we are implicitly increasing their total number because if a block has degrees of freedom owned by different processors this means that its M2L operation are performed more times.
- We have no guarantee that the number of M2L operations should be optimally balanced among the processors.

We stress that such a strategy comes with some advantages

- It requires a minimum amount of communication since we can directly assemble the solution vector since we are evaluating the complete local expansion on each owned block.
- Since the degrees of freedom have been, almost optimally, split according to METIS algorithms we can expect a sensible reduction of the overall M2L operations needed on each processor.

We don't expect a perfectly linear speedup in our algorithm but we expect a sensible performance increase in the BEM matrix vector multiplication. We stress however that if we try to optimally balance the M2L workload we would need an expending communication (basically a AllToAllv) of the local expansion, that are not standard MPI types, at each level of the level. This would introduce a much greater overhead with respect to the M2L increase due by the multiply owned processors.

4.3.4 Preconditioner Setting

We have chosen to use a ILU preconditioner build upon the sparse matrix we have assembled for the direct contributions. At this level we need to consider the actual constraints otherwise the preconditioned would be ineffective. The structure of this method is very similar to what we have highlighted in 4.3.1 so we need only the IndexSet representing the nodes on the processor to split the work among all the processors. The MPI parallelisation is straightforward and it is absolutely needed since the computational cost of the method is of the same order of magnitude of the computation of the short range interactions.

4.4 Strong Scaling Analysis

First of all we want to analyse the performance of a standard BEM matrix vector multiplication in the framework of FMM. We expect a good speed up since we are reducing the number of M2L operations needed by each processor.

4.4.1 Preliminary Multiple to Local Operations Work Balance

In order to better understand how our algorithm is working we analyse the number of Multiple to Local transfer operations needed on each processor. This is basically the sum of the operation needed by each block locally owned by each processor. We stress that we count any contribution on any processor so we are considering the increase number of operation introduced by our algorithm. We consider the same case of Section 4.2.2, so 6534 degrees of freedom. Since this is a preliminary analysis we run up to 3 processors. This is done in order to understand the effects of our algorithm on a small simulation. We can get the following table.

Processors	Proc0	Proc1	Proc2	Total	Increase	%
1	779135	0	0	779135	0	0%
2	376286	430708	0	806994.0	27859	3.57%
3	264962	273188	273188	811338	32203	4.13%

Table 4.2: First profiling of the execution time in a serial environment

We can see that our work balance is suboptimal, especially for two processors we can see that processor one will be slower than processor zero for the increased number of operations assigned. This will result in an overhead for the overall computation time. Anyway we can easily see that even if we get a non negligible increase of the operations this should be well balanced by the reduce number of M2L operations on each single processor. For three processors we can see a better work balance. This suggests us that the work balance is strongly dependent on the number of processors and on the original grid of the problem. This is quite expected since we are relying on the domain subdivision provided by METIS to split the workload in the descending phase of the algorithm. Such a split does not guarantee an optimal workload distribution but it ensures us that the communication are minimised. At this point we are ready to perform the scalability analyses for the BEM-FMA code we have developed.

4.4.2 Strong Scaling up to 40 Processors

In this section we want to see weather our parallelisation strategy is effective in a strong scaling framework. We keep fixed the size of the problem to 25350 degrees of freedom and let the number of processor increase up to 40 processor on Ulysses cluster, thus we are considering two complete nodes of 20 core each. In Figure 4.2 we can see the overall results of our parallelisation.

We can see that our code does not present an ideal linear scalability. In fact we can see that we can hardly reach a speed up of 10 with 40 processors. Thus we can see an overall efficiency of 25%. In order to better understand the reasons that lead us to such a suboptimal performance we analyse the speed up of a single matrix vector multiplication. In Figure 4.3 we analyse the scalability of a single matrix vector operation and for all the operations needed to solve the linear system.

We can see two very important things. First of all the loss of performance is due to the fact that the increasing in the number of processors causes an increasing in the iteration needed to solve the system. This cause a loss of performance of almost the 50% of our code. Another reason of the performance loss is the increased number of Multipole to Local (M2L) operations needed, as stressed in section 4.4.1. We can identify a third very important motivation for the suboptimal behaviour of our code. In Section 4.3 we have stressed that we have kept the ascending phase of the algorithm localised on each processor to minimise the communication overhead. We can't expect any performance increase from this method, moreover we need a AlltoAllv communication at the beginning of this function to localised everything. In Figure 4.4 we can see the relative computational importance in the ascending phase of such a communication. We can see that causes a loss of performance as the communication pattern increase in complexity. The ascending phase is becoming relatively more important, and for Amdahl's law we are loosing performance. In Figure 4.5 we see the comparison between the computational importance of the two phases We can see that for 40 processors the two phases have the same relative importance. Thus the overall performance gain, for Amdahl's law, is limited by the presence of the unoptimised ascending phase.

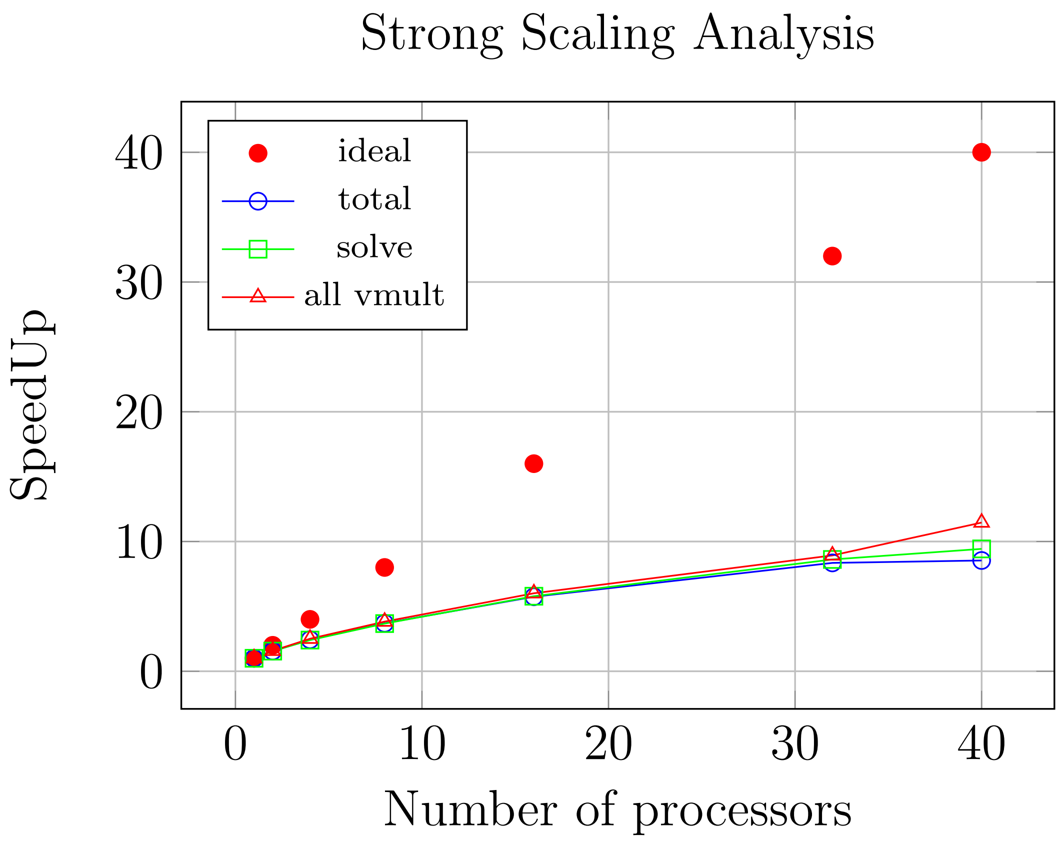

Figure 4.2: Strong Scalability test using 25350 degrees of freedom. We test up to 40 processors. We report the scaling considering the worst timing on all the used processors. The blue line with circles representes the overall scalability for a complete execution, the green squared one the scalability of the linear system solver, the red tringled the scalability for all the needed matrix vector multiplications. The red dots represents the ideal scalability

4.5 Nodes per block

The actual implementation of the code prescribes one node per childless block. This choice should maximise the usage of Fast Multipole Method but it may not be the optimal choice. In [8] we can find a very clear study about the optimal number of nodes per childless block. The author asserts that the scalability and the performances of the code strongly depend on the amount of nodes per block level. We have decided to introduce an additional parameter in our code in order to specify the maximum number of nodes in each childless block. In our opinion this is a key ingredient to achieve better performances. A very important effect of this strategy will be the increase of direct interactions in the algorithm. We stress

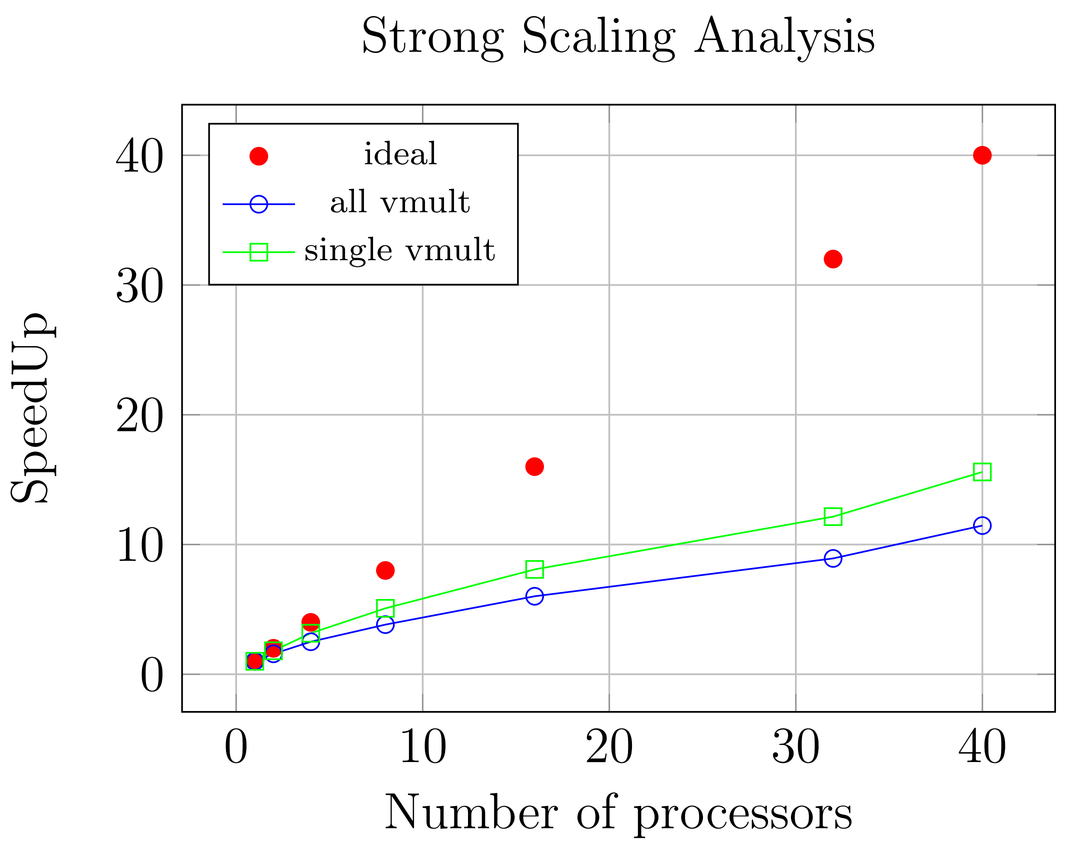

Figure 4.3: Strong Scalability test using 25350 degrees of freedom. We test up to 40 processors. We report the scaling considering the worst timing on all the used processors.In blue with circles we plot the overall scalability for the complete set of matrix vector multiplications, in green with squares the scalability of a single matrix vector multiplication. The red dots represents the ideal scalability

that this should lead to a better cost balance between the short range interactions, needed to deal with nearest neighbours., and the long range ones, needed for the multipole - local parts. We expect the direct part to scale almost linearly since it is implemented very similarly to the direct BEM assembly routine which has been proved to behave optimally.

4.5.1 Optimal Nodes per Leaf

In this section we want to investigate weather we can find an optimal number of nodes per leaf in our FMM algorithm. in [8] there is such optimal condition but we stress that the direct BEM implementation is very faster than our FMM so we may expect a monotone behaviour increasing the number of nodes per leaf. We have

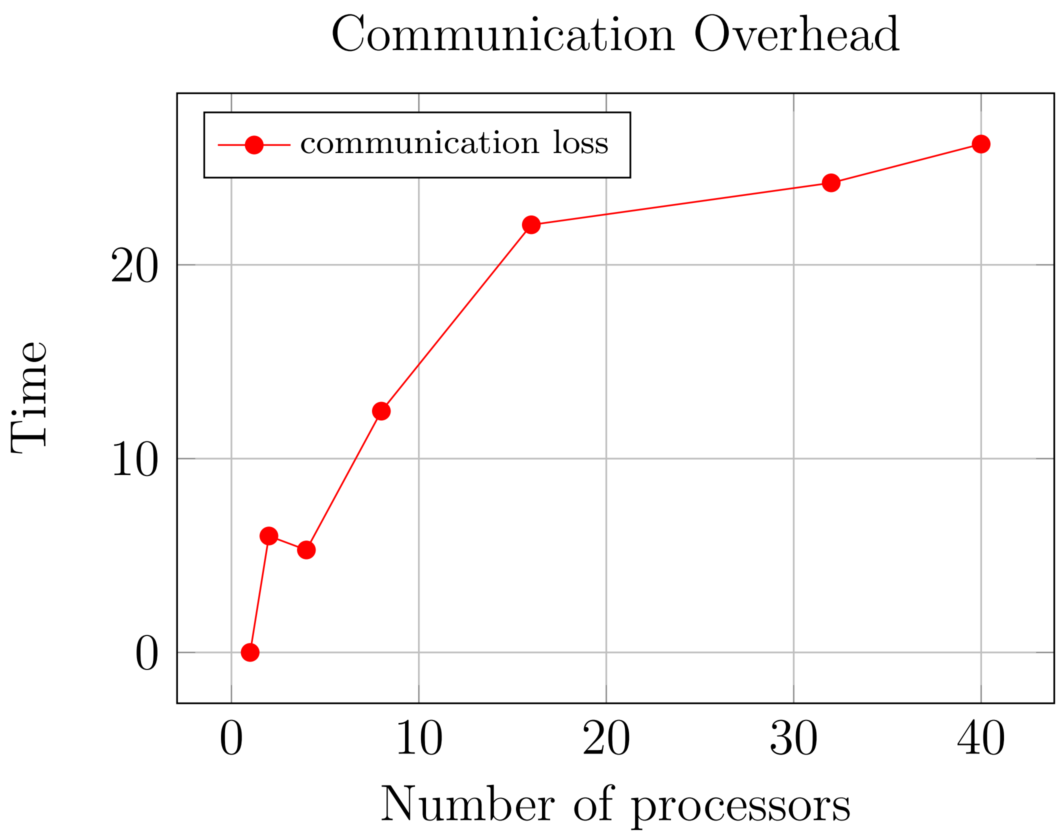

Figure 4.4: Performance loss due to the increase communication in the ascending phase

run our analysis on 32 processors on the Ulysses cluster. We have considered our usual test case with 25350 degrees of freedom. We have let the number of nodes per leaf vary from 20 to 200. We can clearly see from Figure 4.6 that we have an optimal value around 60 nodes per leaf. According to [8] the optimal value depends on the hardware on software specifics. In the paper the author states that his optimal value is around 15 nodes per block. We find quite reasonable an increased value in our case since 25 years have passed with a lot of improvements in terms both of hardware and software. In the following subsection we try to understand why we see an optimum value .

Direct cost

Roughly speaking we can assert that our algorithm can be split in two main parts: the near range method and the long range algorithm. In Figure 4.7 we analyse the cost needed by the direct near range interactions as we increase the

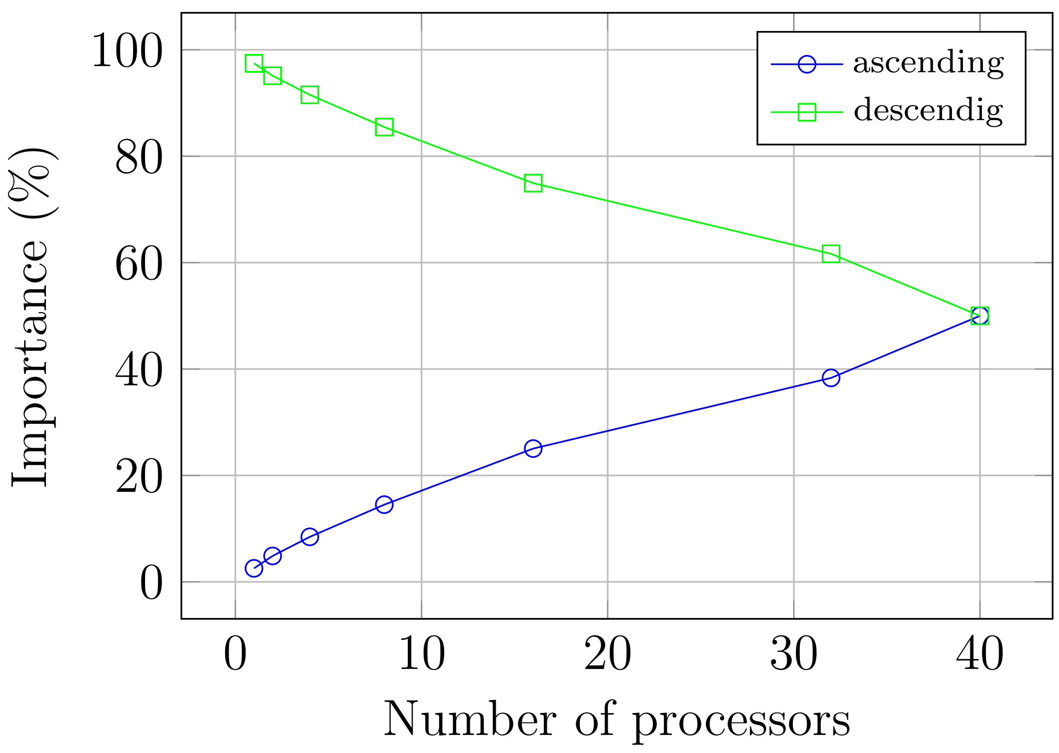

Figure 4.5: Relative importance of the different phases in a matrix vector multiplication using the BEM FMA algorithm. In green with square we see the importance of the descending phase while in blue with circles we see the importance of the ascending phase

number of nodes per leaf. As we can see such a cost is directly proportional to the number of nodes per leaf in our octree. This is due to the fact that we consider the same amount for nearest neighbours per block but each block now contains more sources and nodes. This increases the number of element that we need to directly compute. For instance for 60 nodes we compute directly the 1.89301% of the elements, while with 30 nodes we compute the 1.14803%. This causes a cost increase in our algorithm, that should be compensated by the cost reduction in the FMM part.

FMM Matrix Vector Multiplication Cost

In Figure 4.8 we analyse the computational cost needed by matrix vector operations increasing the number of nodes per leaf in the tree. We can see that such a cost is not monotone. Firstly an increase in the nodes per leaf induces a reduction

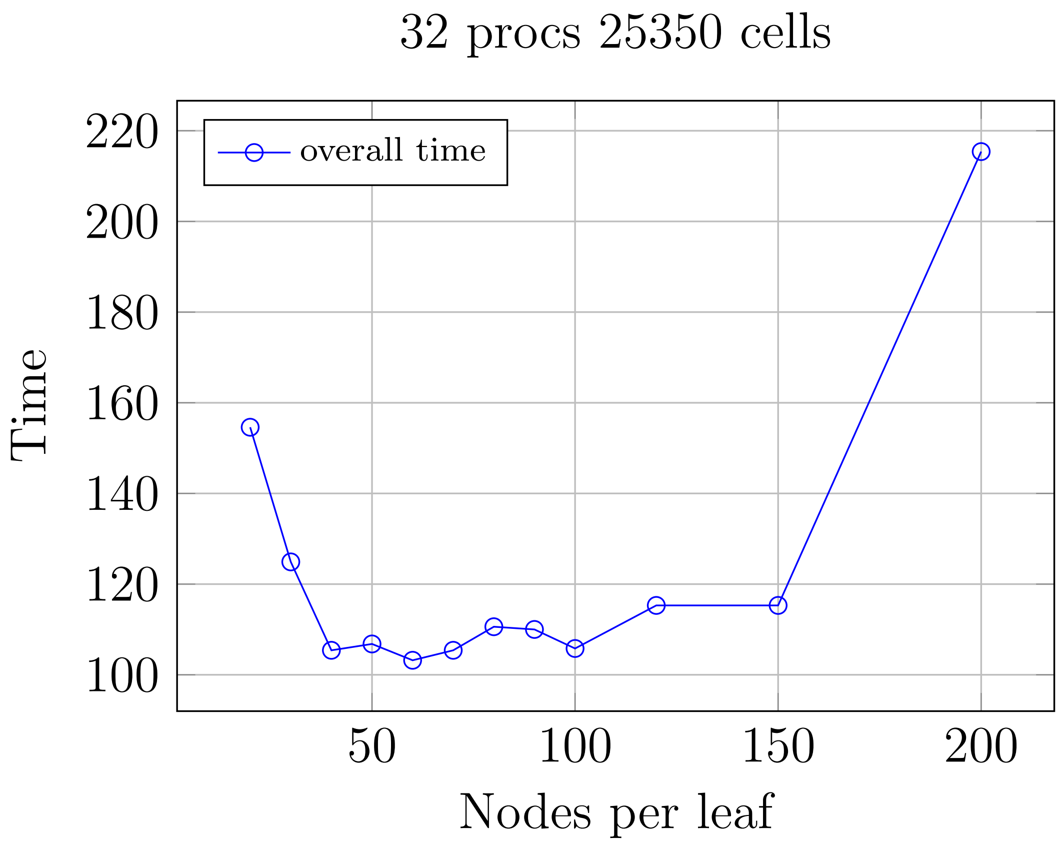

Figure 4.6: Overall time to solution for the BEM-FMA algorithm over 32 processors.

of the number of blocks, thus a reduction of the transfer operation needed. This may explain the first part of the curve where the costs decrease. After 150 nodes per leaf we see a cost increase. This may be due to the unbalanced workload for such a condition. Moreover if we have less blocks we have an increased percentage of blocks shared between processors. This introduces more overheads in our computation.

Optimal Conclusion

From Figures 4.7 and 4.8 we can conclude that the optimal value of 60 nodes per leaf highlighted in Figure 4.6 may be due to the two different effects of the direct cost increase and of the FMM behaviour.

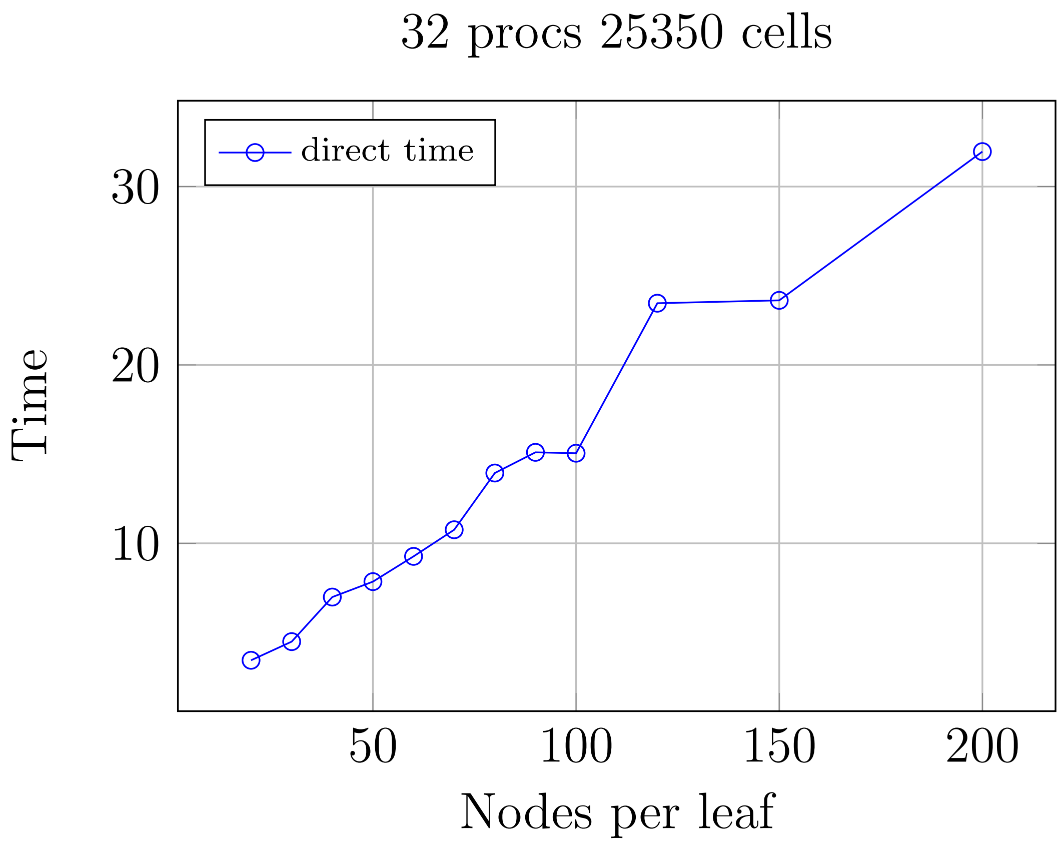

Figure 4.7: Time needed for the assembly of the direct part, needed for the resolution of nearest neighbours interactions.

4.5.2 Strong Scalability with 60 nodes per leaf

In this section we want to analyse the overall scalability of our BEM-FMA algorithm using the optimal number of nodes per leaf we have previously highlighted. In Figure 4.9 we can clearly see that our code has a suboptimal behaviour. In fact we get a worse scalability that what we got for one node per leaf. This seems to be mainly due to the worse scalability of the FMM part. We stress again that the assemble time for a BEM matrix scales almost linearly. We stress a very important difference with respect to the results of section 4.4.1. In Figure 4.10 we report the comparison between the plain BEM and the BEM-FMA with the standard implementation with 1 node per leaf. We can see that, for the number of unknowns of our study, the direct BEM represents the fastest and most accurate option. In Figure 4.11 we report the comparison between the plain BEM and the BEM-FMA

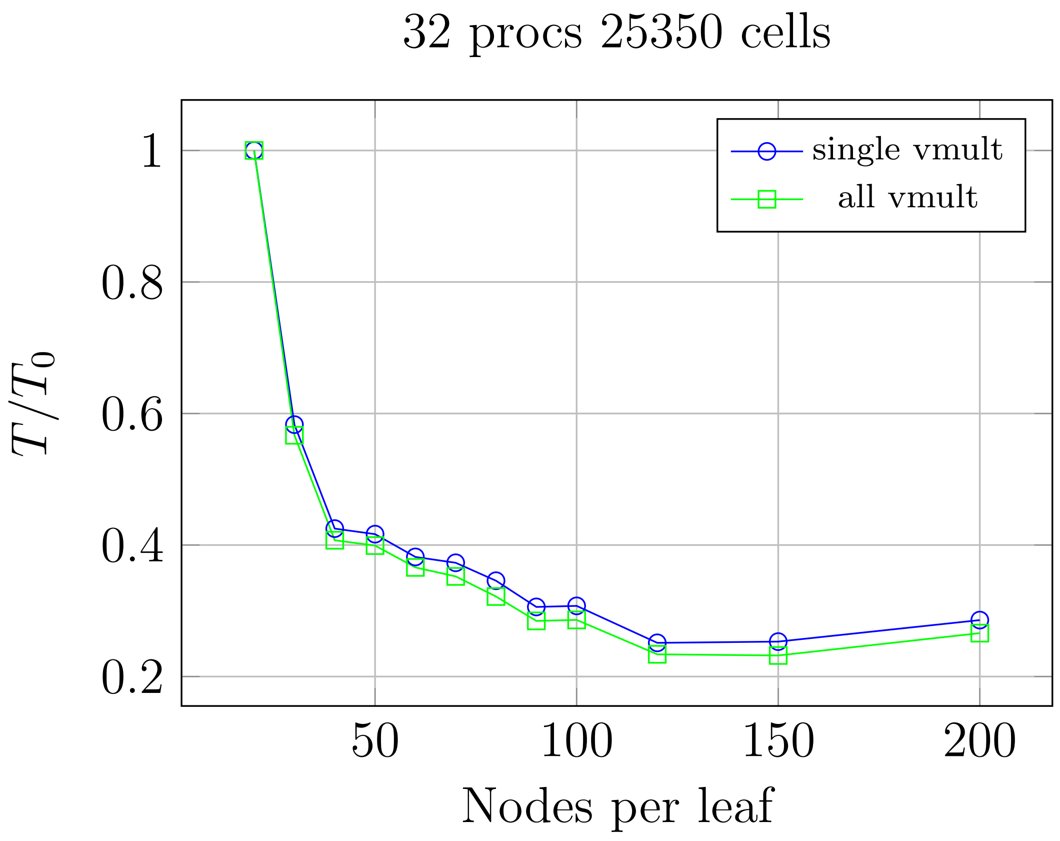

Figure 4.8: In blue with circles we can see the time needed for all the matrix vector multiplication. In green with squares we see the cost for one matrix vector multiplication.

with the 60 nodes per leaf. We can see that the times to solution is less for the BEM-FMA implementation up to 32 processors. The better performance of the BEM for 40 processors is due to its better scaling behaviour in comparison to our FMM. However we stress that even the direct BEM presents some scaling issues as we can see from Figure 3.3. Therefore we stress that the time optimal 60 nodes per leaf implementation has much better performances than the original single node per leaf one.

4.6 Memory Consumption

In this section we want to highlight another major advantage of the BEM-FMA algorithm with respect to the plain direct BEM. While in the former we compute on the fly the effect of the matrix vector multiplication in the latter we need to

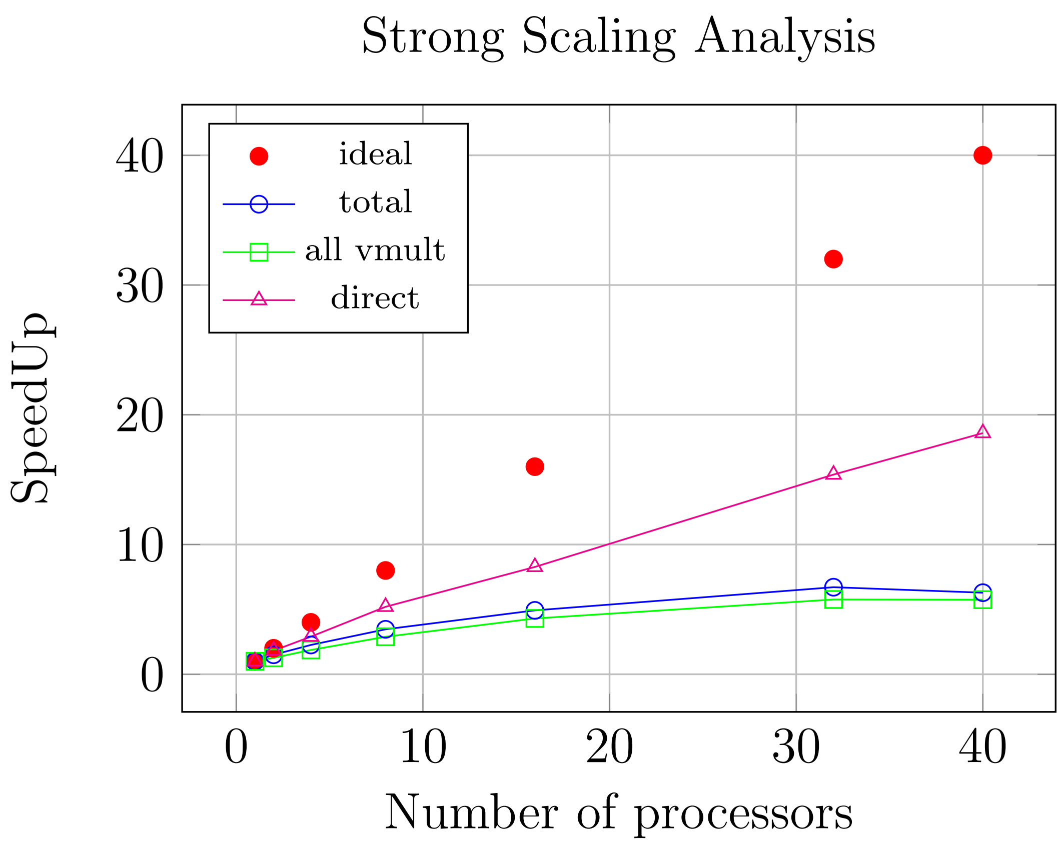

Figure 4.9: Strong Scalability test using 25350 degrees of freedom. We test up to 40 processors. We report the scaling considering the worst timing on all the used processors.In blue with circles we plot the overall scalability, in green with squares the scalability for the complete set of matrix vector multiplications, in magenta with triangles the scalability of the direct matrix assemble. The red dots represents the ideal scalability

actually allocate the memory needed by the full matrices. This may become a major issue. We have chosen to use the TrilinosWrappers of the `deal.II` library. In this framework we are allowed to write on any element of the matrix and then we communicate these effect to the proper processors. We have noticed a very specific error if we require more than 65536 degrees of freedom. The standard Trilinos implementation stores sparse matrices as a single vector indexed by an unsigned integer. This means that we have $4294967296 = 65536^2$ possible indices. In this case Trilinos is not able to compute the sparsity pattern for the distributed full matrix. The BEM-FMA implementation does not present this issue since we don't need to reserve the space for the full matrix. We can continue to use the TrilionsWrappers without any modifications. We may solve this issue compiling

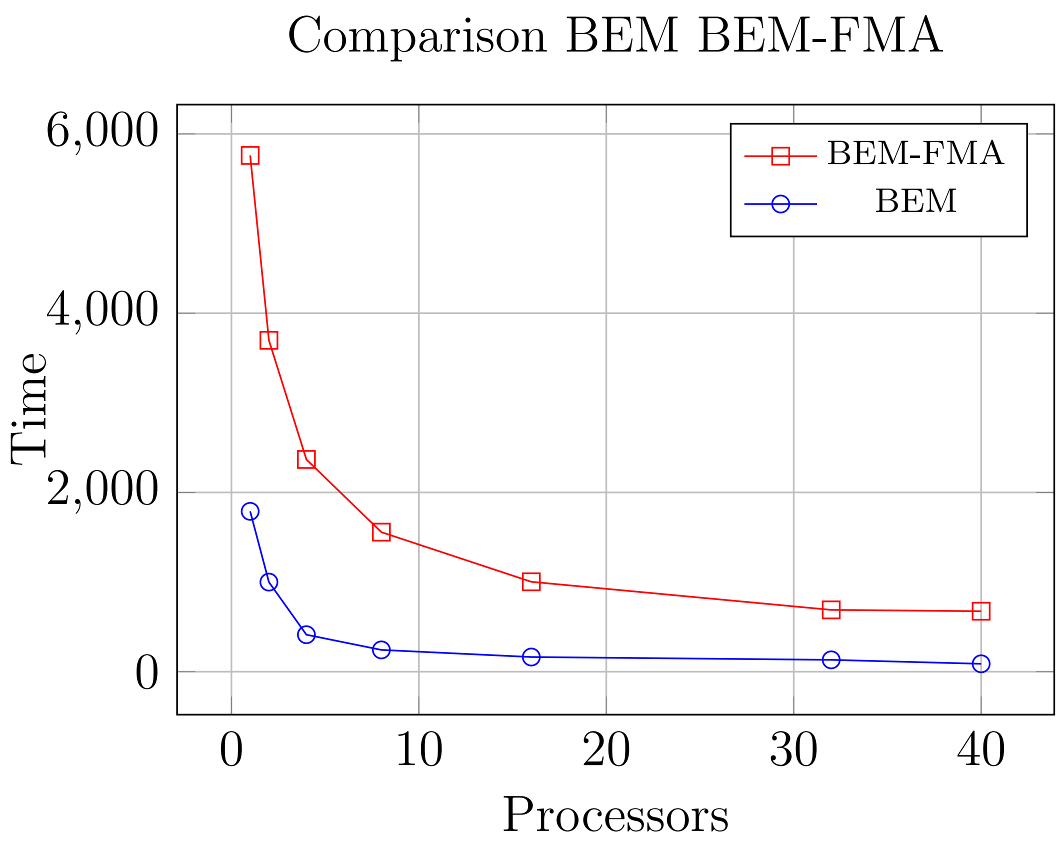

Figure 4.10: Time comparison between plain BEM and BEM-FMA with 1 node1 per leaf. In blu with circles we plot the time needed by the direct plain BEM, in red with square we report the BEM-FMA with 1 node per leaf.

the Trilinos library with 64bits indices but nevertheless we stress that we would anyway encounter this limit if we increase the number of degrees of freedom.
We stress another major disadvantage of the standard BEM approach. We have stressed the need of using already existing parallel libraries, as `deal.II` and Trilinos, to get an efficient and general subdivision among different processors. However we note that the system matrix we are assembling has a very peculiar sparsity pattern. In fact we are assembling a full matrix, its sparsity comes from the fact that we are subdividing it among different processors. It is our belief that it is very hard to optimise the memory consumption of such a matrix. In fact we have noted many problems of memory consumption in running our BEM application with 64 bit indices.

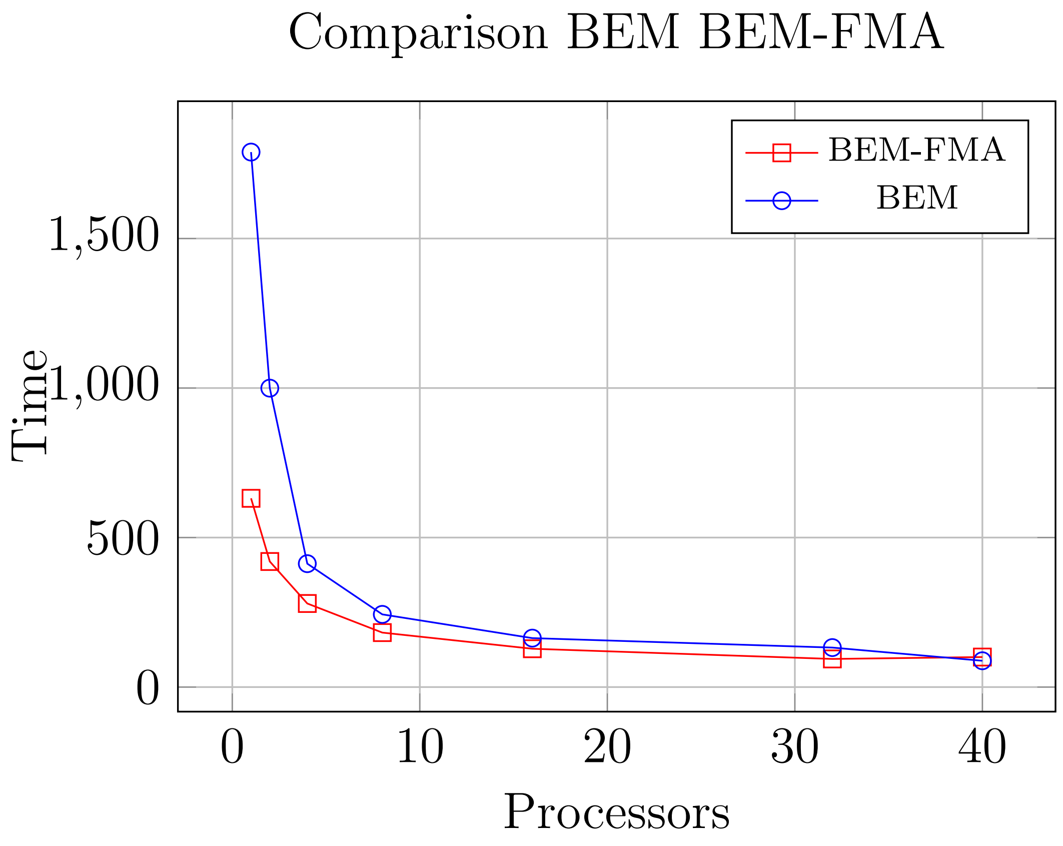

Figure 4.11: Time comparison between plain BEM and BEM-FMA with 1 node1 per leaf. In blu with circles we plot the time needed by the direct plain BEM, in red with square we report the BEM-FMA with 60 nodes per leaf.

Chapter 5

Parallel FMA with MPI and TBB

In the present chapter we present the parallelisation of the BEM-FMA algorithm using an hybrid paradigm. In the Chapter 4 we have seen that the sole MPI paradigm is not enough to efficiently parallelise the BEM-FMA algorithm. The need of a hybrid parallelisation strategy that couples MPI with multicore paradigms has been spotted in recent years in the framework of parallel multipole methods, [20]. We are dealing with small problems with respect to [20] so we are not going to deal with GPU accelerator but only with multithread paradigms on standard CPUs. We will follow the paradigm chosen in the `deal.II` library, it is called Threading Building Blocks (TBB) and is constantly developed by Intel.

5.1 Intel TBB Description

Threading Building Blocks is a widely used C++ template library for task parallelism. Its most important features are the followings

- Parallel algorithms and data structures
- Scalable memory allocation and task scheduling
- Thought for general purpose parallelism
- Supports Windows*, Linux*, OS X* and other OSes.

It is designed to work with any C++ compiler thus simplifying development of applications for multi-core systems. TBB has been designed to match with a code that is highly object oriented, and makes heavy use of C++ templates and user defined types. TBB, as native threads, do not require specific compiler support, while other paradigm, as OpenMP, do. The use of TBB does not require any specific compiler option. More generally TBB can be seen as an evolution of the standard OpenMP paradigm. It was created as something that was more conducive to the object oriented/template based programming style of C++. It has been chosen as multithread paradigm in the `deal.II` library, which is highly object oriented and makes very good use of C++ most recent features. We make use of two different functions that make use of TBB: TaskGroup and Workstream.

5.1.1 TaskGroup

The basic instruction, in TBB, is called Task. Generally it is a job that needs to be done. Tasks creation is managed by a scheduler and different task must be executed concurrently. We stress that a single thread usually has more than a single Task. It is up to the scheduler to associate each Task to a thread. Two different task can join through the TBB command join.
TaskGroup is a `deal.II` class that represents a container for different independent tasks. Its typical framework in the BEMFMA code is in the parallelisation of embarrassingly parallel for loop (in place of pragma for cycles). The typical implementation of TaskGroup is the following

- We have a loop that can be broken into completely independent tasks. We let the scheduler organise and spawn all of them.
- We end all the threads spawned in the previous step.

```
auto f_local_task = [] (...)
 {
   ...do something...
 }
 Threads::TaskGroup<> group;
 for (unsigned int kk = 0; kk <  n_task; kk++)
   group += Threads::new_task ( static_cast<void (*)(...)>
   (f_local_task)...);
 group.join_all();
```

We need to implement a function describing the work that the task needs to perform. We have chosen to use lambda functions to describe the task. Unluckily TaskGroup does not allow lambda functions as argument so we need to statically cast the lambda to a standard void function. We need to be sure that no racing condition occurs. We highlight that the TaskGroup scheduler is not able to deal with racing conditions and that we can't make any assumption on how the Task will be created. At the end of the for cycle we can use *join_all()* to end all the previously spawned tasks. From the point of view of the implementation this is a very easy solution. Basically we need only to create a function that mimes what happens inside the embarrassingly parallel loop. If race conditions occur this solution is not usable since the very easy scheduler of TaskGroup is not able to deal with them.

5.1.2 WorkStream

Modern Finite Element codes usually presents the following pattern: a stream of local independent operation followed by a reduction into a global data structure. In [19] we see how such a software pattern, called WorkStream inside the `deal.II` library, can be efficiently implemented. An explicit synchronisation of the Tasks would be quite inefficient and may not scale well. More importantly we must remember that, in floating point arithmetic, the order of a summation may lead to significant change in the result. Since we can't make any assumption on the order of creation of single Tasks we conclude that with this manual synchronisation we can't obtain the same result two times in a row.
To overcome such difficulties WorkStream separates

- The embarrassingly parallel local computations.
- The reduction operation.

The local computation can run in any order and in parallel, while the reduction operation must run on a single thread, it should avoid manual synchronisations and must perform the summation always in the same order. These constraints assures that the results are both reliable and repeatable. We stress that, for what concerns the local parallel computations, WorkStream usually schedules more tasks to the same thread in order to optimise computation time.
The typical application of such a class is the assembling a matrix in a FiniteElementMethod. In such a case we need to perform computation on each cell and then add together all the contributions inside the global matrix. The assemblage of the local contribution can be done on all the cells simultaneously. However we can't allow all the local contributions to be written at the same time on the global matrix because we would have race conditions and we would corrupt the data. Consequently, we want to ensure that only one thread at a time writes into the global matrix, and that results are copied in a stable and reproducible order. We need the following ingredients to use WorkStream

- A stream of object that is used by the scheduler to spawn the needed Tasks.
- A worker function to be run in parallel on all the objects to perform the local computations
- A copier that reduces the local contributions.

In the following we analyse the basic features of the two functions needed by WorkStream.

Worker function

WorkStream assigns one worker function per threads. We need to keep in mind that all the workers must be able to run in parallel since WorkStream does not check for any race condition at this level. In the following we list the argument of the worker object.

- Input: a ScratchData object. The worker uses this object to make its computation. The object needs to have a working copy constructor since it is copied among all threads when WorkStream is run.
- Output: a CopyData object. Every worker fills its own CopyData. This object will be pass down by reference to the copier function.
- Additional parameters: if we need to pass more informations to the work stream we have to make sure that this information is the same for all the threads. Then we have two strategies.
  - We can use a std::bind function coupled with placeholder mechanism to obtain a function that takes only ScratchData and CopyData, in this way we are binding the additional parameters to some known values
  - We can exploit a functionality of C++11, the *lambda function.* A lambda function is a function that you can write inline in your source code. It uses a so-called capture that allows the user to pass additional information to the function. In this way we can straightforwardly limit the number of parameter of the function erasing the need of a binding.

```
auto f_worker = [this,...]
(iterator, Scratch &scratch_data, Copy &copy_data)
{...}
```

Copier function

The copier function is the object that takes care of copy back the result of the worker computation back to the global memory. It takes as argument the reference to the CopyData object computed by the worker. For any additional parameter we can follow the same strategies of the worker function. The copier function automatically handles any race conditions, this means that it is responsible for any synchronisation overhead of the WorkStream class.

```
auto f_copier = [this,...]
(const &copy_data)
{...}
```

5.2 Implementation

In Section 5.1 we have discussed the basic principles of Threading Building Blocks. Now we look to our implementation in detail. We need to modify almost any function of our FMA algorithm in order to apply these principles. We have chosen to use lambda functions to implement all the TBB functions. In this way we use the captures to pass additional parameters to WorkStream making the implementation much simpler.

5.2.1 Direct Interactions

In section 4.3.1 we have analysed the implementation of the MPI parallelisation of the close range interaction. While the parallelisation with MPI was straightforward the TBB implementation is not trivial at all. Basically wee need firstly to set up the sparsity pattern of the direct interactions and then to compute the corresponding sparse matrix. At this level we don't care for any constraints, these will be applied by another method.

Sparsity pattern assembling

The worker function uses the capture to know the actual state of the class handling the FMA. In this way we can perform the computation of the column to be added at each row quite straightforwardly. Since all the workers must be able to run in parallel we must be sure that no racing condition occurs. We use the global IndexSet to know if the computation belongs to the actual processor or not, thus using a MPI strategy. Since, see 4.3.1, we have to deal with two different types of direct contributions we firstly compute the sparsity pattern due to the contributions of the blocks that are in the childlessList. and then we check for bigger blocks in the non interaction lists of other smaller blocks. Basically the workers just compute the indices of all the degrees of freedom that interacts directly and then the copiers add these indices to the sparsity pattern.

Matrix assembling

Once we have properly computed the sparsity pattern we can assemble the sparse matrix corresponding to the direct interactions. The structure of the assembling is quite complicated since each block updates all the lines of the matrix corresponding to its node. We have set up the a worker that compute all the contribution of a given block in a local dense matrix. Then the copier takes care of copying everything back to the global memory performing a so called local to global operation. Once again we need to run two WorkStream for the two different kinds of direct interactions, luckily we can use the same TBB data structures for both the implementations.

5.2.2 Ascending Phase

In seection 4.3.2 we have stressed that we don't provide any MPI parallelisation to the ascending phase. Since TBB does not require any communication strategy we can set up an effective multithread strategy for the ascending phase, see [8]. We have to provide a parallelisation for different methods.

Structure Setting

First of all we need to set up a TBB parallelisation that builds up properly the structure for the ascending phase. We recall that the sources are the quadrature points and that their intensities depend on the FiniteElement setting of the BEM. The function that sets up the strategy is called "multipole_integrals". We need to set up elemMultipoleExpansionsKer1 and elemMultipoleExpansionsKer2 these are quite complicated objects so we need great care. These objects represent the Multipole Expansions of the potential and its normal derivative. Since the creation of new elements inside a map is not thread safe we need to use WorkStream to ensure that everything is set up properly. Basically we just need to set up a structure so we can allow for an empty Scratch. The WorkStream will replace a loop over the childless blocks that used to set up all the structure. All the workers need to create their own multipole structures and then we build the copiers to copy everything in the global memory using the capture of the lambda functions. This function is called just once in the code but however represents almost half of the cost of the ascending phase so it is essential to provide a TBB parallelisation.

Multipole Expansion Creations and Translations

Once we have built the structure we can start the real ascending phase. The function is called "generate_multipole_expansions". The inputs are the values of the boundary conditions (in general the present values of the potential and its normal derivative). We need these values to assign the proper values to the sources of the multipoles. Since each matrix vector product may update the values of the potential and its normal derivative we need to call this method once per each matrix vector product.

- First of all we set up empty multipole expansions for all the blocks of the octree. This can be done with TaskGroup, see 5.1.1, since all these creations are independent and no race condition occurs.

- Now all the lower level blocks have a multipole expansion containing the contribution to the integrals of all the quadrature points in them. Now we begin summing the child multipole expansion to the the parents expansions: to do that we need to translate che children expansions to the parent block center. In this case we have to set up some synchronisation since more blocks may have the same parent and this would lead to race conditions among thread. Thus we need WorkStream to copy all the expansion up along the tree. We have modified the MultipoleExpansion class in order to be able to set up the center and then if two expansions have the same centre we don't need any translation but we just sum the expansion coefficients.
  - Since the worker function actually runs in parallel we let it take care of all the translation of the blocks to a local expansion centred in the parent block centre. So the worker actually performs the multipole translations
  - The copier functions just add the local coefficients to the real parent exapansion, inside the global memory. In this way the copier has to deal only with the synchronisation of all the threads and almost does not perform any additional work.

In this way we have been able to set up a TBB parallelisation that exploits both the strategies explained in 5.1 that minimises the synchronisation time of the copier funcions. In such a way we should avoid, for the problem sizes we are dealing with, the bottleneck of the ascending phase highlighted in Figure 4.5.

5.2.3 Descending Phase

The function that takes care of the descending phase is called "multipole_matr_vect_products" and actually finalises all the matrix vector products. The method is called once per every matrix-vector product. We need to call this function after having generating the multipole expansions. Since this function takes a considerable amount of time we have chosen to parallelise it using multithreaded TBB on a single node and MPI to allow even a greater level of parallelism. In this case we don't need any communication because we have made sure that the multipole expansions are replicated on each node. Thus we can safely split the descending phase.

Local Expansion Creations and Translations

First of all we need to create all the empty expansions for all the blocks. This is an embarrassingly parallel operation that we can perform using the ThreadGroup strategy without requiring any synchronisation time. In order to perform the descending phase properly we need WorkStream. Inside the worker we check the IndexSet for the MPI parallelisation. The scratch is empty once again since this class only has to copy from parents to children. As in the ascending phase we use captures in order to pass global information to the worker and the copier. For each level of the tree we uses a WorkStream to loop over all the blocks at the current level.

- The worker creates a local LocalExpansion and then it translates the LocalExpansion of the parent block. Then we convert the MultipoleExpansion of blocks of the same size of the current block to the local expansion. With these two steps we have taken care of the translation of the local expansion. The worker needs to compute also some contribution to the final results of the matrix vector product. We have to loop over blocks in the non interaction list that are smaller than the current block: in this case the bound for the conversion of a multipole into local expansion does not hold, but the bound for the evaluation of the multipole expansions does hold. Thus, we will simply evaluate the multipole expansions of such blocks for each node in the block. Also these results are stored in a local array.
- The copier function just add the local Local Expansion to the actual expansions in the global memory handling, once again, all the race conditions. It also copies the local contribuition to the matrix vector multiplication given by the Multipole expansion that we could not translate into a Local expansion.

Local Expansion Evaluation

Finally, when the loop over levels is done, we need to evaluate local expansions of all childless blocks, at each block node(s). This is an embarrassingly parallel operation so it can be easily performed using ThreadGroup. We also check the IndexSet to perform the mixed TBB-MPI parallelisation.

5.2.4 Final Preconditioner Setting

We have seen in Section 4.3.4 that it is quite easy to apply a MPI parallelisation. Unluckily a multithread implementation is not so straightforward since the setting of the final preconditioner has race conditions. We need, once again, WorkStream to handle them.

Final Preconditioner Sparsity Pattern

We set up the same worker copier strategy we have used for the first preconditioner setting inside the close range function, see 5.2.1. The worker memorise the indices to be add per block and then the copier set up the real preconditioner sparsity pattern.

Final Preconditioner Assembling

We just need to loop over all the degrees of freedom. Firstly we check that the unknown belongs to the processor and then we fill the preconditioner depending on the constraints of the current unknown: if it is constrained we use the Constraint-Matrix, otherwise we just copy the values of the first preconditioner we computed. This is embarrassingly parallel so we can safely use ThreadGroup. Lastly we can add the values of α to thepreconditioner, once again we can safely do this operation with ThreadGroup providing a good MPI-TBB parallelisation.

5.3 Optimal Block Size Analysis

In Section 4.5.1 we have seen that is possible to recover an optimal size of the blocks in the tree by means of nodes in the childless blocks. At the time being we have recovered a full MPI-TBB parallelisation and we try to recover a more rigorous analysis.

5.3.1 Theoretical analysis

We follow [8] in order to achieve an optimal block size. We define the following quantities:

$$\begin{aligned} N =& \text{ number of unknowns,} \\ B =& \text{ number of unknowns in a leaf,} \\ p =& \text{ number of MPI processors,} \\ t =& \text{ number of TBB threads.} \end{aligned} \tag{5.1}$$

We introduce the time needed by the ascending phase as the sum of the time needed to compute the multipole at childless level and the M2M translation time

$$T_{asc} = K_1 \frac{N}{t} + K_2 \log\left(\frac{N}{B}\right) \frac{N}{Bt}. \tag{5.2}$$

We can introduce the time of the descending phase as sum of the time for M2L and L2L translations plus the evaluation time at childless level

$$T_{desc} = K_3 \log\left(\frac{N}{B}\right) \frac{N}{Btp} + K_4 \frac{N}{tp}. \tag{5.3}$$

We can define the time needed to compute the short range interactions as

$$T_{dir} = K_5 \frac{NB}{pt}. \tag{5.4}$$

Lastly we introduce the communication-synchronisation time as

$$T_{comm} = e(B, p, t), \tag{5.5}$$

that we can't value as precisely as we did for the others. The constant K_1, K_2, K_3, K_4, K_5 are constant depending on the precision requested to the FMA and the floating point unit. We will try to determine them experimentally in the next section. In order to find a optimum of the block size we compute the partial derivative of all the timings with respect to B

$$\begin{aligned}
\frac{\partial T_{asc}}{\partial B} &= \frac{K_2 N}{B^2 t} \left(\log(B) - \log(N) - 1\right), \\
\frac{\partial T_{desc}}{\partial B} &= \frac{K_3 N}{B^2 pt} \left(\log(B) - \log(N) - 1\right), \\
\frac{\partial T_{dir}}{\partial B} &= \frac{K_5 N}{pt}, \\
\frac{\partial T_{comm}}{\partial B} &= \frac{\partial e(p,t,B)}{\partial B} \sim 0.
\end{aligned} \tag{5.6}$$

Following [8] we have made the assumption that the communication time derivative is small compared to the others. This assumption is needed if we want to recover a theoretical analysis. We introduce the overall time derivative as

$$\frac{\partial T_{tot}}{\partial B} = \frac{1}{t} \left(\left(\frac{K_2 N}{B^2} + \frac{K_3 N}{B^2 p} \right) \left(\log(B) - \log(N) - 1\right) + \frac{K_5 N}{p} \right). \tag{5.7}$$

Thus we can conclude that the optimal block size B_{opt} is such that

$$\left(\left(\frac{K_2 N}{B_{opt}^2} + \frac{K_3 N}{B_{opt}^2 p} \right) \left(\log(B_{opt}) - \log(N) - 1\right) + \frac{K_5 N}{p} \right) = 0. \tag{5.8}$$

From (5.8) we can see first very important conclusion, that makes our analysis agree with [8], B_opt will not depend on the number of threads but only on the number of processors. This is due to our MPI parallelisation strategy that involves only the short range interactions and the descending phase. In order to recover an analytic condition we made another assumptions

$$\log(B_{opt}) - \log(N) - 1 \sim -log(N). \tag{5.9}$$

In order to justify such an hypothesis we remember that that all the logarithms in our analysis are logarithms base 8. Thus we expect $\log(B) \sim 1$. With (5.9) we can conclude

$$\left(-\frac{K_2 N}{B_{opt}^2} - \frac{K_3 N}{B_{opt}^2 p}\right)\log(N) + \frac{K_5 N}{p} = 0, \tag{5.10}$$

so we can derive

$$B_{opt} = \sqrt{\frac{(K_2 + K_3/p)\log(N)}{K_5/p}}. \tag{5.11}$$

As expected our optimum depends on the number of MPI processors we uses.

5.3.2 Block Size Results

In this section we analyse the timings of our hybrid MPI TBB implementation for a computation with 25350 unknowns. We uses a single MPI processor and we exploit the maximum number of threads on Ulysses. We recall that on a standard node of such a machine we have 20 processors on a single node so we can uses up to 20 parallel threads. We have chosen to test block size from 20 to 140 nodes per block. We report a major detail in order to run properly on Ulysses. Since we want to spawn the maximum number of threads on each node we must ensure that the scheduler don't put two MPI processors on the same node if a free one is available, we need to add the option *pernode*. To make sure that each MPI processor is free to spawn threads on different core we need to make sure that no existing configuration limits its possibility, in order to do so the complete command to run the code is

mpirun -np n_{mpi} -pernode –bind-to none bemexe $n_{threads}$,

we have set up the code to accept the number of threads as additional command line input. If no argument is passed the code uses the maximum number possible.

Ascendent Time

In this section we analyse the computational time needed by the ascendent phase and its relation with the block size B. As time we have considered the following average

$$T_{sac}^{true} = \frac{T_{setting} + T_{rising}}{n_{GMRES}}, \tag{5.12}$$

that comprehends both the time of the setting of the Multipole Expansions and the true rising phase. We have taken the overall average in order to study the time needed for a single matrix vector product. We have applied a standard fitting in order to recover the constant in (5.2). We obtain the following results

$$\begin{aligned} K_1 &= 6.02144325217e-05, \\ K_2 &= 0.000214458307065. \end{aligned} \tag{5.13}$$

In Figure 5.1 we compare our theoretical result with the experimental results We

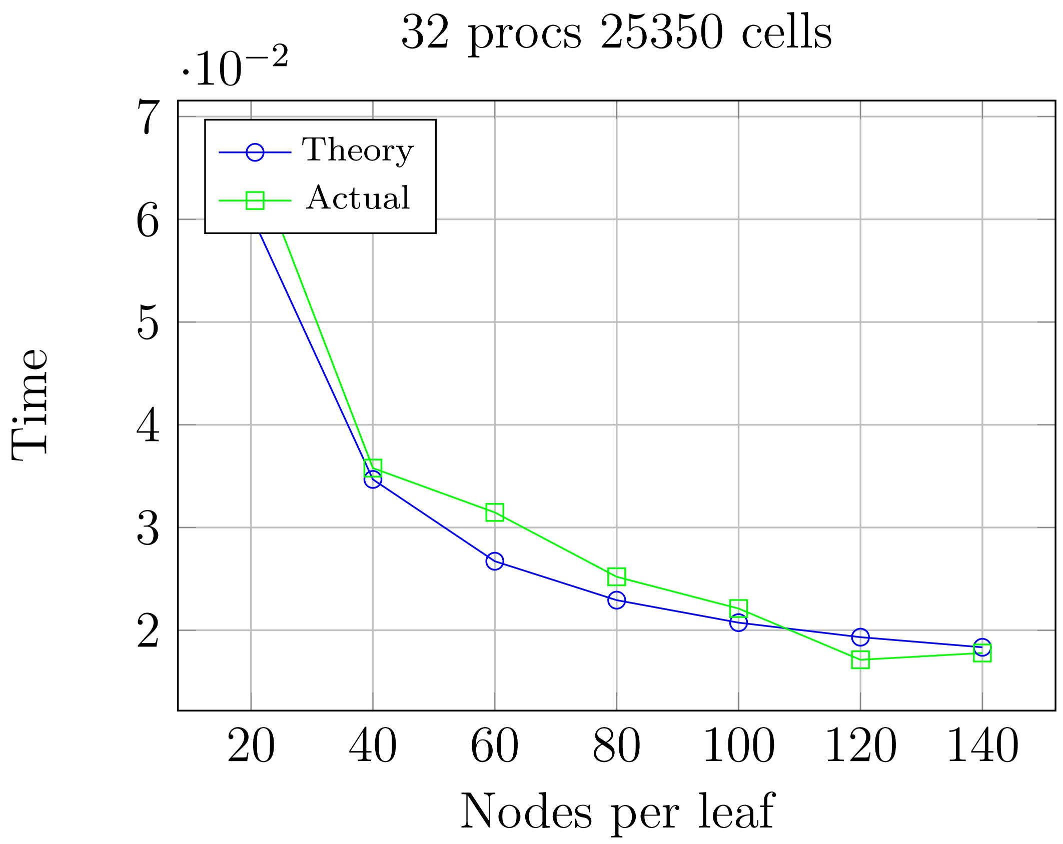

Figure 5.1: Analysis of the time needed by the Ascendent phase of the FMA. In blu with circles we report our theoretical result, in green with squares we can see the experimental results.

can see a pretty good agreement with the theory we have developed and the actual results. We stress that we have chosen to take an overall average of the timings even if the setting time is required only once and it takes almost the same time of all the rising phases. In particular we can see that the time needed by the ascendent phase decreases as the block size increase. This is rather obvious since as B increases the overall number of blocks decreases and so does the time needed by the ascending phase.

Descendent Time

In this section we analyse the computational time needed by the ascendent phase and its relation with the block size B. As time we have considered the average per iteration needed by the iterative solver. With usual fitting techniques we get the following constant for (5.3)

$$\begin{aligned} K_3 &= 0.355460533753, \\ K_4 &= 0.0184668562545. \end{aligned} \tag{5.14}$$

In Figure 5.2 we compare our theoretical result with the experimental results We

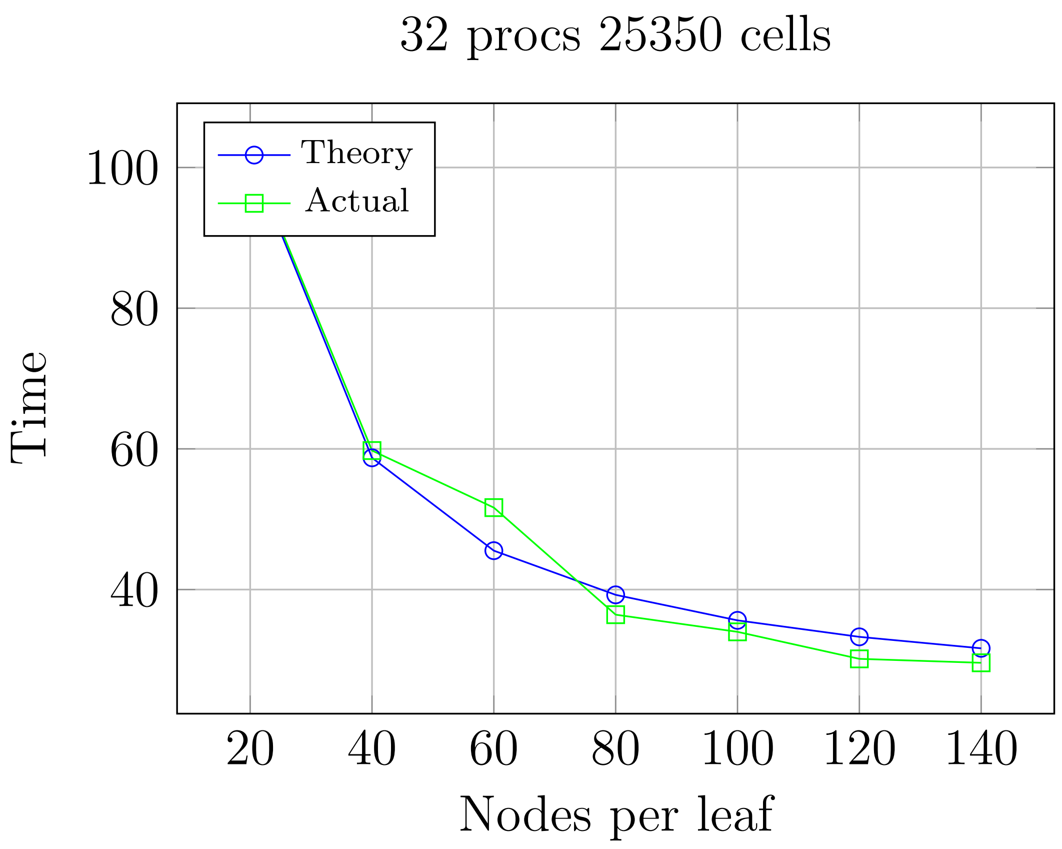

Figure 5.2: Analysis of the time needed by the Descendent phase of the FMA. In blu with circles we report our theoretical result, in green with squares we can see the experimental results.

can see an extremely good agreement between the experimental results and our simple theory. In particular we can see that the descendent time needed by the algorithm decreases as the block size increase. This is rather obvious since as B increases the overall number of blocks decreases and so does the time needed by the descending phase.

Direct Time

In this section we analyse the computational time needed by the short range interaction and its relation with the block size B. Since these interactions are computed once we don't need any average. With usual fitting techniques we get the following constant for (5.3)

$$K_5 = 0.0017587037037. \tag{5.15}$$

In Figure 5.3 we compare our theoretical result with the experimental results We

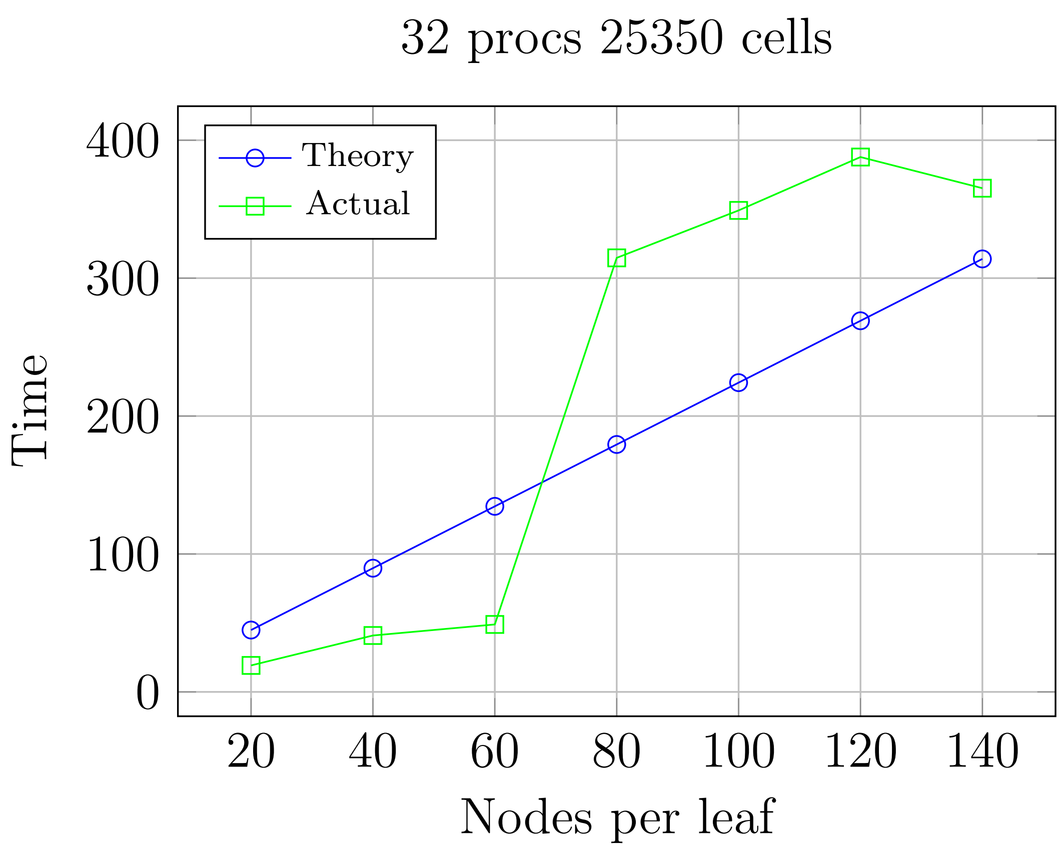

Figure 5.3: Analysis of the time needed by the Direct phase of the FMA. In blu with circles we report our theoretical result, in green with squares we can see the experimental results.

can see that we can't recover a good agreement. This is due to the extremely complicated structure needed to compute the short range interactions. We have parallelised it using WorkStream but we can clearly see that for $B > 60$ we get a lot of overhead to synchronisation time needed to avoid racing conditions. In order to have a better estimate of the direct time we compare the time needed to compute the short range interactions and the one needed to compute the final preconditioner. In Figure (5.4) we see the results We can see a quite good agreement

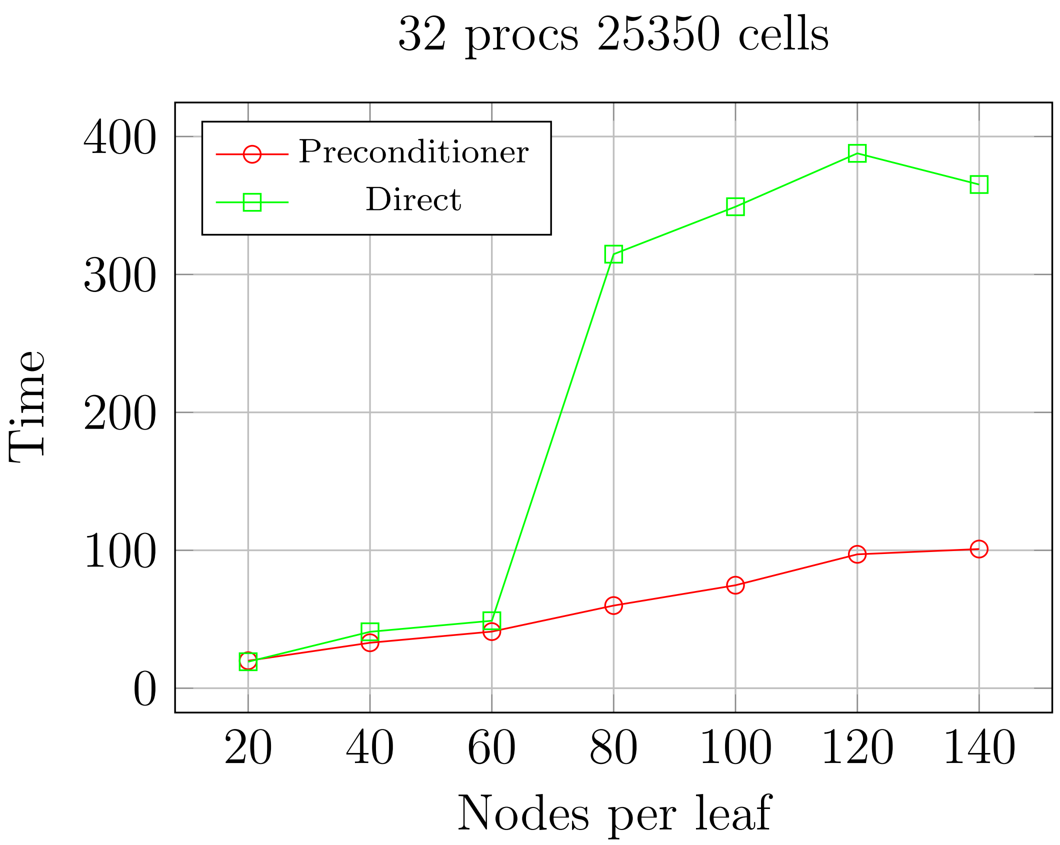

Figure 5.4: Analysis of the time needed by the Direct phase of the FMA and the time needed to set up the final preconditioner. In green with squares we report the time needed by the short range interactions, in red with circles we can see the time needed for the final preconditioner.

if $B \leq 60$. This is due to the fact that both the functions computes operation of order NB. If we increase the block size we can see that the easier structure of the preconditioner setting allows for reduced synchronisation timings. We believe that we can safely use the time needed by the preconditioner to estimate the constant

K_5 getting

$$K_5 = 0.000628596949891. \tag{5.16}$$

In Figure 5.5 we compare the corrected theoretical result with the direct timing We can see a quite good agreement for small block sizes. We stress that since the

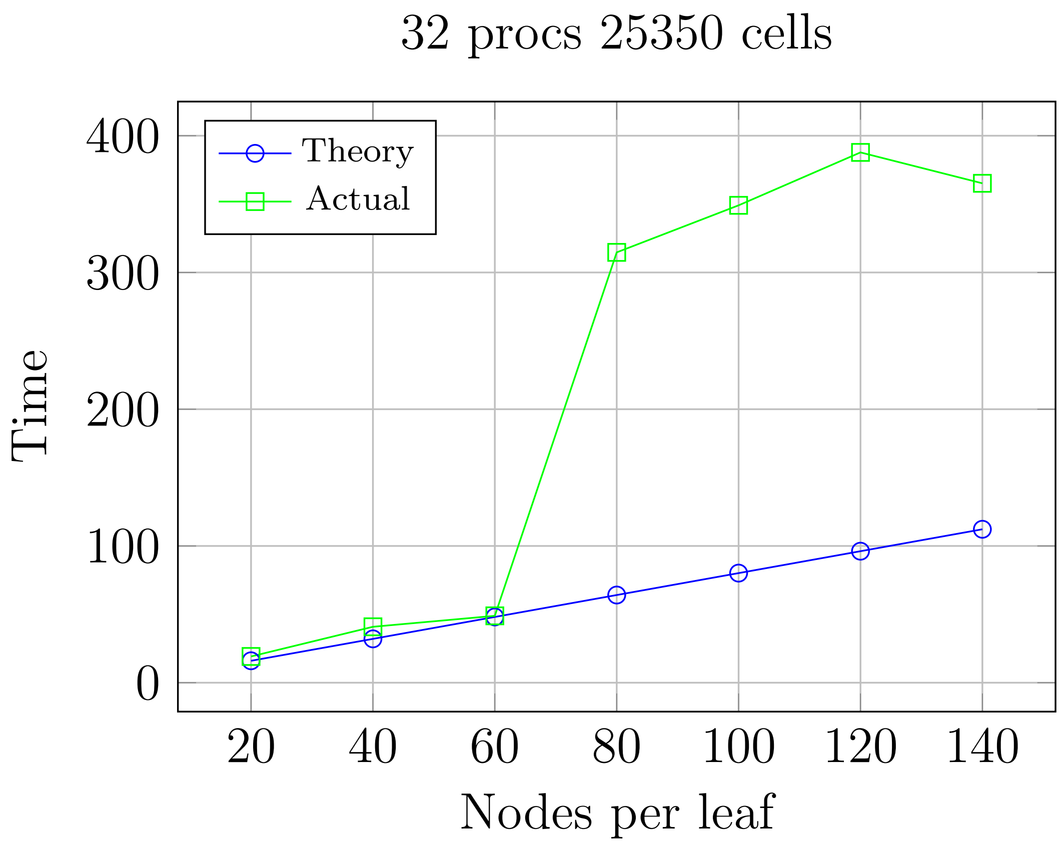

Figure 5.5: Analysis of the time needed by the Direct phase of the FMA. In blu with circles we report our theoretical result using the precondiotener timings, in green with squares we can see the experimental results.

experimental minus is between 40 and 60 we are interested in a good estimated in such a size range.

Overall Comparison

In this section we report a comparison between the overall timings we can recover from our theory and the actual experiments. In Figure 5.6 we can see the

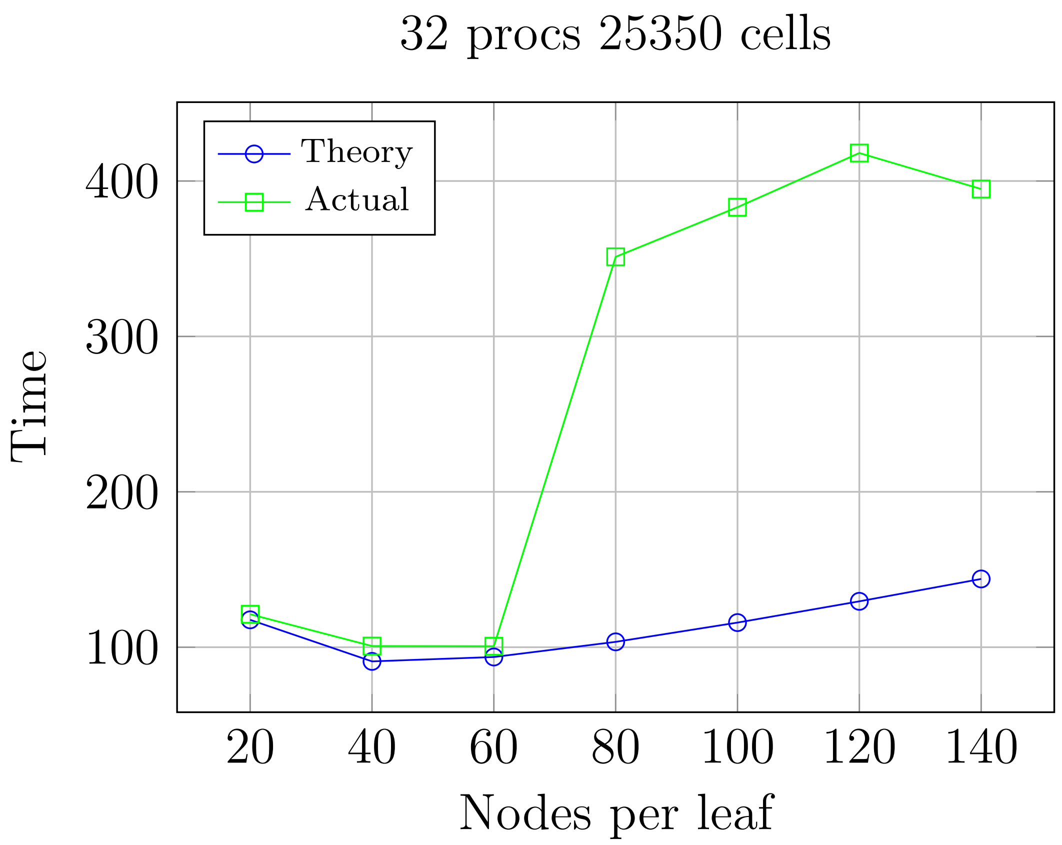

Figure 5.6: Analysis of the time needed by the overall FMA. In blu with cirlces we report our theoretical result using the precondiotener timings, in green with squares we can see the experimental results.

results Once again we can see quite a good agreement for small block sizes. With our theory we get

$$B_{opt} = \sqrt{\frac{(K_2 + K_3/p)\log(N)}{K_5/p}} = 52.5440911254, \tag{5.17}$$

which agrees with experimental results that show the minimum for $40 < B_{opt} < 60$. In the following section we will therefore take $B = 60$ in order to have a comparison with the results of Section 4.5.2.

5.4 Strong Scaling Analysis

In this section we analyse the Strong Scaling of our BEM-FMA algorithm with hybrid TBB-MPI parallelisation up to 2 nodes, 40 processors, on Ulysses. We consider two different scenario in order to assert the scalability dependence on the degrees of freedom of our problem.

5.4.1 Strong Scaling with 25350 dofs

We will study the problem with 25350 degrees of freedom. We consider 2 MPI processes and up to 20 TBB threads per process. We can see that our method does

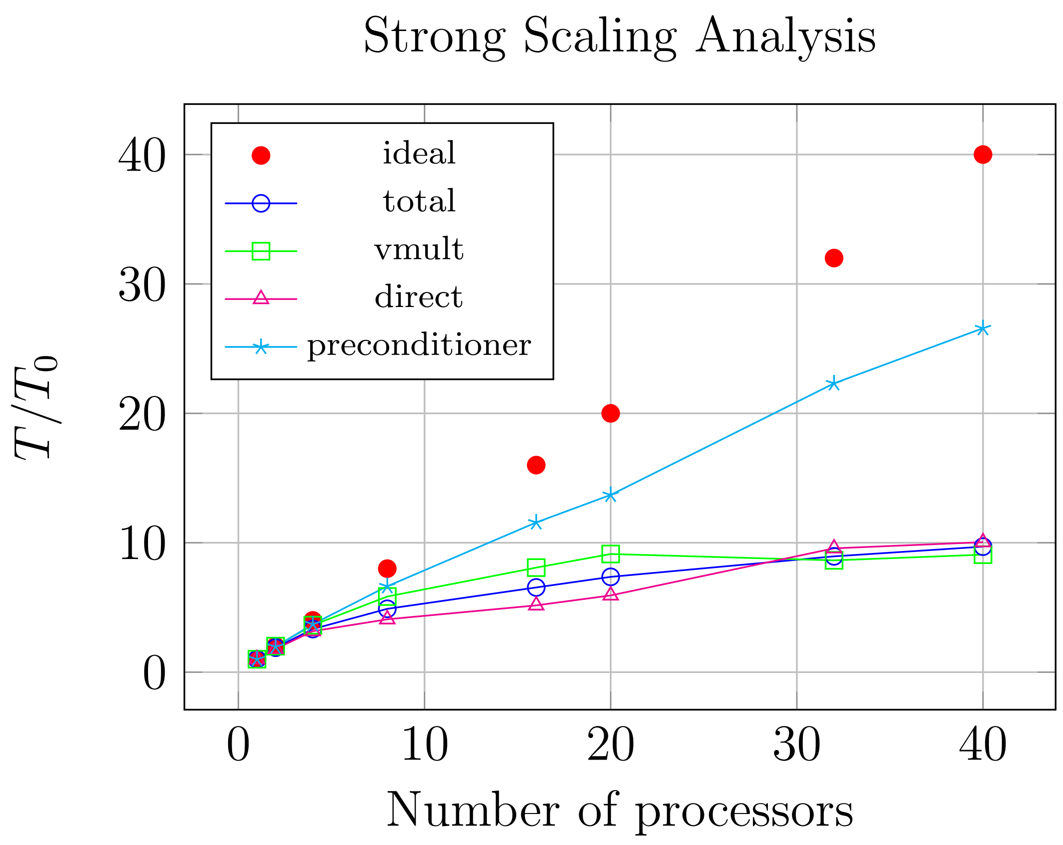

Figure 5.7: Strong Scalability test using 25350 degrees of freedom. We test up to 40 processors. We report the scaling considering the worst timing on all the used processors. In blue with circles we plot the overall scalability, in green with squares the scalability for the complete set of matrix vector multiplications, in magenta with triangles the scalability for the setting time of FMM, in cyan with stars the scalability of the preconditioner building method. The red dots represents the ideal scalability

not scale ideally. In particular we recover some scalability issue for the short range interactions. As we stressed in Section 5.3.2 this is due to the complex pattern of the method that computes such interactions. In particular we can see that the preconditioner method, that compute the same number of operation but has a simpler structure, scales almost ideally. We stress that for this kind of interaction we recover a far better scalability with a pure MPI parallelisation, see Figure 4.9. If we compare our results in Figure 5.7 and we compare them with Figure 4.9 we can see that we achieve much better performances with the new TBB-MPI parallelisation, a scalability of 9.69 against 6.3. In particular if we consider only the performances of the FMM by means of a single matrix vector multiplication we get Figure (5.8) We can see that with the current setting we are able to get

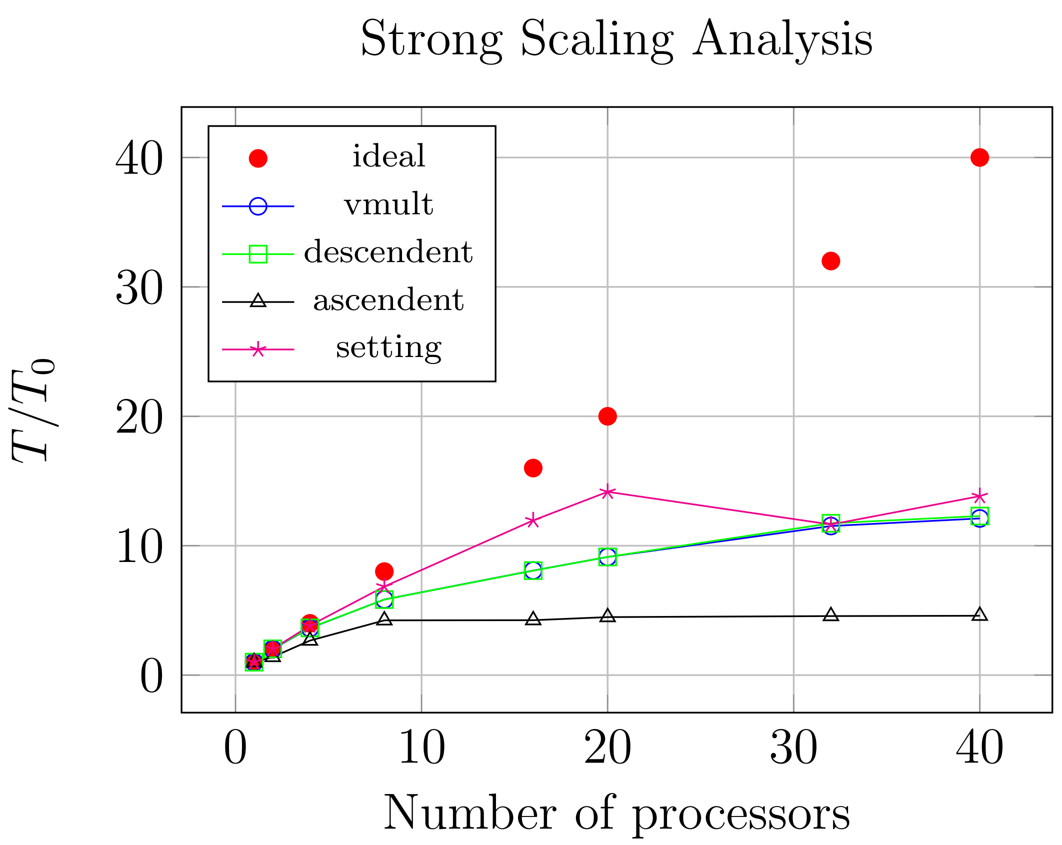

Figure 5.8: Strong Scalability test for a single FMM matrix vector multiplication using 25350 degrees of freedom. We test up to 40 processors. We report the scaling considering the worst timing on all the used processors. In blue we plot the overall scalability, in green the scalability descending phase of matrix vector multiplications, in magenta the scalability of the setting time and in black the scalability of the ascending phase of the algorithm. The red dots represents the ideal scalability

an overall scalability for a single matrix vector multiplication of 11.83 against the ideal 40. At the present point we want to assert whether this is due to the reduced number od degrees of freedom or if it is an issue of our method.

5.4.2 Strong Scaling with 98306 dofs

In this section we consider a more refined problem with 98306 degrees of freedom. Since the computational cost is increased we expect our method to scale better that in the previous section. We can see that the increased computational

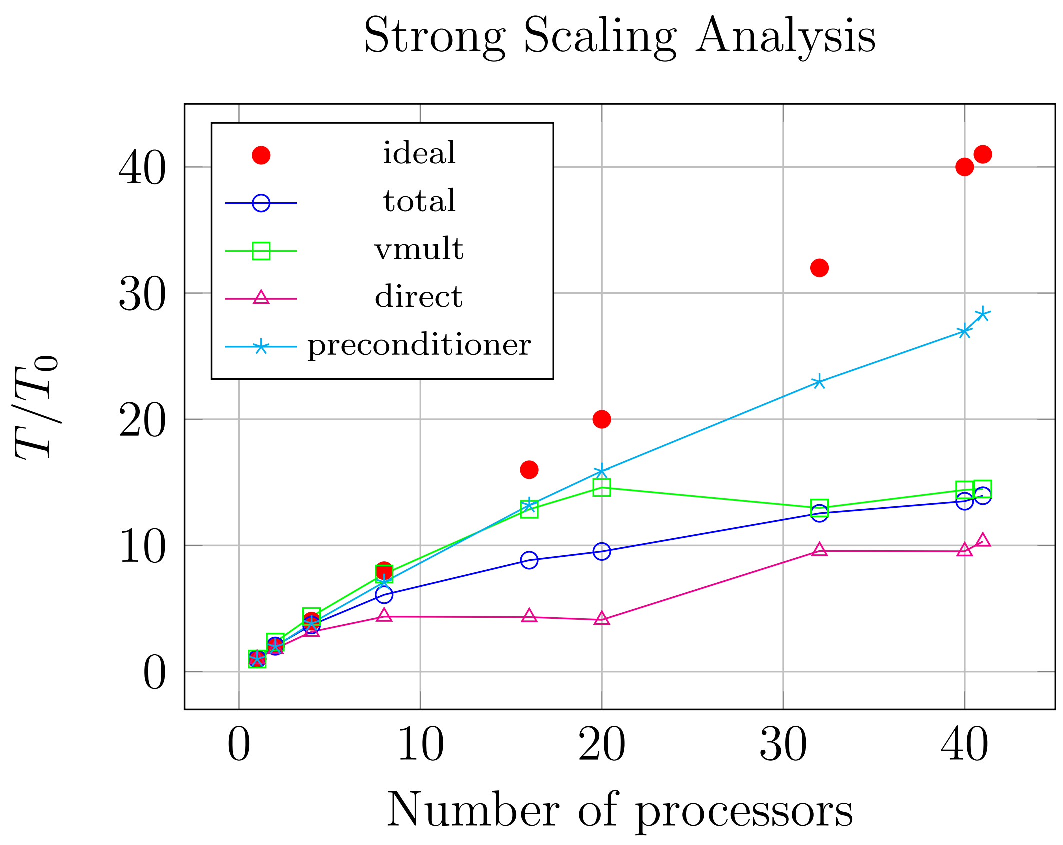

Figure 5.9: Strong Scalability test using 25350 degrees of freedom. We test up to 40 processors. We report the scaling considering the worst timing on all the used processors. In blue we plot the overall scalability, in green the scalability for the complete set of matrix vector multiplications, in magenta the scalability for the setting time of FMM, in cyan the scalability of the preconditioner building method. The red dots represents the ideal scalability.

cost has led to increase scalability performances since we are able to get a scalability of 13.93 against 9.69 we got in the previous section. Once again we can see the scalability issue for the short range interactions. We can see however that the preconditioner method scales almost optimally. We stress that the performance of the matrix vector multiplication may be induced by the augmented number of iteration needed by the iterative solver when MPI is involved. In order to verify this fact in Figure 5.10 we study the scalability of a single matrix vector multiplication. We can see that we get a much better scalability when we consider a single

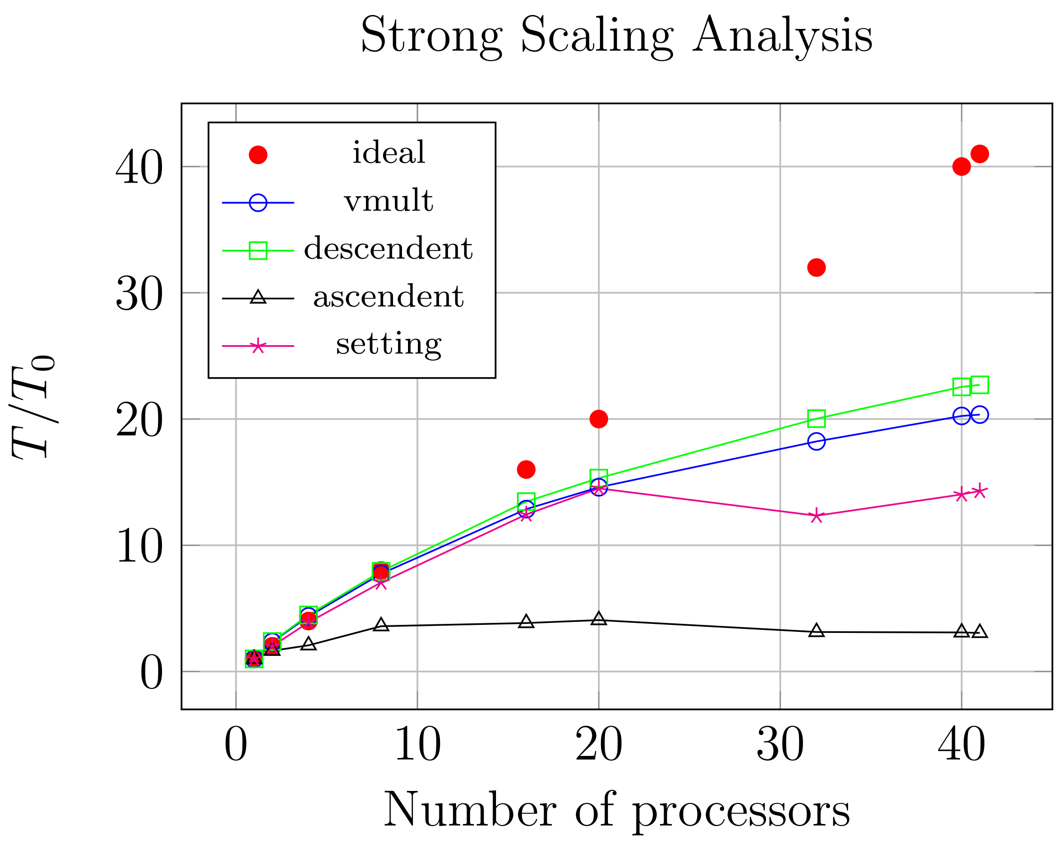

Figure 5.10: Strong Scalability test for a single FMM matrix vector multiplication using 98306 degrees of freedom. We test up to 40 processors. We report the scaling considering the worst timing on all the used processors. In blue with circles we plot the overall scalability, in green with squares the scalability descending phase of matrix vector multiplications, in magenta with stars the scalability of the setting time and in black with triangles the scalability of the ascending phase of the algorithm. The red dots represents the ideal scalability.

matrix vector multiplication. This means that the augmented computational cost has effectively increased the performances of our hybrid MPI-TBB code. In particular we can get a speed up of 20.36, which is almost the double of what we got in the previous section.

5.5 Weak Scaling Analysis

In this section we report two different weak scalability analysis. In Section 5.5.1 we consider a cost of $O(N^2)$, while in Section 5.5.2 we consider a computational cost of $O(N)$.

5.5.1 Computational Cost $O(N^2)$

In this section we keep constant the amount of computations for each processor. Since we are performing a Boundary Element Method we consider a cost of $O(N^2)$. We are executing global refinements, thus we increase by a factor 16 the number of processors per cycle. This force us to perform a very coarse analysis since we use 16 nodes on Ulysses cluster at the last cycle. Moreover, since we use 393216 cells, we stress that we use nodes with 160 GigaBytes of RAM since we are allocating very big matrices for the preconditioner and the direct contributions. We report in Figure 5.11 the weak scalability analysis for our BEM-FMA algorithm.

We can see that we are able to get an optimal weak scalability up to 16 processors. This is due to the fact that we are fully exploiting the TBB parallelism on a single computational node. In fact, we can see that both the vmult time and the direct one are scaling both than optimally. This is due to two different motivations: firstly the FMM method has a computational cost of $O(N)$, secondly TBB cost may vary greatly depending on the number of threads. The first reason explains the behaviour of vmult time, in fact, it is scaling with $O(N)$ leading to a superoptimal result considering a cost of $O(N^2)$. The second motivation explains instead the non linear behaviour of the direct integral function. It has a computational cost of $O(N^2)$, but it is able to produce a superoptimal result for the TBB internal characteristics.
However we can see that when we go for a MPI parallelisation we get much worse results. As we have spotted out in Section 5.4, our parallel BEM-FMA does not scale optimally, and we can see a confirmation from the weak analysis. In order to get a better representation of the performances we report the analysis for a single vmult operation in Figure 5.12.

We see that when we use only TBB paradigm we reach a superoptimal behaviour while when we use more than one computational node we report clearly suboptimal behaviour. We can see however that we recover the same parallelisation efficiency in weak and strong scalability. In fact, we get an efficiency of roughly 30% in both cases.

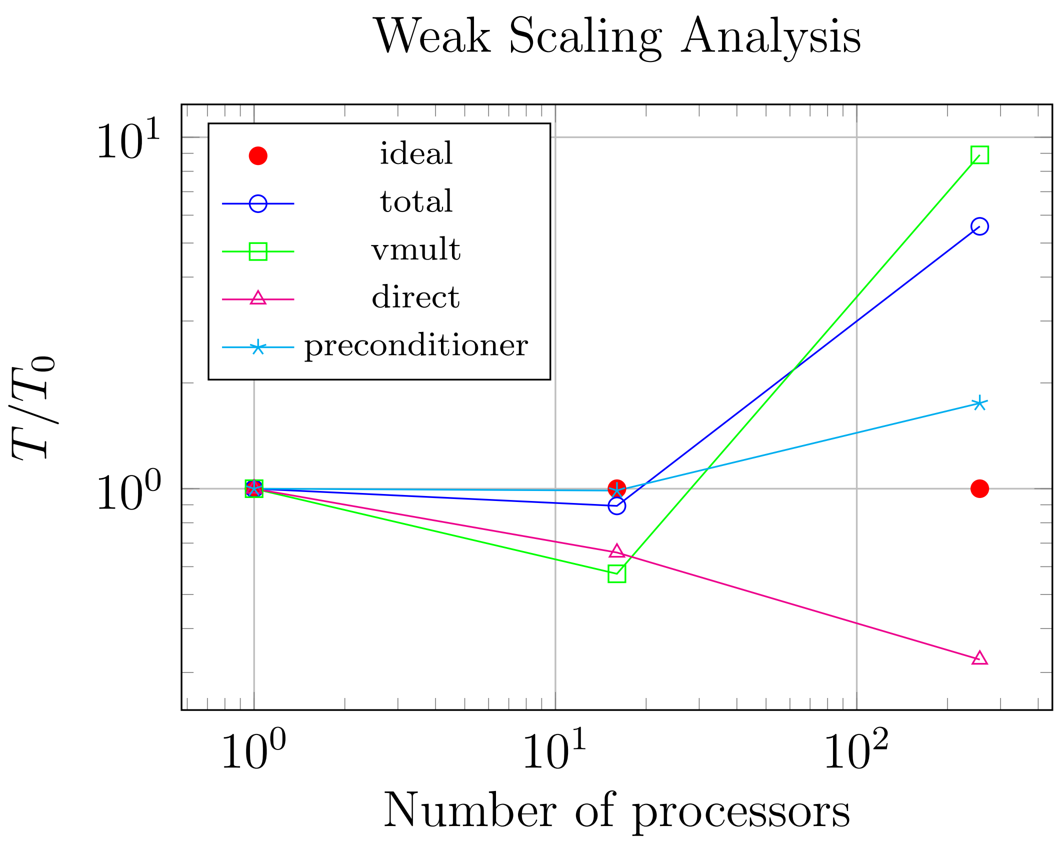

Figure 5.11: Weak Scalability test for BEM FMA algorithm up to 393218 degrees of freedom. We test up to 256 processors. We report the scaling considering the worst timing on all the used processors. In blue we plot the overall scalability, in green the scalability for the complete set of matrix vector multiplications, in magenta the scalability for the setting time of FMM, in cyan the scalability of the preconditioner building method. The red dots represents the ideal scalability.

5.5.2 Computational Cost $O(N)$

In this section we analyse the weak scalability of our BEM-FMA code considering its theoretical computational cost $O(N)$. We have stressed in Section 5.5.1 that we have a good scalability when we use a multithread paradigm, thus we expect to recover the same in the present analysis. We report the overall weak scalability results in Figure 5.13. While the time needed for the preconditioner assembling has clearly a non optimal behaviour, we can see that up to 16 threads the BEM-FMA algorithm presents a reasonable suboptimal weak scalability. We must keep in mind that we are increasing the number of degrees of freedom together with the

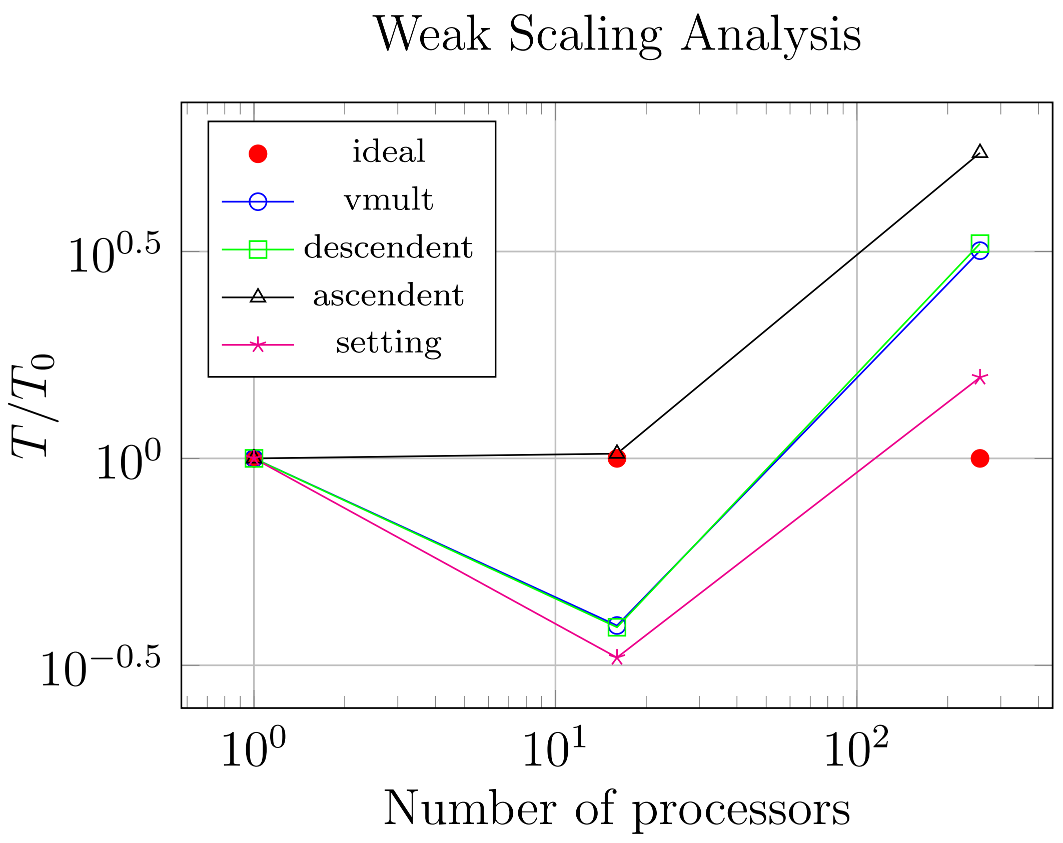

Figure 5.12: Weak Scalability test for a single FMM matrix vector multiplication using up to 393218 degrees of freedom. We test up to 256 processors. We report the scaling considering the worst timing on all the used processors. In blue with circles we plot the overall scalability, in green with squares the scalability descending phase of matrix vector multiplications, in magenta with stars the scalability of the setting time and in black with triangles the scalability of the ascending phase of the algorithm. The red dots represents the ideal scalability.

number of processors, thus the number of iterations required to solve the linear system increase too. Therefore it is more significant to analyse the weak scalability of a single vmult operation, as depicted in Figure 5.14. We see that the FMM part of the code has a good weak scalability considering a multithread paradigm. We can clearly see that when we use the hybrid MPI-TBB strategy, thus when we require more than 16 processors, we no more have a computational order of $O(N)$. This may be due to two main reasons: we introduce some communication overhead in the multipole algorithms, the ascending phase is not parallelised with MPI paradigm.

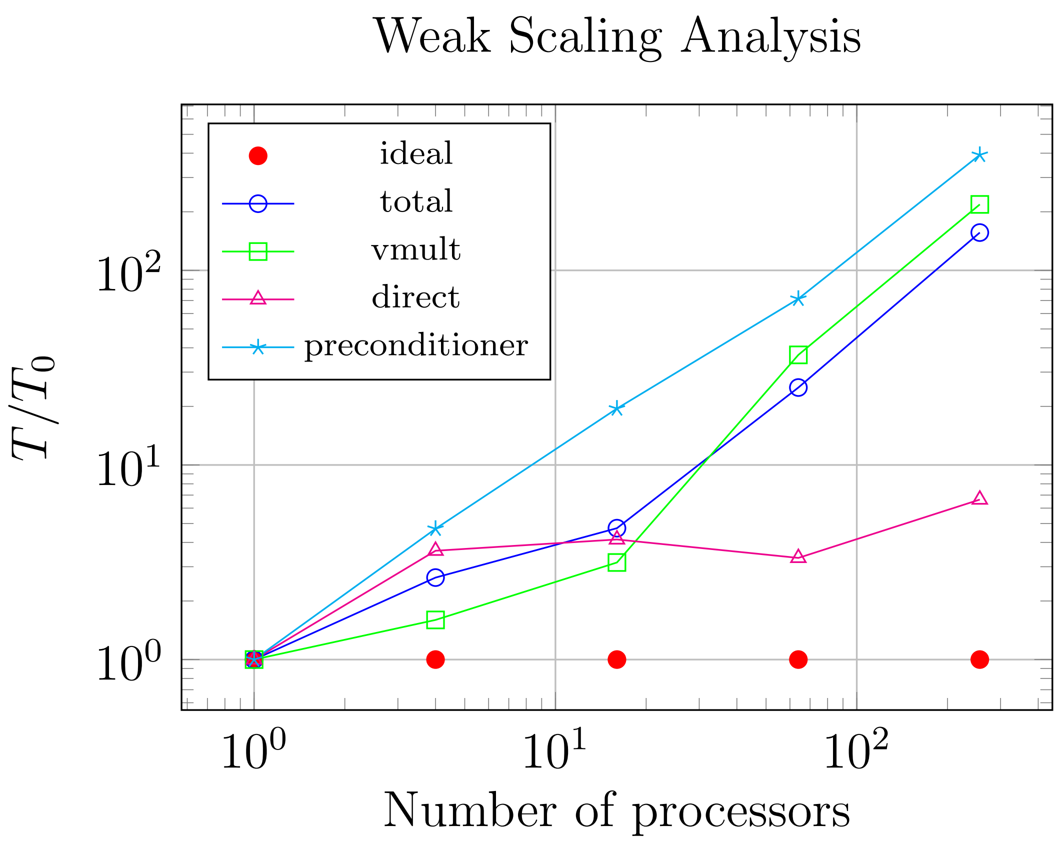

Figure 5.13: Weak Scalability test for BEM FMA algorithm up to 393218 degrees of freedom. We test up to 256 processors. We report the scaling considering the worst timing on all the used processors. In blue we plot the overall scalability, in green the scalability for the complete set of matrix vector multiplications, in magenta the scalability for the setting time of FMM, in cyan the scalability of the preconditioner building method. The red dots represents the ideal scalability.

5.6 Comparison with MPI algorithm

In this Section we want to sum up our considerations regarding the parallelisation of the BEM-FMA algorithm. In Section 4.5.2 we have seen that the sole MPI paradigm is not enough to reach an effective scalability. In fact, we have assessed that the ascending phase of the algorithm quickly becomes a bottleneck.
The usage of a hybrid parallelisation, see [8, 20] strategy combining multithread and MPI is necessary to get a proper code parallelisation. Therefore we have chosen to implement a multithread strategy for the entire FMM using Intel Threading Building Block that ensures a very high reliability, and has been developed explicitly for Object Oriented C++ library. We have maintained the same MPI

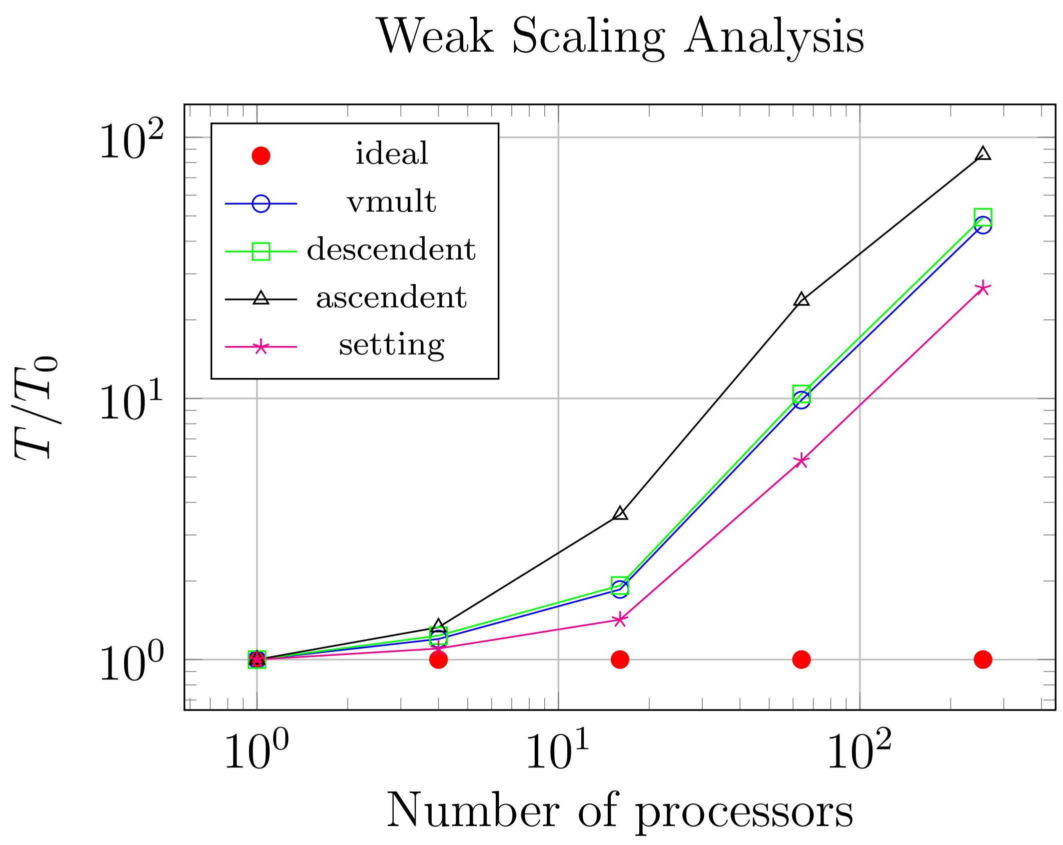

Figure 5.14: Weak Scalability test for a single FMM matrix vector multiplication using up to 393218 degrees of freedom. We test up to 256 processors. We report the scaling considering the worst timing on all the used processors.In blue with circles we plot the overall scalability, in green with squares the scalability descending phase of matrix vector multiplications, in magenta with stars the scalability of the setting time and in black with triangles the scalability of the ascending phase of the algorithm. The red dots represents the ideal scalability.

parallelisation as in Chapter 4. We have seen a substantial improvement in our results since we are able to get a strong scalability efficiency of about 30%.
Moreover, the weak scalability analysis of Section 5.5 has clearly pointed out that our BEM-FMA code has quite good performance as long as we can increase the thread to be used. However we have noted that the structure of the short range interactions in the FMM is not straightforward to be parallelised. If we consider Figure 5.7 we can see that the direct interaction present a suboptimal behaviour while in Figure 3.3 we see that the same kind of interactions scales almost optimally. This difference is due only to the code structure, in the FMM case we have to assess the block, inside the octree, to be considered in such operation, this

is an extremely complex action and actually TBB needs a lot of synchronisation overheads to handle it. In standard BEM we can manually split the cycle over the degrees of freedom and we can fully exploit the embarrassingly parallel structure of our equations.

Chapter 6

Conclusions

We have developed an efficient parallel solver for the Laplace equation by means of a Boundary Element Method. The code is able to automatically subdivide the computational domain between an arbitrary number of processors. This has been possible with the usage of METIS partitioning tool, which is wrapped inside the `deal.II` library. Then we have used the Trilinos library to manage to communication pattern between the processors.
The parallelisation strategy we have proposed is particularly effective for the assemblage of the system matrix. We have observed an optimal strong scalability up to an arbitrary number of processors. We stress that at the time being the real bottleneck of the BEM is becoming the solver cycle. A research of an optimal preconditioner for the system matrix is already underway.
In order to achieve a higher level of computational efficiency we have proposed the coupling with a Fast Multiple Method. We have seen that, in order to achieve a proper parallelisation of the BEM-FMA algorithm we need a hybrid multithread multiprocessor strategy. We have developed a full parallelisation using MPI together with Intel Threading Building Blocks.
Finally we have carried out an optimal condition for the setting of the FMM. We have proved that such a setting increases dramatically the efficiency of our code. We believe that a further efficiency improvement may come from the parallelisation of the hierarchical tree. At the moment a feasibility study for such an issue in underway. We want to stress that the only loss of accuracy of our BEM-FMA method is the order of the expansion we use to approximate the kernel. We don't use any cut off radius to speed up the computations and this makes our code extremely accurate.
Finally we stress that our code makes extensive use of parameter files. In fact it is possible to set up a new simulation simply changing such files. This has been possible with the usage of the `deal2lkit` library. This is essential in order to reduce the compilation timings. Moreover it makes our code easier to use. We

believe that this is an essential tool in order to reach a wider range of users, which is one of the most important principles we depicted in Section 3.1.

Bibliography

[1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries. In *Modern Software Tools for Scientific Computing*, pages 163–202. Birkhäuser Boston, Boston, MA, 1997.

[2] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and T. D. Young. The \texttt{deal.II} Library, Version 8.2. *Archive of Numerical Software*, 3(1), Dec. 2015.

[3] L. Barba and R. Yokota. How Will the Fast Multipole Method Fare in the Exascale Era? *SIAM News*, 46(6):8–9, 2013.

[4] C. A. Brebbia. *The Boundary Element Method for Engineers.* Pentech Press, 1978.

[5] C. Fochesato and F. Dias. A fast method for nonlinear three-dimensional free-surface waves. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 462(2073):2715–2735, 2006.

[6] V. K. George Karypis. A fast and high quality multilevel scheme for partitioning irregular graphs. 20(1):359–392, 1998.

[7] N. Giuliani, A. Mola, L. Heltai, and L. Formaggia. Engineering Analysis with Boundary Elements FEM SUPG stabilisation of mixed isoparametric BEMs : Application to linearised free surface fl ows. *Engineering Analysis with Boundary Elements*, 59:8–22, Oct. 2015.

[8] L. Greengard and W. Gropp. A parallel version of the fast multipole method. *Computers & Mathematics with Applications*, 20(7):63–71, 1990.

[9] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987.

[10] S. T. Grilli, P. Guyenne, and F. Dias. A fully non-linear model for three-dimensional overturning waves over an arbitrary bottom. *International Journal for Numerical Methods in Fluids*, 35:829–867, 2001.

[11] M. a. Heroux, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, K. S. Stanley, R. a. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, and R. P. Pawlowski. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, 2005.

[12] M. Kronbichler, T. Heister, and W. Bangerth. High accuracy mantle convection simulation through modern numerical methods. *Geophysical Journal International*, 191(1):12–29, 2012.

[13] I. Lashuk, G. Biros, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, and D. Zorin. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Ieee-Micro*, 55(5):101, 2012.

[14] A. Mola. *Model for Olympic Rowing Boats*. PhD thesis, Politecnico di Milano, 2009.

[15] V. Rokhlin. Rapid Solution of Integral Equations of Classical Potential Theory. *Journal of Computational Physics*, 60(1983):187–207, 1983.

[16] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[17] A. Sartori, N. Giuliani, M. Bardelloni, and L. Heltai. deal2lkit : a Toolkit Library for High Performance Programming in deal . II. *submitted*, pages 1–26, 2015.

[18] J. C. F. Telles. A Self-Adaptive Co-ordinate Transformation For Efficient Numerical Evaluation of General Boundary Element Integrals. *International Journal for Numerical Methods in Engineering*, 24:959–973, 1987.

[19] B. Turcksin. WorkStream a design pattern for multicore-enabled finite element computations. *submitted*, pages 1–24, 2015.

[20] R. Yokota and L. Barba. A Tuned and Scalable Fast Multipole Method as a Preeminent Algorithm for Exascale Systems. 2011.