FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Domain Decomposition for the Wave Equation

Jonas Thies

Bachelorarbeit

Domain Decomposition for the Wave Equation

Jonas Thies

Bachelorarbeit

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Ph.D. F. Hülsemann
Bearbeitungszeitraum:	20.5.2003 – 20.8.2003

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 20. August 2003 ....

Abstract

Domain decomposition methods are suitable methods for the parallel solution of partial differential equations. In this thesis I will introduce the basic theory of the overlapping additive Schwarz method and describe the practical implementation as a preconditioner for the conjugate gradient method. As model problems I am using the Poisson problem and the wave equation. Numerical results will be given for convergence rates and performance in different settings.

Contents

1 Introduction **5**

2 Theory of domain decomposition methods **6**
2.1 Definitions . . . 6
2.2 The iterative Schwarz methods . . . 7
2.3 Domain decomposition on the discrete level . . . 8
2.4 Multi-domain formulation . . . 10
2.5 Convergence . . . 11

3 The sequential solver **13**
3.1 Discretization . . . 13
3.2 Boundary conditions . . . 13
3.3 Linear algebra classes . . . 14
3.4 The solver . . . 14

4 Implementation of a parallel framework for domain decomposition **15**
4.1 Parallelization concept . . . 15
4.2 Decomposing the domain . . . 17
4.2.1 Data structures for the partitioned graph . . . 17
4.2.2 Creating overlap . . . 19
4.2.3 Performance of the partitioning class . . . 20
4.3 Linear algebra classes . . . 21
4.3.1 Parallel vector class . . . 21
4.3.2 Parallel matrix class . . . 22
4.4 Communication . . . 23
4.5 Additive Schwarz preconditioner . . . 24
4.6 Numerical problems with the penalty method . . . 25

5 Numerical results and conclusions **26**
5.1 Test cases . . . 26
5.2 Convergence behavior . . . 27
5.3 Performance . . . 30
5.4 Concluding remarks . . . 33

1 Introduction

In many applications of modern science and engineering the numerical core consists of the solution of large, sparse linear systems of equations, which arise, for example, from the discretization of partial differential equations (PDEs). Highly efficient techniques for their solution have been developed over the past decades. In most cases they employ iteration schemes in order to approximate the exact solution to the system. Examples of modern solvers are the Krylov subspace methods (CG, GMRES, ...), multigrid techniques and suitable cross-breeds of the two such as multigrid preconditioners for the conjugate gradient method.
On the other hand the development of the computers on which the algorithms are implemented has to be taken into account. The fast pace of progress both of the numerics and the hardware is, however, somewhat levelled by the increasing complexity and size of the problems to be solved.
In order to cope with large scale applications often the only possibility is to use a supercomputer. In this field the course fairly seems to be set. Basically two types of systems are currently dominating the field: Networks of cheap off-the-shelf processors and clusters of shared memory nodes. For the purpose of this thesis it is not of importance which alternative is chosen since both are essentially distributed memory systems if it comes to large scale applications.
And this is where the problems start. Most of the powerful numerical techniques briefly mentioned before can not straightforwardly be implemented on a distributed memory computer and even if ways are found parallelization is often tedious work and good performance is not guaranteed beforehand.
Wouldn't it be nice to just let every processor/SMP-node solve a part of the global system using an existing powerful sequential solver? Although the theory is somewhat more involved, this is basically what domain decomposition (DD) methods are all about:

1. Partition the computational domain into N parts
2. Solve local subproblems in parallel using a sequential solver on every processor
3. Enforce certain smoothness conditions on the interfaces between subdomains

DD methods come in many different flavors:
In step 1 the decomposition may be based on physical aspects of the problem such as different materials or different equations to model the problem in different parts. Also, the essential choice has to be made if the subdomains should overlap and, if so, how much overlap is advantageous. Large regions of overlap increase the convergence rates of the methods since the coupling between the domains becomes stronger, but they also reduce the efficiency of parallelization since parts of the domain have to be treated by several processors at the same time. If the subdomains do not overlap at all other ways of communication have to be used to ensure convergence towards the correct global solution. In general, one has to solve a special interface problem after every step. Steps 2 and 3 are often embedded in an iterative procedure. Although direct DD methods exist (often called sub-structuring techniques) they are not further discussed in this thesis. Finally, not all DD methods have as great a potential for parallelization as implied up to now, as we will see later on.
The method I want to study in particular - the overlapping additive Schwarz method - goes

back to H. A. Schwarz in 1870. It is an iterative process based on overlapping subdomains, and in its simplest form it is inherently parallel.
In the next section I will provide a mathematically founded introduction to domain decomposition, and especially the overlapping additive Schwarz method. The third section will describe the sequential program which our implementation is based on: the discretization techniques used and the sequential solver intended for the subdomain solves.
At that point the thesis will turn towards more practical implementation issues. In section 4 the implementation of a parallel framework for domain decomposition is treated in detail. The final chapter then yields numerical results concerning convergence and performance issues, concluding remarks and possible improvements to the implementation.

2 Theory of domain decomposition methods

This section is meant to give an introduction to the theory of domain decomposition algorithms, especially the Schwarz iterative methods. The notation and content follow the works of Quarteroni and Valli [1] and Chan and Mathew [2].

2.1 Definitions

Throughout the following chapter I will discuss DD methods for an arbitrary elliptic partial differential equation in d dimensions (d=2,3):

$$\mathcal{D}\hat{u}(x) = \hat{f}(x), x \in \Omega, \Omega \subset \mathbb{R}^d \tag{2.1}$$

with
$\mathcal{D}$: the elliptic differential operator
$\hat{u}$: the unknown quantity
$\hat{f}$: the inhomogeneity

Suppose we want to find a solution to (2.1) on the domain Ω with given conditions on the boundary of Ω, denoted by $\partial\Omega$. For the moment let us assume Dirichlet boundary conditions:

$$\hat{u}(x) = \hat{g}(x), x \in \partial\Omega \tag{2.2}$$

We now partition Ω into two overlapping subdomains Ω_1 and Ω_2 such that $\Omega_1 \cup \Omega_2 = \Omega$ and $\Omega_1 \cap \Omega_2 \neq \emptyset$. That means the partitions completely cover the domain and have at least a small region of overlap.
Both subdomains now have their own boundaries $\partial\Omega_1$ and $\partial\Omega_2$. Furthermore we shall denote by Γ_1 and Γ_2 those parts of $\partial\Omega_1$ and $\partial\Omega_2$ resp. which lie in the interior of the respective other subdomain (see figure 2.1 for a classical example).

$$\Gamma_1 = \partial\Omega_1 \cap \Omega_2 \tag{2.3}$$
$$\Gamma_2 = \partial\Omega_2 \cap \Omega_1 \tag{2.4}$$

We can now finally define the local subproblems $\mathcal{D}\hat{u}_1 = \hat{f}$ on Ω_1 and $\mathcal{D}\hat{u}_2 = \hat{f}$ on Ω_2, which will be well-posed as long as valid boundary conditions are specified on $\partial\Omega_1$ and $\partial\Omega_2$ respectively.

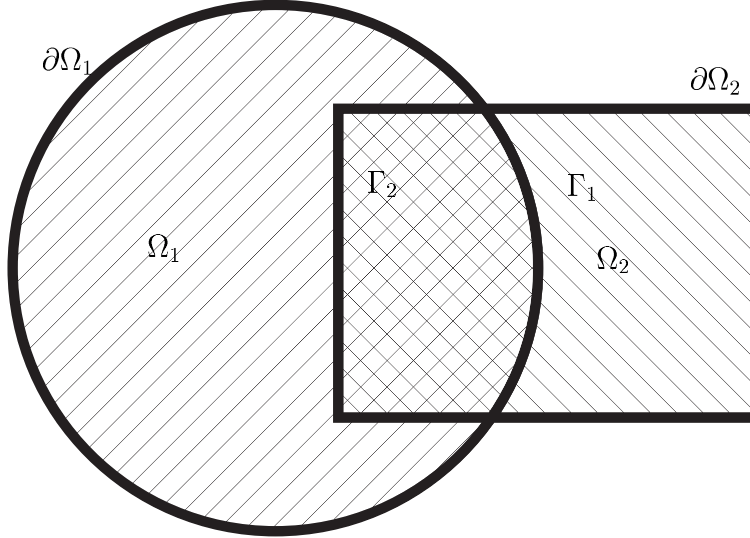

Figure 2.1: overlapping two-way partitioning of a domain

2.2 The iterative Schwarz methods

The classical DD algorithm to solve equation (2.1) is the alternating (multiplicative) Schwarz method. It solves the two subproblems in an alternating way, always using the most current approximation of u as Dirichlet boundary condition on Γ_1 and Γ_2 resp., and the global boundary conditions on $\partial\Omega_1 \setminus \Gamma_1$ and $\partial\Omega_2 \setminus \Gamma_2$.

Algorithm 2.1 continuous multiplicative Schwarz

Choose an initial guess $\hat{u}^{(0)}$
while approximation not good enough **do**
 solve
 $\mathcal{D}\hat{u}_1^{(k+1)} = \hat{f}$ in Ω_1
 $\hat{u}_1^{(k+1)} = \hat{g}$ on $\partial\Omega_1 \setminus \Gamma_1$
 $\hat{u}_1^{(k+1)} = \hat{u}^{(k)}$ on Γ_1
 and
 $\mathcal{D}\hat{u}_2^{(k+1)} = \hat{f}$ in Ω_2
 $\hat{u}_2^{(k+1)} = \hat{g}$ on $\partial\Omega_2 \setminus \Gamma_2$
 $\hat{u}_2^{(k+1)} = \hat{u}_1^{(k+1)}$ on Γ_2
 set $\hat{u}^{(k+1)} = \begin{cases} \hat{u}_1^{(k+1)} & \text{on } \Omega_1 \\ \hat{u}_2^{(k+1)} & \text{on } \Omega \setminus \Omega_1 \end{cases}$
end while

The disadvantage of algorithm (2.1) is that the two subdomain solves have to be carried out one after the other, which means that the method is only as parallel as the solvers employed on the subdomains are. However, with a slight modification the so-called additive Schwarz method can be derived. It uses the last global approximation for the boundary conditions on both subdomains, so that the subdomain solves can be performed in parallel - possibly at the cost of a slight decrease in the numerical efficiency since the boundary conditions of the subdomains are not quite up to date.

2.3 Domain decomposition on the discrete level

In order to actually use the technique of domain decomposition on a computer, a discrete formulation has to be derived. We therefore assume from now on that equation (2.1) has been discretized by finite differences (FD), finite elements (FE) or any other grid-based method, yielding a (potentially large) sparse linear system of equations

$$Au = f, A \in \mathbb{R}^{n\times n}, u, f \in \mathbb{R}^n \tag{2.5}$$

In our case let A be a symmetric and positive definite real-valued matrix. The domain Ω is now represented by an index set I containing the n indices of the nodes of the FD/FE mesh. An overlapping partitioning is achieved by introducing the subsets I_1 and I_2 of sizes n_1 and n_2 respectively, such that $I_1 \cup I_2 = I$, $I_1 \cap I_2 \neq \emptyset$, resp. $n_1 + n_2 > n$.

If we now renumber the indices in I_1 and I_2 so that those global indices on the region of overlap get the highest local indices in I_1 and the lowest indices in I_2, it is possible to display the system (2.5) as shown in figure 2.2. The renumbered (local) index sets are denoted by L_i (i=1,2).

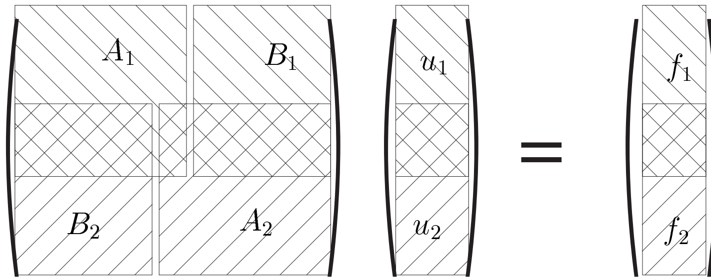

Figure 2.2: two-way partitioned linear system with overlap

The subdomain matrices A_1 and A_2 retain the properties of A: they are positive-definite and symmetric, which means that a sequential solver written to solve (2.5) may as well be used to solve the subdomain problems

$$A_1 u_1 = f_1 \tag{2.6}$$
$$A_2 u_2 = f_2 \tag{2.7}$$

The subproblems can be solved faster since they are substantially smaller than the original system. The overlapping subvectors v_1 and v_2 of a global vector v can be obtained by applying to v the *restriction operators* R_i (i=1,2).

$$v_1 = R_1 v \ , \ \ v_2 = R_2 v \tag{2.8}$$

R_i are orthogonal matrices of dimension $n_i \times n$ containing the entries

$$(R_i)_{jk} = \begin{cases} 0: & \text{j} \notin I_i \\ 1: & \text{j corresponds to local index k in } \mathrm{L_i} \end{cases} \tag{2.9}$$

Each R_i has at most one entry equal to 1 per row and column, which means that it is a bijective mapping from the subset I_i of I to the set of local indices L_i. The transpose ${R_i}^T$ is called *extension operator*. It has the inverse effect: The local indices of v_i are mapped to the global indices of v . The resulting vector has dimension n and contains the value 0 at nodes outside the subdomain i.
With these definitions the relation between the subdomain matrices A_i and the global matrix A can be written as

$$A_1 = R_1 A {R_1}^T \ , \ A_2 = R_2 A {R_2}^T$$

We can now write the Schwarz methods presented in the last section in an algebraic notation:

Algorithm 2.2 multiplicative Schwarz

$r^{(0)} \Longleftarrow f - Au^{(0)}$
$\delta \Longleftarrow {r^{(0)}}^T r^{(0)}$
k=0
while $\delta > \epsilon$ **do**
  $\omega^{(k+1/2)} \Longleftarrow ({R_1}^T {A_1}^{-1} R_1) r^{(k)}$
  $u^{(k+1/2)} = u^{(k)} + \omega^{(k+1/2)}$
  $r^{(k+1/2)} \Longleftarrow f - Au^{(k+1/2)}$
  $\omega^{(k+1)} \Longleftarrow ({R_2}^T {A_2}^{-1} R_2) r^{(k+1/2)}$
  $u^{(k+1)} \Longleftarrow u^{(k+1/2)} + \omega^{(k+1)}$
  $r^{(k+1)} \Longleftarrow f - Au^{(k+1)}$
  $\delta \Longleftarrow {r^{(k+1)}}^T r^{(k+1)}$
  $k \Longleftarrow k + 1$
end while

Algorithm 2.3 additive Schwarz

$r^{(0)} \Longleftarrow f - Au^{(0)}$
$\delta \Longleftarrow {r^{(0)}}^T r^{(0)}$
k=0
while $\delta > \epsilon$ **do**
  $\omega^{(k+1/2)} \Longleftarrow ({R_1}^T {A_1}^{-1} R_1) r^{(k)}$

  $\omega^{(k+1)} \Longleftarrow \omega^{(k+1/2)} + ({R_2}^T {A_2}^{-1} R_2) r^{(k)}$
  $u^{(k+1)} \Longleftarrow u^{(k)} + \omega^{(k+1)}$
  $r^{(k+1)} \Longleftarrow f - Au^{(k+1)}$
  $\delta \Longleftarrow {r^{(k+1)}}^T r^{(k+1)}$
  $k \Longleftarrow k + 1$
end while

If we define $P_1^{-1} := R_1^T A_1^{-1} R_1$ and $P_2^{-1} := R_2^T A_2^{-1} R_2$ we can eliminate the intermediate step $u^{(k+1/2)}$ in algorithm (2.2):

$$u^{(k+1/2)} = u^{(k)} + P_1^{-1}(f - Au^{(k)}) = (I - P_1^{-1}A)u^{(k)} + P_1^{-1}f \tag{2.10}$$

$$u^{(k+1)} = u^{(k+1/2)} + P_2^{-1}(f - Au^{(k+1/2)}) \tag{2.11}$$

$$\Longrightarrow u^{(k+1)} = (I - P_2^{-1}A)(I - P_1^{-1}A)u^k + [(I - P_2^{-1}A)P_1^{-1} + P_2^{-1}]f \tag{2.12}$$

Using the same notation, the additive Schwarz method simplifies to

$$u^{(k+1)} = u^{(k)} + (P_1^{-1} + P_2^{-1})(f - Au^{(k)}) \tag{2.13}$$

We can now define the additive Schwarz preconditioner used in the above iteration scheme:

$$P_{AS}^{-1} = R_1^T A_1^{-1} R_1 + R_2^T A_2^{-1} R_2 \tag{2.14}$$

As noted before, the additive version is inherently parallel whereas the multiplicative Schwarz method has little potential for parallelization. In the case of many subdomains, however, a strategy of subdomain coloring may be successful, i.e. on subdomains without any overlap the solves can be carried out in parallel as well. This corresponds to the multi-coloring techniques required in order to parallelize the Gauss-Seidel/SOR method. In equations (2.12) and (2.13) it also becomes obvious why the algorithms (2.2) and (2.3) are called multiplicative and additive respectively. The decisive term in (2.12) is a product which prevents parallelism, whereas (2.13) is dominated by a sum. In order to achieve a more efficient algorithm, we now want to use P_{AS}^{-1} as preconditioner for the conjugate gradient method. To do so we must ensure that the transformed system

$$(P_{AS}^{-1}A)u = P_{AS}^{-1}f \tag{2.15}$$

is symmetric and positive definite (spd). This is true if, and only if the preconditioning matrix P_{AS}^{-1} is spd, which is the case since it is the sum of the inverse spd subdomain matrices mapped by the orthogonal projection operators R_i and the domain is fully covered by our partitioning. The multiplicative Schwarz method on the other hand is generally not symmetric, but it might be modified in the same way an SOR method is symmetrized, by applying the preconditioner once from the left and once from the right.

2.4 Multi-domain formulation

The preconditioner P_{AS}^{-1} described so far can be used for the concurrent execution of a PCG solver on two processors. However, it is easy to obtain an even more parallel method since the additive Schwarz method can be extended to an arbitrary number of subdomains without any modifications.

We still consider the index set I of the nodes of the global FE/FD mesh. We now partition the domain into k subsets I_i, $i = 1 \dots k$ such that $\bigcup_i I_i = I$ and $\forall i = 1 \dots k \; \exists \; j \neq i : I_i \cap I_j \neq \emptyset$. That is, the subdomains form a covering of the original domain and any subdomain overlaps with at least one other subdomain. We furthermore require that, considering a division of the k subdomains into two groups, there is always at least a minimal region of overlap between the groups to ensure that information is propagated between the subdomains (see figure 2.3 for an example).

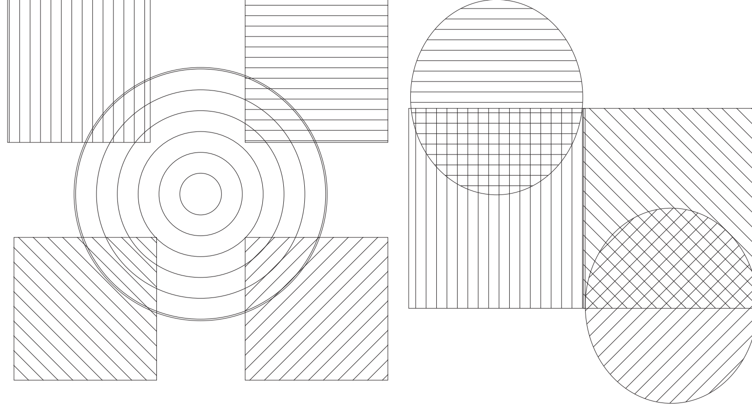

Figure 2.3: left - valid 5-way partitioning. Right - not allowed due to missing overlap

Analogous to the two-domain case we define restriction operators R_i, $i = 1 \ldots k$ such that the subdomain vector v_i containing the entries of the global vector v which are in partition i can be obtained by $v_i = R_i v$ and the k spd subdomain matrices are given by $A_i = R_i A R_i^T$. The overlapping additive Schwarz preconditioner with k subdomains is then defined as

$$P_{AS,k}^{-1} = \sum_{i=1}^{k} (R_i^T A_i^{-1} R_i) \tag{2.16}$$

Although this generalization is straightforward, the convergence behavior will usually deteriorate dramatically with a growing number of subdomains. This is only intuitive since information between subdomains can only progress one domain per iteration step. Many iterations may be required only to get a good approximation of the boundary conditions on the interior boundaries Γ_i, in which steps continually the "wrong" subdomain problems are being solved. In the next section the convergence behavior of the Schwarz methods will be examined more closely and the solution to this problem will be discussed.

2.5 Convergence

We now have a preconditioner $P_{AS,k}^{-1}$ which is easily parallelized on an arbitrary number of processors. The questions that remain to be answered are

- does the algorithm converge at all?
- can we make predictions on the convergence rate?

Since no statements can be given in the general form described up to now, we will discuss the convergence theory for two exemplary PDEs. The first is the two-dimensional stationary Poisson equation with Dirichlet boundary conditions:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y), (x,y) \in \Omega \subset \mathbb{R}^2 \tag{2.17}$$

$$u(x,y) = g(x,y), (x,y) \in \partial\Omega \tag{2.18}$$

which is a stationary elliptic equation. The second problem is the wave equation with initial conditions and, again, a Dirichlet boundary:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x_i^2}, x_i \in \Omega \subset \mathbb{R}^d, d = 2, 3 \tag{2.19}$$

$$u(x, t = 0) = u_0 \tag{2.20}$$

$$u(x_i, t) = u_D(x_i, t), x_i \in \partial\Omega \tag{2.21}$$

This is a time-dependent hyperbolic equation. We assume that both equations have been discretized by a suitable method (i.e finite elements) yielding the system matrices $A_\triangle$ and $A_\sim$ for the Poisson- and the wave equation respectively.

According to Quarteroni and Valli [1], the condition number κ of the preconditioned Poisson problem can be written as follows:

$$\kappa(P_{AS,k}^{-1} A_\triangle) \leq C H^{-2} (\delta H)^{-2} \tag{2.22}$$

On the one hand, the convergence is determined by the factor H^{-2}, where H is the characteristic length scale of the partitions. If we think of the partitioning as a (very) coarse mesh, H would be its mesh size. the more subdomains there are, the smaller will H be and the convergence will deteriorate quadratically with a growing number of partitions. The other term appearing is δH, again inverse to the power of 2. It denotes a measure for the size of the overlapping region of two adjacent subdomains. C is a constant determined only by the coefficients of the system matrix A, so that the convergence is independent of the mesh size h of the global problem. One approach to achieve better convergence would be to increase the amount of overlap. However, this leads to less computational efficiency since the unknowns on the overlap have to be computed by several processors and also have to be communicated through the network (see section 4). If we set $\delta H = \frac{1}{H}$, the entire domain is covered by all subdomains. Convergence will be achieved in one step and the parallel speedup will be 1, which means that we might as well use the sequential solver on a single CPU. The approach to remove this disadvantage is to introduce means of communication other than only the overlap. Since by overlap only neighboring subdomains can communicate their enhanced solution in every step, the information may take in the worst case k iterations to be transported from a certain subdomain to another, and it will be damped by every domain it has to pass. In fact, in every iteration the "wrong" problem is solved since the correct boundary conditions on the interior Dirichlet boundaries Γ_i are not known. In order to allow for global communication between the subdomains, one usually introduces a coarse grid correction. That is, the global problem is solved in every step on a coarser grid which covers Ω completely. The solution on the coarse grid can then be interpolated back onto the fine grid. This mechanism is similar to a multigrid method with two grids, and it removes the dependency on H^{-2} in equation (2.22). The coarse problem can either be solved in parallel or on one processor and then be scattered to the others. Although in most modern applications of DD methods a coarse grid is involved for higher efficiency, in our implementation there is currently no such mechanism. And although it would be possible to add coarse grid correction, we believe that it might not even be necessary for the problem we actually want to solve, which is the wave equation (2.19). Again following the lines of Quarteroni and Valli [1], we will discretize (2.19) using an implicit Newmark time stepping scheme. We are then left to solve a second order elliptic

problem like the Poisson problem discussed before in every time step. However, if finite elements are used for the spatial discretization, the condition number of the system matrix now also depends on the time step size Δt and can be expressed as

$$\kappa(A_{\sim}) = C(1 + (\Delta t)^2 h^{-2}) \tag{2.23}$$

If the time step size is chosen of the same order as the mesh size h, we expect that a coarse grid correction is not required in order to make the domain decomposition method numerically efficient, and also that a relatively small amount of overlap may suffice.

3 The sequential solver

In the following section, I want to present the sequential program which the later parallelization by domain decomposition is based on. I will address the discretization of the equations to be solved, the treatment of boundary conditions and finally the actual solver. The name of the complete software environment is CFS, which stands for coupled field simulator. It is a general framework for the numerical solution of PDEs related to sensor technology. The package is written in C++ and covers everything from discretization techniques up to the solver classes. The part that is of special interest here is the linear algebra system LAS, in which matrix and vector classes as well as linear system solvers and preconditioners are implemented.

3.1 Discretization

The discretization technique commonly used in the CFS package is the finite element method (FEM). Covering the theory of the FEM would exceed the bounds of this thesis, so I will only briefly describe the necessary steps in order to be able to discuss their parallelization. For an introduction to the FEM see, for example, [4].
Starting from a specified mesh, an FE graph is created. For Lagrangian finite elements, the graph nodes represent the positions of the unknowns whereas the connectivity represents the coupling of these grid points in the computational domain. The nodes of the graph are grouped into elements with a specified shape, i.e. triangular elements in the 2D case or tetrahedra/hexahedra in three dimensions. Depending on the equation to be solved and the formulation used (i.e. effective mass or effective stiffness), the system matrix and the right-hand-side are composed from mass- and stiffness matrix with according coefficients. These in turn are assembled by integrating the elements using a suitable numerical integration scheme. For hyperbolic (i.e. time dependent) equations such as the acoustic wave equation mentioned before, we use an implicit Newmark scheme for the time discretization, which leads to the solution of an algebraic system with an updated right-hand-side in every time step.

3.2 Boundary conditions

Since the FEM implicitly deals with Neumann boundary conditions, the only boundary conditions that require special treatment for the implementation of the domain decomposition algorithm are the Dirichlet conditions. In CFS Dirichlet boundary conditions (that is, specified values for the unknown on the boundary $\partial\Omega$ or a part of it) are implemented using the penalty method: An equation is set up for all unknowns, even those for which the values are

known due to the Dirichlet boundary. This means that couplings to other nodes are created which in fact should not be there since the Dirichlet nodes should remain unperturbed. To remove these couplings again, the following procedure is applied to enforce a value D_i on node i:

- a huge value H is determined (i.e. the maximum diagonal entry multiplied by 10^{18})
- HD_i is added to the corresponding entries in the right-hand-side: $\widehat{f}_i = f_i + HD_i$
- H is added to the corresponding diagonal entry of the matrix: $\widehat{A_{ii}} = A_{ii} + H$

This is meant to enforce the Dirichlet value by making the diagonal matrix entry dominate all others in the row. After one preconditioner step, i.e a Jacobi step $u_i = f_i/A_{ii}$, the correct value is present in u:

$$u_i = \frac{\widehat{f}_i}{\widehat{A_{ii}}} = \frac{\underset{\sim}{f_i} + HD_i}{\underset{\sim}{A_{ii}} + H} \stackrel{!}{=} D_i \tag{3.1}$$

The underlined quantities are small enough in relation to the others that they can be ignored.

The residual calculation then theoretically yields a value of 0 on the Dirichlet nodes:

$$\widehat{r}_i = \widehat{f}_i - \widehat{A_{ii}}u_i - \underset{\sim}{\sum_{j \neq i} A_{ij}u_j} = (\underset{\sim}{f_i} + D_iH) - (\underset{\sim}{A_{ii}} + H)D_i \stackrel{!}{=} 0 \tag{3.2}$$

which is vital for the DD algorithm since no correction ω should be computed on $\partial\Omega_i$. As it will turn out, the trick of the penalty method is its actual drawback. Several problems arise from the boundary treatment and some adjustments have to be made to the additive Schwarz algorithm, which will be discussed in section (4.5).

3.3 Linear algebra classes

The implementation of the linear algebra classes in CFS is fairly straightforward: The vectors are stored internally as double-precision arrays and sparse matrices use the well-known compressed row storage (CRS). Specific entries are accessed via C++ access functions. The possibility of using several right-hand-sides simultaneously is given by providing the extra memory in the value arrays of vectors and storing scalar values in pseudo-vectors. It is also possible to simultaneously solve several equations, for which case the according entries are inserted into the matrix. These additional functionalities are not yet fully implemented in the parallel version, but they can be easily added and are not required for our purposes.

3.4 The solver

The solver we want to use as our subdomain solver is a preconditioned conjugate gradient (PCG) method. Several preconditioners are implemented, but to us it is not of great importance which is used since anything that works fine for the global problem should also do for the subdomains. The most powerful preconditioner currently available is an algebraic multigrid method (AMG), and since it has the "quality" of being relatively hard to parallelize it is a good choice for a domain decomposition approach. Again, the conditions forbid going

into the details of the PCG method or AMG. For introductions to CG and AMG resp. see, for example, the works of Shewchuck [5] and Briggs et al. [6]. Since our final algorithm will use a PCG solver not only for the subdomains but also in the global context, I will briefly present the algorithm (see algorithm (4)) and discuss its basic operations since we will have to provide a parallel version of it.

Algorithm 3.1 preconditioned conjugate gradients

$i \Leftarrow 0$
$r \Leftarrow f - Au$
$d \Leftarrow P^{-1}r$
$\delta_{new} \Leftarrow r^T d$
$\delta_0 \Leftarrow \delta_{new}$
while $i < i_{max}$ **and** $\delta_{new} > \epsilon^2 \delta_0$ **do**
  $q \Leftarrow Ad$
  $\alpha \Leftarrow \frac{\delta_{new}}{d^T q}$
  $u \Leftarrow u + \alpha d$
  $r \Leftarrow r - \alpha q$
  $\omega \Leftarrow P^{-1}r$
  $\delta_{old} \Leftarrow \delta_{new}$
  $\delta_{new} \Leftarrow r^T \omega$
  $\beta \Leftarrow \frac{\delta_{new}}{\delta_{old}}$
  $d \Leftarrow \omega + \beta d$
  $i \Leftarrow i + 1$
end while

P^{-1} denotes a preconditioning step, which often does not involve a matrix inversion or matrix-vector product. In addition, the PCG algorithm requires the following non-trivial operations:

- scaling a vector by a factor
- adding two vectors
- computing the scalar (inner) product of two vectors
- multiplying a vector by a matrix

the first two are trivially parallel and do not require communication. The inner product requires only the transmission of one value by a reduction call, so that the only difficulties arise when implementing a parallel matrix-vector product and the preconditioner itself. How this is accomplished will be treated in the next section.

4 Implementation of a parallel framework for domain decomposition

4.1 Parallelization concept

The basic concept which the implementation follows was proposed by Xing Cai and Petter Langtangen [3] who named it the SP approach. SP stands for "simulator parallel" which

means that a sequential solver is parallelized by using it as the subdomain solver for a domain decomposition algorithm. The entire parallel framework is hidden from the user such that no knowledge of parallelization is required. An object oriented approach is suited very well for the implementation: Once the parallel framework has been written the whole task of parallelizing a sequential solver can be done by implementing the according C++ interface. In addition to the easy parallelization another advantage is that the performance of the parallel program is mostly determined by the performance of the subdomain solvers, so that highly tuned sequential solvers can be expected to provide an efficient parallel solver as well. In our implementation of the SP model there are the linear algebra classes for a parallel vector, a parallel matrix, the global CG solver and parallel preconditioners, especially the additive Schwarz preconditioner. In addition, we need two objects to coordinate the parallel program: the first is called GraphDD and is responsible for partitioning the domain. Only one instance is created and passed to all the parallel objects required later on. Any information about the subdomains can be retrieved from the GraphDD object. The second is the Communicator and is responsible for all non-trivial communication issues. Again only one instance is required which is used by all other parallel objects. This has the advantage that several objects of the same type can share communication buffers for the same operations, which saves some memory. An overview of the parallel objects is shown in figure 4.1. Since the GraphDD object is obviously only required for setup purposes, most of the memory it requires can be released before the actual solution phase.

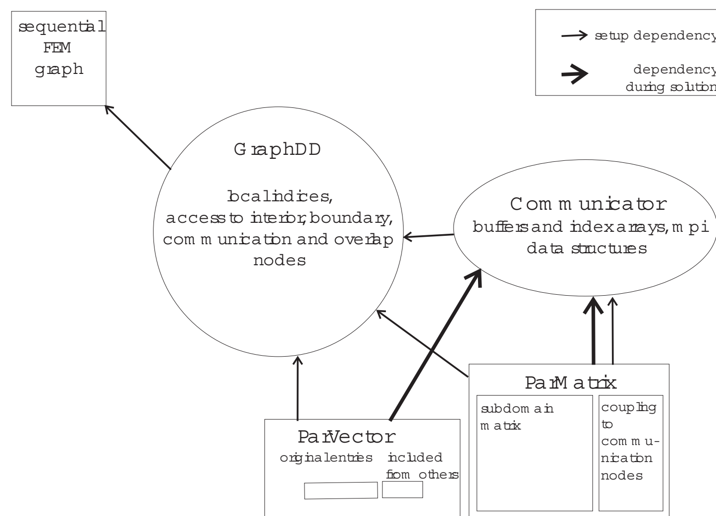

Figure 4.1: objects required for solving a linear system in parallel and their dependencies

The implementation described in this chapter is specially designed for domain decomposition, but it can also efficiently be used for most other parallel preconditioners, provided they are adapted to the data structures. Since we want to use our code on distributed memory systems, we choose MPI for all the communication required. It is both portable and flexible and also relatively easy to use. However, our program will not run truly parallel from the start: some of the data structures are created redundantly on all processors and only the setup and actual solution of the algebraic system are carried out in parallel.

4.2 Decomposing the domain

All tasks related to partitioning are performed by the GraphDD object. The easiest point to perform the partitioning is when the FE graph has been set up. Partitioning in this context means associating with every node a partition number from 0 up to but excluding K, where K is the total number of partitions. We assume that no details on the shape of the domain are known in advance. Two approaches are implemented in our program: the first simply assigns to all nodes in the interval $[i(n/K)\dots(i+1)(n/K)[$ the partition number i and to those in $[n-n\%K\dots n-1]$ the partition number K-1. n is the total number of nodes and i ranges from 0 to K-1. The second possibility is to use the graph partitioning algorithms implemented in Metis. This allows us to create high quality partitions in the sense that they minimize the edge-cut between subdomains so that less communication will be required in order to compute a matrix-vector product for global matrices based on the FE graph. It also ensures that the subdomains are connected regions and - especially in three dimensions - should lead to a higher numerical efficiency since the domains will have a compact shape independent of the numbering scheme of the graph.
After the global mesh and the finite element graph have been created by all the threads (which means that the entire domain is known in every memory location), it is copied to the compressed row storage format required by Metis. Depending on the number of partitions, we use the multilevel recursive bisection algorithm (for up to eight partitions) or the multilevel k-way partitioning, both of which are provided by Metis. For more information on these algorithms please refer to the Metis documentation [7]. In any case we obtain an array of dimension n containing for every node a partition number. I will refer to this array as "part" from now on. It has to be noted that Metis is not capable of creating overlapping partitions. The steps after obtaining the array of partition numbers are first to create data structures which make it easier to access certain nodes of the graph, i.e the boundary nodes between different partitions. Secondly overlap has to be created and finally all requests by other objects (matrices, vectors etc.) have to be dealt with. All of these steps are performed truly parallel, i.e. only for one specific partition per thread.

4.2.1 Data structures for the partitioned graph

In order to create the matrices and vectors corresponding to the local subdomain we need a mapping from global to local indices. This mapping replaces the restriction operators R_i introduced in the theoretical part before. However, unlike the matrix notation implies, neither restriction nor prolongation have to be applied during the solution process since, as we will soon see, the local and the global matrices/vectors share the same memory locations.
The array for the global-to-local mapping has dimension n and takes the following entries on partition i (with n_i nodes) depending on the node N_j:

$$\text{LocalIndex[j]} = \begin{cases} 1 \le k \le n_i: & N_j \in \Omega_i \cup \partial\Omega_i \setminus \Gamma_i \\ -k \quad : & N_j \in \Gamma_i \\ n+1 \quad : & N_j \notin \Omega_i \cup \partial\Omega_i,\ \exists N_l \in \Gamma_i : [N_j, N_l] \neq 0 \\ 0 \quad : & \text{else} \end{cases} \tag{4.1}$$

$[N_j, N_l]$ denotes an edge between the nodes j and l, n is again the total number of nodes of the graph. With the part array every process can now quickly find out which partition a node belongs to. LocalIndex allows to classify a given node as follows: If it is nonzero, it is of

interest to the local partition in some way. If it is equal to n+1 it is on an adjacent partition and its value is required for the global matrix-vector product. If it has a negative sign it is on the interior boundary of the local subdomain and has to be treated as a Dirichlet value in the domain decomposition process. Otherwise, it is an interior node and can be treated normally, except that it might still be shared due to overlap (see later). The next step we have to take is to allow access to all nodes of the above classes. For instance, in order to set up the subdomain matrix, we will have to access all the interior nodes and the boundary nodes since they make up all local finite elements. The communication setup for a matrix-vector multiplication (MVM) demands access to the nodes with local index n+1 and so on. Since the process of extending the overlap will require a certain flexibility regarding memory allocation and access, we use the map class from the standard template library (STL) to provide these access operations. The global indices of all nodes with nonzero local index are therefore stored in the according map sorted by their local indices. There is one map for interior nodes, one for boundary nodes and one for communication nodes. In the latter the entries are sorted by global indexing since no local indices are available. Each node is present in at most one of these three maps. In addition, it might be contained in one or more overlap maps, which are created for every subdomain and contain all nodes shared by the local and the corresponding other partition. Additionally, a temporary list structure ("AllNodes") is created, which stores for every process all the nodes which are currently inside its domain. This is necessary to keep track of the partition overlap during the process of extending the subdomains without altering the part or local index arrays. Finally during setup the following quantities are determined and stored: the total number of edges in the interior of the subdomain, which is equal to the number of entries in the subdomain matrix; the number of edges "cut through" by the partition boundary, equivalent to the number of entries outside the subdomain matrix but within the rows of the local partition; and finally the amount of nodes in each map container. Figure 4.2 shows an overview of the GraphDD object.

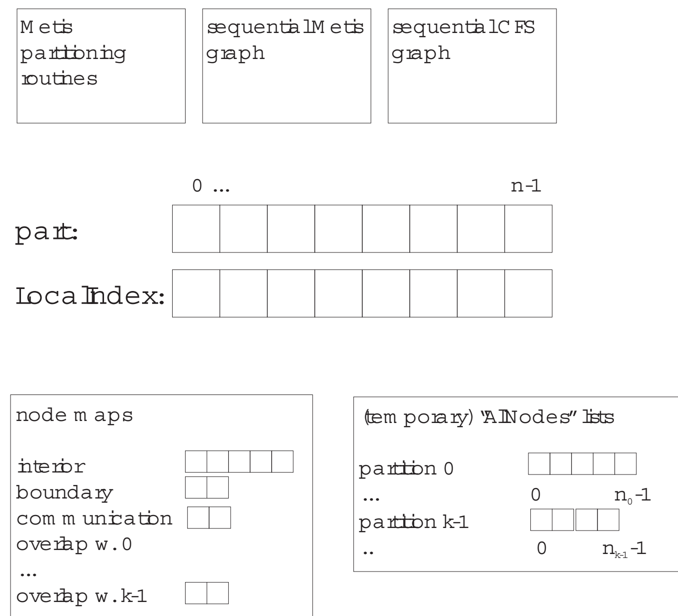

Figure 4.2: overview of the GraphDD object.

An example of a square mesh partitioned into four domains using Metis and the local indexing is shown in figure 4.3. The edges between the nodes (marked by their global/local indices) have been omitted for a better overview, but they correspond to a nine-point stencil, which leads to the diagonal boundaries created by Metis.

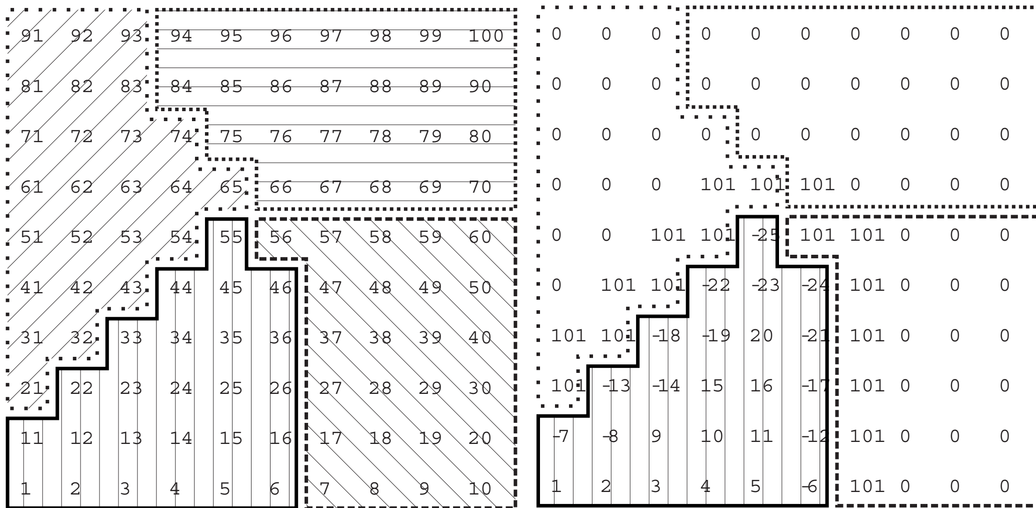

Figure 4.3: Left: example partitioning. Right: local indexing for partition 0

4.2.2 Creating overlap

The partitioning process up to now only assigned exactly one partition number to every node. Since the subdomain solves will not modify the local boundaries $\partial\Omega_i$ the approximation there will never be improved by the preconditioner and the convergence will be relatively slow. Such a preconditioner without any overlap is called block Jacobi preconditioner. However, as soon as at least one layer of nodes beyond the original member nodes is added to each subdomain, a correction will be calculated for all nodes by at least one subdomain solver. We therefore require a subroutine that includes all nodes just outside the subdomain into it. By calling this function several times it is possible to create larger regions of overlap, i.e after m calls all points with at most m nodes distance from the original subdomain will be included. Since this is done by all partitions, the layer of overlap between two adjacent subdomains will have a thickness of 2m nodes. In figure 4.4 the example of the 10×10 square is shown again after creating one layer of overlap.

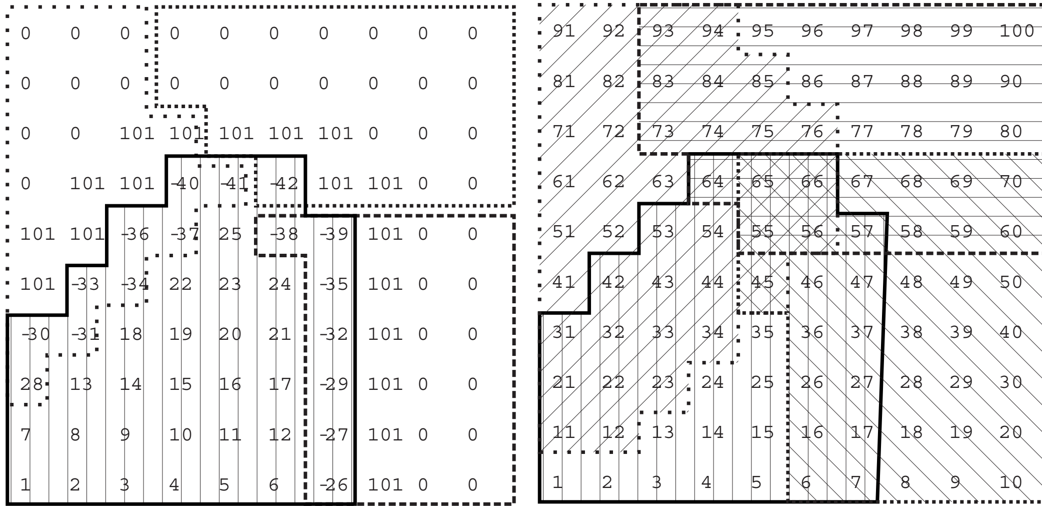

Figure 4.4: left: extension of overlap for partition 0; right: final partitioning with one layer of overlap

in order to extend the overlap by one layer, the following steps are performed:

- all former boundary nodes become interior nodes. They are moved between the according STL maps and their local indices are made positive.
- all former communication nodes become either boundary nodes or interior nodes of the extended subdomain. Local indices are given to the new members starting from n_i+1, and their global indices are stored in an temporary array.
- using the adjacency structure of the graph, the new communication nodes are determined and inserted in the according map.
- finally, the global indices of newly inserted nodes are sent to all CPUs and the information is inserted into the "AllNodes" lists and the overlap maps.

In the case shown in figures 4.3 and 4.4, global node 21 is an example of a communication node which immediately becomes an interior node when included, not a boundary node. During the above process, some more information has to be kept track of: To create the parallel matrix, we need the number of interior edges and the local edge-cut, which will change every time a node is included. The communication of newly added nodes is performed via global indices since they are globally unique can easily be converted to local indices by the GraphDD object. The new global partition structure is created in the "AllNodes" lists so that after the extension process each thread has the full partitioning information of the global domain.

4.2.3 Performance of the partitioning class

Our implementation as described so far does not earn the attributes efficient or elegant. The global graph is created on every CPU, then copied into another format for Metis. All the nodes are stored once in the maps and in addition to that in the list structure. However, one has to keep in mind that we are still in the setup phase long before the actual solution of a linear system. The object GraphDD is required only to create the Communicator, the matrices

and vectors. After everything has been set up, most of the data structures in GraphDD are deleted and the memory is released once more. It is also worth noting that for time dependent problems after the short setup phase, there follows a sequence of many - possibly thousands of - linear system solves. During this time consuming process no significant access to GraphDD will be required.

4.3 Linear algebra classes

At the heart of any linear algebra package there are usually the classes for matrices and vectors. For our purposes we require classes for a parallel real-valued scalar matrix in the compressed row format and for a real-valued vector. Both classes are designed to suit the requirements of DD algorithms, but if no overlap is created it is possible to write other preconditioners for their format as well. Before creating a matrix or vector, both an object of type GraphDD and of type Communicator have to be created and fully set up, that is the overlap can not be extended after creating any parallel matrix or vector based on the decomposition. I will anyway describe the linear algebra classes before the Communicator in order to make clear which requirements have to be met by the Communicator.

4.3.1 Parallel vector class

The implementation of a parallel vector for the PCG algorithm has to provide the following functionality:

- addition of two vectors
- scaling by a factor
- inner product with another vector

For our purpose - the overlapping Schwarz preconditioner - we also need the possibility to obtain the vector corresponding to the subdomain and to exchange and sum up those values shared with other processes due to overlap. Finally, we want to gather the solution on a single node in order to print it to a file or use it otherwise.
The class RParVector provides this functionality. When the constructor is called, pointers to both a GraphDD and a Communicator object are passed to the newly created vector, which are shared by all parallel vectors and matrices. From the GraphDD we obtain the local partition size including the overlap and allocate the according amount of memory. It is worth noting that due to the extension procedure described in the last section those nodes which where originally not in the local partition are automatically placed at the end of the value array. When multiplying a vector by a scalar, no communication is required and all values - even those on the overlap are scaled. Thus, the values on the overlap will be consistent with the other partitions. The same holds for the addition of two vectors. When performing an inner product $c = a_i b_i$, every process locally sums up those values which were originally in its partition. Then a call to MPI_Allreduce computes the global inner product. Since the values required are stored sequentially in the first $n_{original}$ positions, no indirect memory access is required.
In order to obtain the local vector for the subdomain, we can simply use the value array and embed it in a sequential vector. Thus the global parallel vector will be affected by all

operations on the subdomain vector without further effort. The last abilities of the vector class are to sum the values from all processes on the overlapping region and to collect the parts from all CPUs. Since these are pure communication tasks we leave them to the Communicator object.

4.3.2 Parallel matrix class

The structure of our parallel matrix class is again determined by the algorithm we want to use it for. In the DD preconditioner we need the matrices corresponding to the subdomains, whereas the global CG requires the complete system matrix. In order to save memory, our design is as follows: Each CPU stores those matrix entries corresponding to the interior subdomain edges in a sequential matrix, the subdomain matrix. All entries corresponding to edges cut through by the overlapping partitioning are stored in a special CRS format - special in the sense that we store positions in the communication buffer instead of column indices of the matrix. Accessing the subdomain matrix is again only a pointer operation and no further requirements have to be met for the preconditioner part - neither extension nor restriction are required.
To create a parallel matrix, two steps are required: first, the row pointer and index array are set to the correct values. This is achieved by first obtaining for every row of the global matrix the local index from the GraphDD object. If it is not equal to zero or to the global size + 1, i.e. if it is in the subdomain or on its boundary, we proceed as follows: interior nodes (i.e. those with positive local index) are passed on to the subdomain matrix without further treatment. For boundary nodes we determine all edges inside the subdomain and pass only those to the subdomain matrix. All edges to communication nodes are placed in the additional array for buffer positions, where the buffer position is obtained directly from the Communicator. The row pointer is set according to the number of edges between the node and communication nodes (i.e. the edge-cut for the node). We then have to assemble the matrix entries from the finite element graph. The numerical integration has to be performed by every process only for those elements which have at least one node in the corresponding partition. The values computed by the integrator are again either passed to the subdomain matrix - if the edge is inside the subdomain - or added to the corresponding value in the array for additional entries if the edge is from a boundary node to a communication node.
Given this data structure, we now only have to be able to compute a global matrix-vector multiplication (MVM) for the overall CG solver. As we will soon see, our design is suited very well for an MVM based on non-blocking MPI calls. Using the Einstein notation, the sequential MVM can be written as follows:

$$y_i = A_{ij} x_j \tag{4.2}$$

The writing access to y requires only values already present in the memory of each CPU. Furthermore the entries of the subdomain matrix are only multiplied by entries of x which are in the local memory as well. Thus, the entries of x which are not in the local subdomain but required for the MVM can be communicated while performing an MVM with the subdomain matrix and the local parts of x and y. The parallel matrix-vector product is visualized for a specific partition i in figure 4.5.

Before starting the subdomain multiplication, the Communicator starts the sending and receiving of those - and only those - entries which are required by the single partitions. After

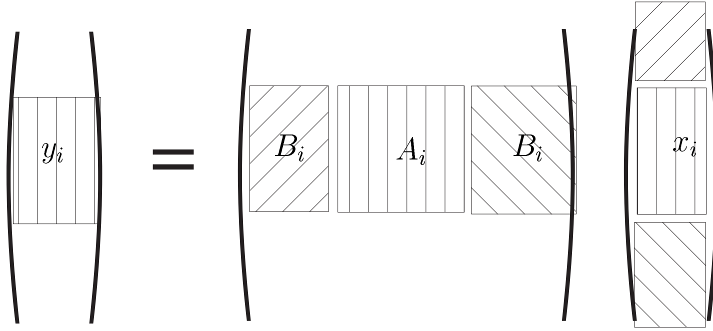

Figure 4.5: parallel MVM for one thread

finishing the local computation we have to wait until all the entries have arrived. In order to avoid indirect memory access, we do not store the column index for the matrix entries outside the diagonal block. Instead we store the buffer positions of the corresponding y entries received by the Communicator. The total process of computing y=Ax in a C++ like notation is shown in algorithm 4.1. "val" is the pointer to the values of the subdomain vectors or the matrix entries outside the diagonal block respectively. In the "pos" array of the matrix the buffer positions are stored, and "row" contains the starting indices for all rows in the "pos" and "val" arrays.

Algorithm 4.1 parallel matrix-vector-multiplication

```
Communicator->StartMVM(x->val);
SubdomainMatrix->Mult(x->val, y->val);
double *buf=Communicator->ReceiveMVM();
for (i=0; i<localsize; i++) do
  for (j=row[i]; j<row[i+1]; j++) do
    y->val[i]+=A->val[j]*buf[A->pos[j]]
  end for
end for
```

From a performance point of view our design has three advantages: Firstly, the non-blocking communication allows to make use of the CPUs floating point units even during communication. Secondly, the storage of buffer positions instead of column indices removes one level of indirection. Finally, perhaps the most important point is that even though we are using a parallel matrix class, the performance is still dominated by the MVM of the sub-domain matrix. As soon as a faster sequential matrix class is available it can be used as the subdomain matrix class and the parallel performance will benefit as well since usually only few matrix entries will be outside the diagonal block.

4.4 Communication

We have up to now described the partitioning of the domain and the classes for vectors and matrices. However, those functions requiring anything more than trivial communication have not been addressed in detail. These are the tasks of the Communicator object: It has to

perform the communication for the MVM, the exchange of vector overlap and the gathering of the solution vector.
The Communicator is created and set up after the GraphDD has been created and the domain is fully partitioned into possibly overlapping subdomains. In order to save memory for buffers and index arrays, only one Communicator is created and passed to all parallel vectors and matrices required later on. This also saves the time required for setup since i.e. all vectors created based on one decomposition have the same overlap structure. During the setup phase, the following data structures are created for the MVM:

- a buffer to store values of a vector for sending
- a buffer to receive values from other processes
- an array to store the indices of values to be send
- the data structures required for the MPI calls, i.e. send offsets, requests etc.

An array for the indices of received values is not required. They are determined once using the GraphDD object and stored by the matrix itself. The buffer positions for entries to be sent are obtained as follows: Every process determines the global indices of entries it requires from others by a call to the GraphDD object. By an MPI_Alltoallv call the information is then send exactly to those other CPUs requiring it for their send indices.
The routine to start the MVM communication uses persistent communication, which means that the MPI interior setup for th communication has to be performed only once, saving some time during later calls. This is possible since the values communicated are always send/received from/to the same locations relative to the pointer passed to the MPI call, regardless of which vector is subject to the communication.

The second task of the Communicator is to exchange the nodal values of a vector on the overlap and sum up the contributions of all processors. Since the GraphDD object has kept track of the overlap with all other processes, the setup is not very complicated anymore. Since values for the same nodes are sent and received, only one index array is needed for overlap communication, but again two buffers.
The final task of our Communicator is to gather the solution vector after a successful global solve. This is done by gathering the original values - without any overlap - and their global indices from all partitions and then reordering the vector entries according to the global indices. Both a function for gathering to a specific CPU for output and for gathering to all CPUs are provided. The latter is required only in transient analysis when the right-hand-side for the next time-step has to be computed from the solution of the last time-step and its derivatives.

4.5 Additive Schwarz preconditioner

We have now developed the parallel framework for our solver. We have created optionally overlapping partitions of our domain and can construct the parallel system matrix and vectors with the required functionality. At this point it is possible to start calculations with a conjugate gradient solver as it is already implemented in the CFS package.
In the following section I will discuss the actual implementation of the overlapping additive

Schwarz preconditioner and some modifications we had to make to the original algorithm.
In the PCG algorithm (3.1), a correction ω is computed by performing a preconditioning step with the global residual as the right-hand-side (RHS). On the other hand, the Schwarz preconditioner requires that Dirichlet values are provided not only on global, but also on interior subdomain boundaries. If we use the Schwarz procedure to calculate a correction rather than a solution to the original system, we would expect that correction to be 0 both on global and on the interior boundaries, which is achieved if the RHS, i.e. the global residual, is 0 on the corresponding nodes. We can achieve this by modifying the global RHS such that it contains not only the global Dirichlet values, but also the current approximation u on interior boundary nodes. However, this implies that the residual has to be calculated in every global CG cycle according to the formula $r = f - Au$ since the internal Dirichlet values continually change.
This form of the residual calculation has two major disadvantages compared to the update formula $r = r - \alpha q$ used in algorithm 3.1. It is computationally a lot more expensive since a matrix-vector product has to be computed. Additionally, it is highly unstable from a numerical point of view if the penalty method is used - a problem which shall be discussed in the next section. From the viewpoint of the implementation other problems arise. As discussed before, the penalty method requires the modification of both the RHS vector and the diagonal entries of the matrix. Since we want to solve the subdomain problems with Dirichlet values on $\partial\Omega_i$ and Γ_i, the subdomain matrix has to be adjusted accordingly. This adjustment however must not be present in the global CG environment since there are no internal Dirichlet values involved there. Because of the memory sharing of global and subdomain matrix, we have to adjust the subdomain matrix before each local solve and reset it to the global entries afterwards. We also cannot use the local part of the global residual as RHS for the subdomain solve since its computation did not consider internal boundaries. We therefore resort to computing the RHS for the subdomain solve as follows: We make a copy of the local part of the global RHS, $\tilde{f}_i = f_{|\Omega_i}$ which contains those entries of f resulting from source terms and the modified global Dirichlet values $H \cdot D_k$ on the global boundary $\partial\Omega_i \setminus \Gamma_i$. We then insert at interior boundary nodes the values of the current approximation u and compute the local residual according to $\tilde{r}_i = \tilde{f}_i - \tilde{A}_i u_i$. The correction is then computed by solving all the subdomain problems $\tilde{A}_i \omega_i = \tilde{f}_i$ and summing up the values of ω computed by all CPUs on the overlap. Note that $\tilde{A}_i$ are the subdomain matrices with the penalty term H added to diagonal elements on Γ_i, which have to be reset to A_i after the local solve. In the global CG we can still use the update formula for the residual calculation.

4.6 Numerical problems with the penalty method

The idea behind the penalty method is to make the diagonal entry of the matrix so dominant that all the other entries of a row have only negligible effects. Since we modify both the RHS and the diagonal entry by a huge value H, the computation of f-Au results in the subtraction of two values both on the edge of the floating point accuracy of the CPU, and very close together. As is well known, the result will never be reliable in this case. Please note that the problem arises only because we have to use the formula $r = f - Au$ for the residual. Otherwise, the penalty method may well be reliable. The solution in our case is that we know the residual at the critical points, i.e. the Dirichlet nodes. It is 0 as soon as the Dirichlet value is placed in the corresponding entry of the vector u. We therefore simply set r=0 both on the global and local Dirichlet boundaries every time we have computed r=f-Au locally.

Combining all the points discussed, we can now write down the preconditioning step as it is implemented. Capital letters shall denote global (parallel) quantities, small letters the corresponding parts of subdomain i. Quantities marked with a ˆ have been modified due to the penalty method for Dirichlet nodes.

Algorithm 4.2 a Schwarz preconditioning step

include Dirichlet structure in subdomain matrix

$\widehat{\widehat{a_{jj}}} \Leftarrow \widehat{A_{jj}} + H$ if $j \in \Gamma_i$

construct a local right-hand-side

$\widehat{\widehat{f_j}} = \begin{cases} \widehat{F_j} & : \text{ node j} \in \Omega_i \cup \partial\Omega_i \setminus \Gamma_i \\ \widehat{F_j} + HU_j & : \text{ node j} \in \Gamma_i \end{cases}$

calculate the local residual

$\widehat{\widehat{r}} = \widehat{\widehat{f}} - \widehat{\widehat{a}}u$

correct the residual

$\widehat{\widehat{r_j}} = 0$ if node j $\in \partial\Omega_i$

perform preconditioning step

$\omega_i = \widehat{\widehat{a}}^{-1}\widehat{\widehat{r}}$

construct the global correction

$\omega = \sum_{k=1}^{K} \omega_k$, where ω_k is extended by 0 outside the subdomain

restore the matrix

5 Numerical results and conclusions

5.1 Test cases

The first test case we want to study is the two-dimensional Laplace equation (i.e. equation 2.17 with f(x,y)=0) on a unit square. For the northern boundary we prescribe a Dirichlet value of 1, for the southern boundary a Dirichlet value of 0. Both on the eastern and western boundaries we assume homogeneous Neumann conditions, so that the solution u(x,y) is a linear function in y and constant in x:

$$u(x, y) = y, \ (x, y) \in [0, 1] \times [0, 1] \tag{5.1}$$

The second equation we want to solve is the wave equation (2.19). Both equations have been described in section 2.5. As noted at that point, we expect faster convergence for the wave equation since the condition number improves with a decreasing time step-size. Following this line of thought the Poisson equation can be regarded as the worst case of a wave equation with an infinitely large time step-size.

Before presenting the results achieved with our program, I would like to make some remarks. All test runs have been performed either on the Intel Cluster of the RRZE - which consists of 76 dual Xeon CPUs with 2.6 GHz each - or on a small network of four Athlon 700 PCs. Since the complete program does not yet fully run on the Intel cluster, I have used the Athlons to study the convergence behavior, leaving the performance measurements to the Xeons. Because of the memory limitations on the Athlons the maximum number of subdomains I use is 16. This is, however, a realistic number since the problem sizes we actually

want to solve are about 500.000 unknowns, for which more than 16 subdomains probably do not make sense anyway. On the Xeon cluster we can only run the program for the stationary Laplace problem, but for performance and scale-up measurements that is sufficient, too. Other problems arising in practice are the high work load on the testing machines and a lack of test cases for the wave equation, especially in three dimensions.

There is also still a problem within the code: our preconditioner is still not quite stable. In rare settings for the Laplace problem (i.e. 316^2 unknowns, six subdomains and one layer of overlap), the computation of δ according to $\delta = r^T P_{AS}^{-1} r_{local}$ yields a negative value at a point, which implies that our preconditioner is no longer positive definite. This phenomenon occurred twice during the development of the program: Before we enforced the residual to be 0 on the interior and global Dirichlet boundaries, and a second time when the boundary values were not correctly updated for time-dependent problems. Although the phenomenon can not fully be explained at this time, it obviously has to do with the interior boundary conditions. Moreover, since it occurs only in the case of six subdomains, we may assume that it depends on the shape of the partitions provided by Metis.
In our experiments with the Poisson problem I at some points had to slightly alter the problem size in order to obtain results anyway.

5.2 Convergence behavior

There are several points we want to study in the following sections, which are the following:

- How does the Schwarz preconditioner improve convergence of the global CG solver for the test cases?
- How does the convergence rate depend on the size of the overlapping region?
- Is the effect of the partitioning algorithm beneficial or irrelevant?
- Is the overall performance of the parallel classes satisfactory?

At first I want to demonstrate the effect of the partitioning algorithm on the convergence. We solve the Laplace problem with approximately 100.000 unknowns. We use a growing number of partitions and only one layer of overlap in between. In the first case we partition the domain according to node numbers as described previously. We then partition the domain using Metis and compare the results, which are shown in figure 5.1.

In this case, the advanced partitioning algorithms obviously have a negative effect on the convergence. What seems contra-intuitive at first glance, however, can be easily explained. Our test problem is only two-dimensional and has a Dirichlet boundary with the value 1 in the north and the value 0 in the south. The numbering is row-wise from the south-west corner to the north-east corner. Therefore, if we use our primitive partitioning scheme, the last domain will include all nonzero Dirichlet values, and can therefore compute a sensible correction from the start. The example shows that the use of Metis is not always recommendable, and if details on the geometry are known in advance it is beneficial to adapt the partitioning algorithm accordingly. On the other hand, as soon as the geometry gets more complicated, the Metis partitioning may very well outperform our primitive scheme. This is revealed as soon as we apply both versions of the solver to a tiny three-dimensional Poisson problem with 231 nodes.

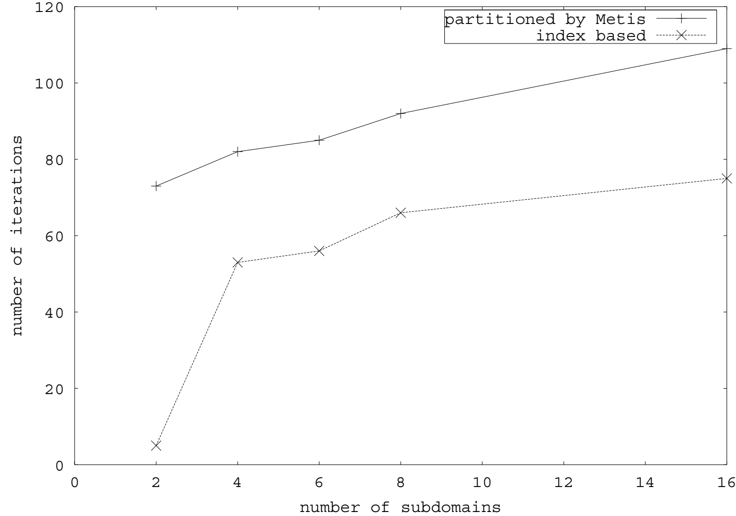

Figure 5.1: effect of the partitioning algorithm for the 2D Laplace problem

We use 8 subdomains, again with only one layer of overlap. This time, the "primitive" version requires 21 global iterations, whereas the PCG with Metis partitioning requires only 14. For the wave equation we will see later that the negative effect of Metis in the 2D case is negligible.

The results for the convergence behavior shown so far are far from satisfactory. The next step to improve convergence will be to create larger regions of overlap in order to improve the communication between subdomains. We shall from now on use the partitioning provided by Metis since it provides more general results (i.e. the convergence rate depends less on the geometry and boundary conditions). Our test problem remains the Poisson problem in two dimensions as described above, again with 100.000 unknowns. We now gradually increase the amount of overlap to up to 5 layers and again monitor the convergence for a growing number of subdomains.

The results are shown in figure 5.2. Although the number of iterations can easily be reduced, the results are still unacceptable since the solution in parallel will probably take longer than if the sequential AMG preconditioner is used directly on the entire domain. But the poor convergence results are exactly what the theory predicts for the Poisson equation. As explained in section 2.5, a coarse grid has to be introduced if the Schwarz methods are to be efficiently used for the stationary Poisson problem. Let us therefore turn to the equation that lends its name to the title of this thesis: the wave equation. Our test setting is as follows: We again use a two-dimensional domain discretized by finite elements. As mentioned before, the time discretization is implicit so that we have to solve an elliptic problem within every time-step. For now we use a relatively small problem size of approximately 26.000 grid points. The domain contains a tiny box emitting a signal which is responsible for the development of an acoustic wave. Our preconditioner uses Metis to partition the domain and creates only one layer of overlap. We then compute 100 time-steps and take the average of the number of global iterations.

As shown in figure 5.3, it turns out that in every time-step no more than two iterations are required for up to six subdomains. For 16 subdomains the maximum number of PCG steps is seven. As can be seen in figure 5.4, the number of global iterations remains stable

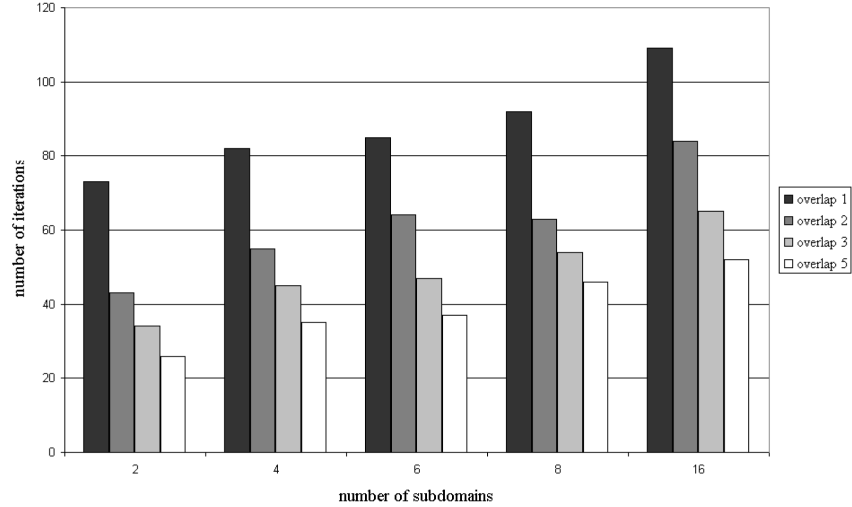

Figure 5.2: convergence and overlap

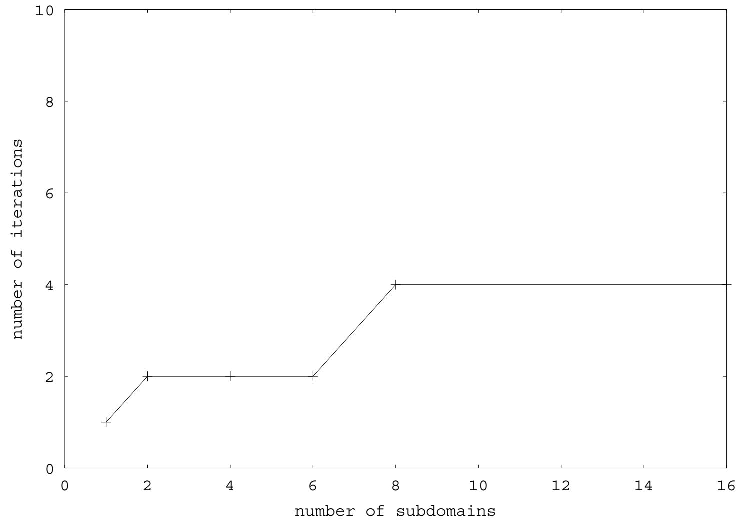

Figure 5.3: convergence for the wave equation with up to 16 subdomains

even if we run our simulation for 500 time-steps.

Our next experiment in order to explore the convergence behavior of domain decomposition methods for the wave equation is to increase the time step-size by decreasing the frequency of the wave. According to the estimate (2.23) from section 2.5 we expect the convergence for a lower frequency to get slower. This is verified in figure 5.5, where we use a time-step of the order of 10^{-5} (corresponding to an audible wavelength) instead of 10^{-8}, which we used up to now (and which corresponds to a frequency in the ultra-sound area). We use up to sixteen subdomains in 100 time-steps. Already for 6 subdomains we observe the expected effect: The convergence rate becomes similar to that achieved for the stationary Laplace problem.

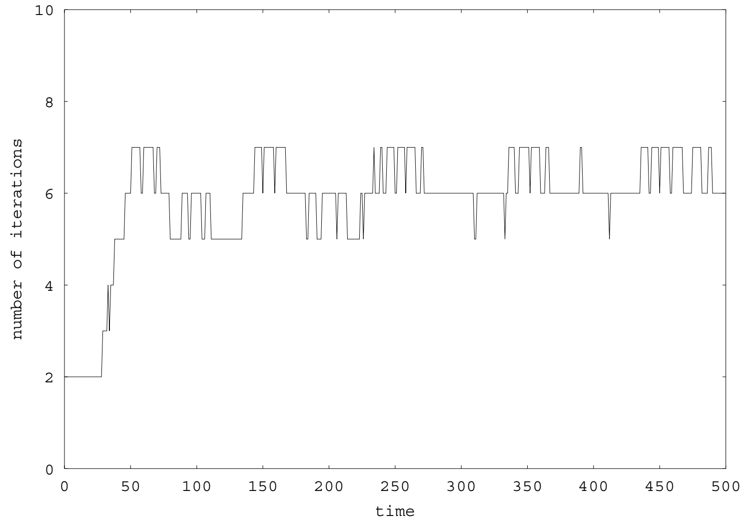

Figure 5.4: convergence for the wave equation, 500 time-steps

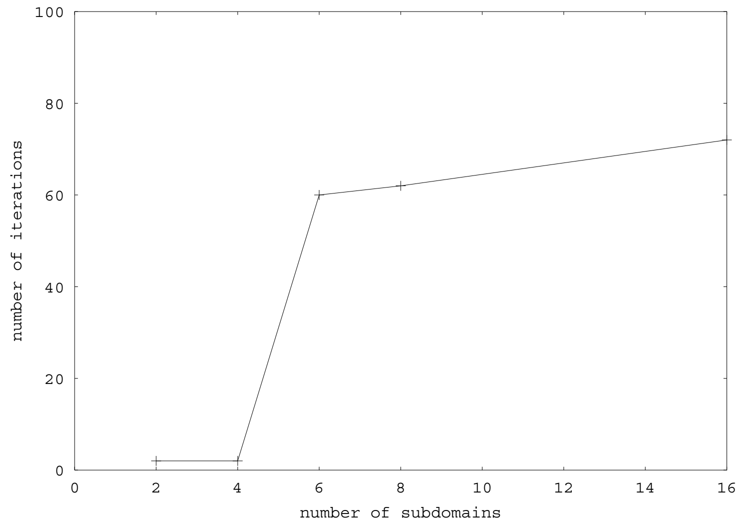

Figure 5.5: convergence for the wave equation with a low frequency

5.3 Performance

The final statistics I want to present are concerned with the parallel performance of our program. The first point we will verify is the remark made earlier that the performance of the Schwarz/CG solver is dominated by the subdomain solves. The following is an excerpt from the profile of a test run for the wave equation. One has to be aware that it has been obtained using the GNU profiler, which only measures the user time. That means that synchronization and communication effects are not taken into account. The argument lists have been removed for reasons of clearness.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 14.67      5.52     5.52      987     0.01     0.01  CoupledField::RScalarMatrix::Solve(...)
 14.46     10.96     5.44     1974     0.00     0.00  CoupledField::RScalarFastGSSmoother::StepForward(...)
 11.46     15.27     4.31     1974     0.00     0.00  CoupledField::RScalarGSSmoother::StepBackward(...)
  8.00     18.28     3.01     1187     0.00     0.00  CoupledField::RScalarMatrix::Mult(...)
  7.89     21.25     2.97     2130     0.00     0.00  CoupledField::Vector<double>::operator=(...)
  5.00     23.13     1.88     1500     0.00     0.00  CoupledField::Vector<double>::Vector[in-charge](...)
  3.91     24.60     1.47     1537     0.00     0.00  CoupledField::Vector<double>::Resize(...)
  3.80     26.03     1.43     1974     0.00     0.00  CoupledField::RScalarTransfer::MulthH(...)
  3.75     27.44     1.41     1974     0.00     0.00  CoupledField::RScalarTransfer::MultHh(...)
  3.00     28.57     1.13      300     0.00     0.01  CoupledField::Vector<double>::operator-()
  2.84     29.64     1.07      800     0.00     0.00  CoupledField::Vector<double>::operator*()
  2.42     30.55     0.91      400     0.00     0.00  CoupledField::Vector<double>::operator+()
  1.62     31.16     0.61        1     0.61     0.61  CoupledField::RScalarMatrix::Factor()
  1.38     31.68     0.52      987     0.00     0.00  CoupledField::RealVector::Add(...)
  1.36     32.19     0.51      100     0.01     0.01  CoupledField::RScalarMatrix::MultAdd(...)
  0.98     32.56     0.37     1174     0.00     0.00  CoupledField::RealVector::Add(...)
```

Obviously, the parallel routines do not appear here, which means that no more than one percent of the user time is spent in any of them - be it a setup routine or a part of the solver. Most of the time is spent in routines related to the algebraic multigrid preconditioner on the subdomain, i.e the coarse grid solve and the smoothing steps. However, as I mentioned before, the time spent waiting for communication and synchronization might very well change this behavior to the worse. We therefore need some measurements of the real time required to solve our test problems.

At first, I will demonstrate the performance of the linear algebra classes. In order to do so, we solve the Laplace problem with 500.000 unknowns. But this time we do not use the Schwarz preconditioner. Instead we use a Jacobi preconditioner (i.e. a simple diagonal scaling), so that 706 iterations are required for the solve and the performance is dominated by the MVM and the vector operations. In order to diminish the effect of thread startup, we solve the system 10 times, the architecture used is the Intel Cluster. Figure 5.6 shows the real time required (including the setup phase), figure 5.7 shows the performance scale-up in MFlop/s, i.e. 10^6 floating point operations per second. Though a substantial speedup can be observed, the difference to the ideal case of a linear scale-up becomes quite large as the number of CPUs is increased. In order to achieve a better speedup behavior, we possibly would have to increase the problem size.

Finally, let us now turn again to domain decomposition for the wave equation. We shall run a simulation with the same settings as before, the high frequency and approximately 230.000 unknowns for 100 time-steps. The testing environment are the four Athlon PCs, and we now directly compare the parallel Schwarz preconditioner with the sequential AMG.

As can be seen in figure 5.8, the real time required for the solution rather rises than going down. Though we might blame the behavior on the workload of the machines and the slow interconnection, this would be too easy since for the Jacobi preconditioner a speedup can very well be observed. There is a more theoretical approach, which I briefly want to present. According to [6], the performance of an AMG solver is proportional to the *operator complexity* σ, defined as the fraction of the nonzero entries in all coarse grid operators and those in the fine grid operator. In the sequential case, the operator complexity for our setting is $\sigma_s = 1.065 \approx 1$. As the factor of proportion we assume the number of nonzero entries in the matrix (nz). The CG algorithm requires essentially one MVM, yielding a total complexity of

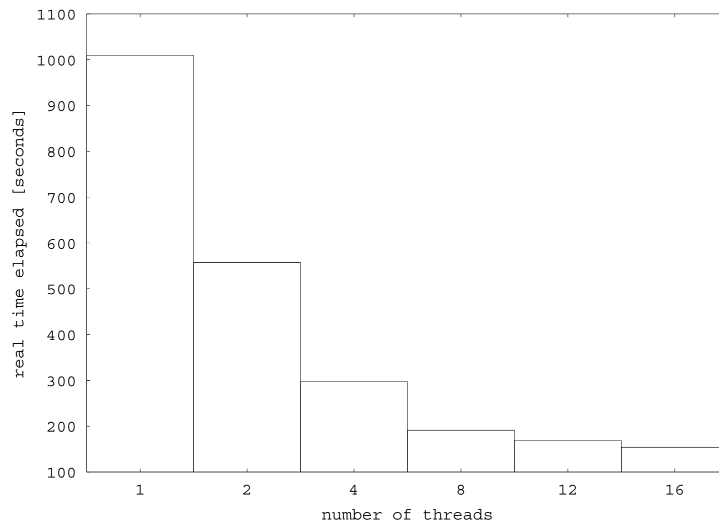

Figure 5.6: real time required for 10 linear system solves with the Jacobi preconditioner

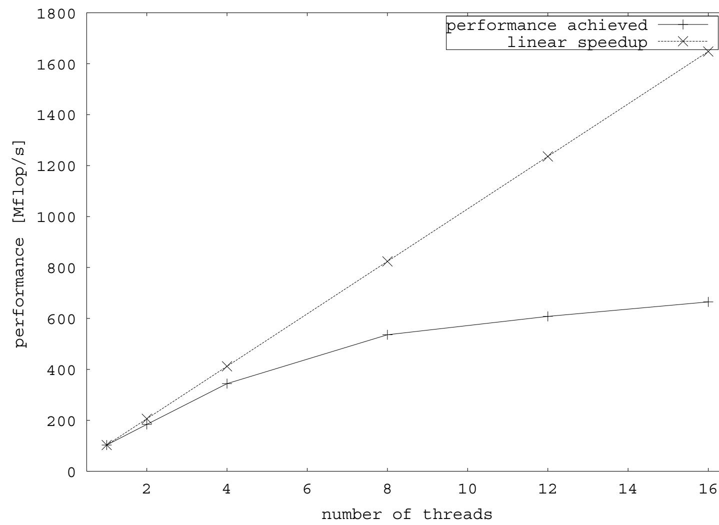

Figure 5.7: performance of the parallel CG solver with Jacobi preconditioning

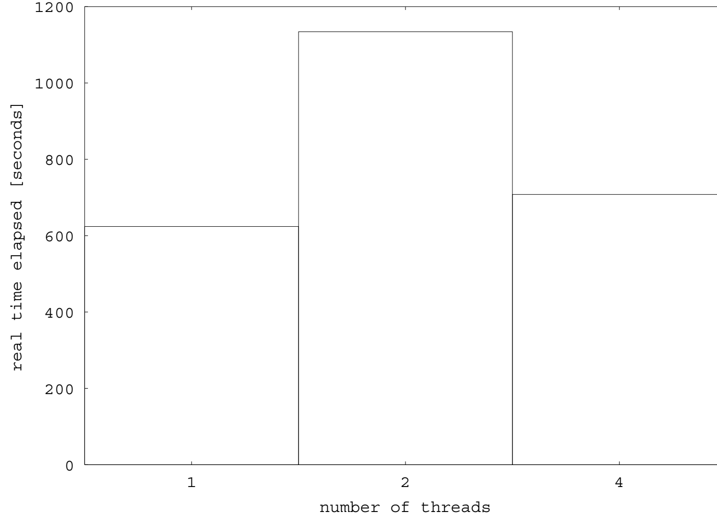

Figure 5.8: real time required to solve the wave equation using the Schwarz preconditioner

2nz. Since we require an average of five iterations per time-step and our system matrix has about two million entries, $5 \times 4 \times 10^6$ should be a figure representative for the complexity of the solution phase in the sequential case.

When using the Schwarz preconditioner for two subdomains, we have to use the maximum number of subdomain iterations (6) multiplied by the number of global iterations (2) as a worst case assumption for each time-step. This makes sense since the threads are synchronized after the parallel subdomain solves. We thus obtain a number of 12 subdomain iterations per time-step. Again, we obtain an operator complexity which is approximately 1, but this time on the subdomains with approximately 1.000.000 nonzero entries each. Together with the local CG solvers, we obtain a characteristic value of $12 \times 2 \times 10^6$ for each subdomain. The global CG is dominated by the MVM, but this time with only half of the global nonzero entries because of the parallelism and with only the two global iterations. In addition, we need one more local MVM for the residual calculation, which leads to a total of 3×10^6 operations. In total, we have to perform an estimated 27 million operations *per subdomain*, which is a lot more than the 20 million in the sequential case. Rather then the somewhat over-protective synchronization of my code and the hardware used, this fully explains the poor performance achieved. For the case of four subdomains the above estimate yields a figure of 15 million, which means that on each CPU almost as much work has to be done as in the sequential case. A similar calculation for the case of 16 subdomains however - under the assumption that the number of global and local iterations remains the same - yields a value of 3.375 million, which is in fact a lot lower than in the sequential case. Unfortunately, I do not have the possibility to run the program for this setting, but I expect it to outperform the sequential solver by far.

5.4 Concluding remarks

In my thesis I have presented both theoretical and practical aspects of the overlapping additive Schwarz preconditioner. The step from theory to implementation has proved to be a lot more difficult and time consuming than I had expected. The Schwarz method in matrix notation can be written down in a few lines, but that does not consider the treatment of boundary

conditions. Especially the treatment of the interior boundaries with the penalty method has continually caused problems during the implementation phase, and may still continue to do so. The results presented in the last chapter, however, support the predictions of the theory: The algorithm proves to be numerically very efficient for the wave equation at a high frequency, and an improvement to the convergence when increasing the overlap size has been noticed. For a small number of subdomains the sequential solver is still the better choice, but as shown in the last section, we might expect this to change with a growing number of subdomains. The parallel performance of our implementation has been shown to be very good provided the problem size is large enough for the number of threads to pay off. Some of the points which might still be improved in future work include the use of a coarse grid correction, fine-tuning of the code (for example, some of the synchronization included is possibly obsolete) and a lot more testing, i.e. for a larger number of subdomains and the three-dimensional wave equation. In order to achieve an efficient algorithm I expect a large number of subdomains (maybe 32) and a direct solver on the subdomains to be a good choice since the costs of the subdomain solves will be drastically reduced.

List of Figures

2.1 overlapping two-way partitioning of a domain 7
2.2 two-way partitioned linear system with overlap 8
2.3 left - valid 5-way partitioning. Right - not allowed due to missing overlap . . 11
4.1 objects required for solving a linear system in parallel and their dependencies . 16
4.2 overview of the GraphDD object. 18
4.3 Left: example partitioning. Right: local indexing for partition 0 19
4.4 left: extension of overlap for partition 0; right: final partitioning with one layer of overlap 20
4.5 parallel MVM for one thread 23
5.1 effect of the partitioning algorithm for the 2D Laplace problem 28
5.2 convergence and overlap 29
5.3 convergence for the wave equation with up to 16 subdomains 29
5.4 convergence for the wave equation, 500 time-steps 30
5.5 convergence for the wave equation with a low frequency 30
5.6 real time required for 10 linear system solves with the Jacobi preconditioner . 32
5.7 performance of the parallel CG solver with Jacobi preconditioning 32
5.8 real time required to solve the wave equation using the Schwarz preconditioner 33

List of Algorithms

2.1 continuous multiplicative Schwarz 7
2.2 multiplicative Schwarz 9
2.3 additive Schwarz 9
3.1 preconditioned conjugate gradients 15
4.1 parallel matrix-vector-multiplication 23
4.2 a Schwarz preconditioning step 26

References

[1] A. Quarteroni, A. Valli: **Domain Decomposition Methods for Partial Differential Equations**,
Oxford Science Publications, Clarendon Press, Oxford 1999

[2] T. F. Chan, T. P. Mathew: Domain decomposition algorithms, acta numerica (1994), pp. 61-143

[3] H. P. Langtangen, X. Cai: A Software Framework for Easy Parallelization of PDE Solvers Department of Informatics, University of Oslo, P.O. Box 1080, Blindern, N-0316 Oslo, Norway, hpl,xingca@ifi.uio.no
obtained from `http://heim.ifi.uio.no/ hpl/MI-www_docs/reports/pcfd2000/`

[4] M. Kaltenbacher: Numerical Simulation of Mechatronic Sensors and Actuators

[5] J. R. Shewchuck: An Introduction to the Conjugate Gradient Method without the agonizing Pain, school of computer science, Carnegie Mellon University, Pittsburgh 1994

[6] W. L. Briggs, Van Emden Henson, S. F. McCormick: A Multigrid Tutorial, second edition, Society for Industrial and Applied Mathematics (SIAM) 2000

[7] G. Karypis, V. Kumar: METIS* A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, 1998