Lecture 8

Domain Decomposition

Domain decomposition is a term used by at least two different communities. Literally, the words indicate the partitioning of a region. As we will see in Chapter **??** of this book, an important computational geometry problem is to find good ways to partition a region. This is not what we will discuss here.

In scientific computing, domain decomposition refers to the technique of solving partial differential equations using subroutines that solve problems on subdomains. Originally, a domain was a contiguous region in space, but the idea has generalized to include any useful subset of the discretization points. Because of this generalization, the distinction between domain decomposition and multigrid has become increasingly blurred.

Domain decomposition is an idea that is already useful on serial computers, but it takes on a greater importance when one considers a parallel machine with, say, a handful of very powerful processors. In this context, domain decomposition is a parallel divide-and-conquer approach to solving the PDE.

To guide the reader, we quickly summarize the choice space that arises in the domain decomposition literature. As usual a domain decomposition problem starts as a continuous problem on a region and is disretized into a finite problem on a discrete domain.

We will take as our model problem the solution of the elliptic equation $\nabla^2 u = f$, where on a region Ω which is the union of at least subdomains Ω_1 and Ω_2. ∇^2 is the Laplacian operator, defined by $\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$. Domain decomposition ideas tend to be best developed for elliptic problems, but may be applied in more general settings.

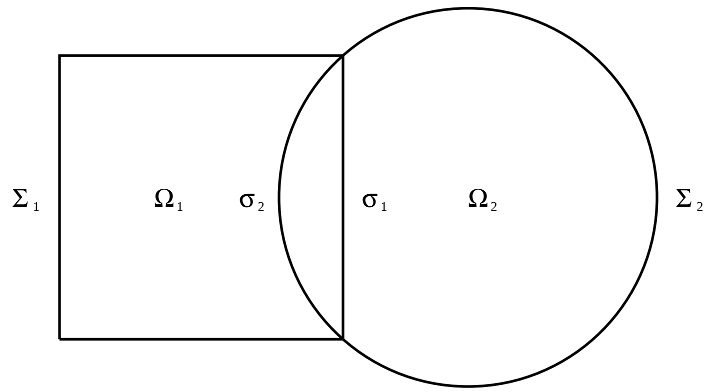


Figure 8.1: Example domain of circle and square with overlap

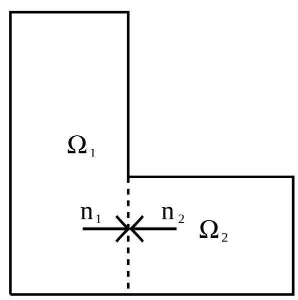


Figure 8.2: Domain divided into two subdomains without overlap

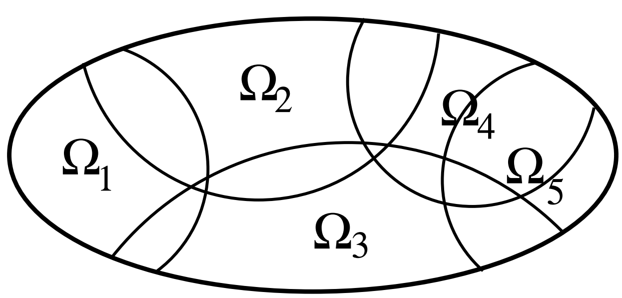


Figure 8.3: Example domain with many overlapping regions

DOMAIN DECOMPOSITION OUTLINE

1. Geometric Issues
   - Overlapping or non-overlapping regions
   - Geometric Discretization
     - Finite Difference or Finite Element
     - Matching or non-matching grids
2. Algorithmic Issues
   - Algebraic Discretization
     - Schwarz Approaches: Additive vs. Multiplicative
     - Substructuring Approaches
   - Accelerants
     - Domain Decomposition as a Preconditioner
     - Course (Hierarchical/Multilevel) Domains
3. Theoretical Considerations

8.1 Geometric Issues

The geometric issues in domain decomposition are 1) how are the domains decomposed into subregions, and 2) how is the region discretized using some form of grid or irregular mesh. We consider these issues in turn.

8.1.1 Overlapping vs. Non-overlapping regions

So as to emphasize the issue of overlap vs. non-overlap, we can simplify all the other issues by assuming that we are solving the continuous problem (no discretization) exactly on each domain (no choice of algorithm). The reader may be surprised to learn that domain decomposition methods divide neatly into either being overlapping or nonoverlapping methods. Though one can find much in common between these two methods, they are really rather different in flavor. When there is overlap, the methods are sometimes known as Schwarz methods, while when there is no overlap, the methods are sometimes known as substructuring. (Historically, the former name was used in the continuous case, and the latter in the discrete case, but this historical distinction has been, and even should be, blurred.)

We begin with the overlapping region illustrated in Figure 8.1. Schwarz in 1870 devised an obvious alternating procedure for solving Poisson's equation $\nabla^2 u = f$:

1. Start with any guess for u_2 on σ_1.
2. Solve $\nabla^2 u_1 = f$ on Ω_1 by taking $u_1 = u_2$ on σ_1. (i.e. solve in the square using boundary data from the interior of the circle)
3. Solve $\nabla^2 u_2 = f$ on Ω_2 by taking $u_2 = u_1$ on σ_2 (i.e. solve in the circle using boundary data from the interior of the square)
4. Goto 2 and repeat until convergence is reached

The procedure above is illustrated in Figure 8.4.

One of the characteristics of elliptic PDE's is that the solution at every point depends on global conditions. The information transfer between the regions clearly occurs in the overlap region.

If we "choke" the transfer of information by considering the limit as the overlap area tends to 0, we find ourselves in a situation typified by Figure 8.2. The basic Schwarz procedure no longer works. Do you see why? No matter what the choice of data on the interface, it would not be updated. The result would be that the solution would not be differentiable along the interface. An example is given in Figure 8.5.

One approach to solving the non-overlapped problem is to concentrate directly on the domain of intersection. Let g be a current guess for the solution on the interface. We can then solve $\nabla^2 u = f$ on Ω_1 and Ω_2 independently using the value of g as Dirichlet conditions on the interface. We can define the map

$$T : g \rightarrow \frac{\partial g}{\partial n_1} + \frac{\partial g}{\partial n_2}.$$

This is an affine map from functions on the interface to functions on the interface defined by taking a function to the jump in the derivative. The operator T is known as the Steklov-Poincaré operator.

Suppose we can find the exact solution to $Tg = 0$. We would then have successfully decoupled the problem so that it may be solved independently into the two domains Ω_1 and Ω_2. This is a "textbook" illustration of the divide and conquer method, in that solving $Tg = 0$ constitutes the "divide."

8.1.2 Geometric Discretization

In the previous section we contented ourselves with formulating the problem on a continuous domain, and asserted the existence of solutions either to the subdomain problems in the Schwarz case, or the Stekhlov-Poincaré operator in the continuous case.

Of course on a real computer, a discretization of the domain and a corresponding discretization of the equation is needed. The result is a linear system of equations.

Finite Differences or Finite Elements

Finite differences is actually a special case of finite elements, and all the ideas in domain decomposition work in the most general context of finite elements. In finite differences, one typically imagines a square mesh. The prototypical example is the five point stencil for the Laplacian in two dimensions. Using this stencil, the continuous equation $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$ is transformed to a linear system of equations of the form:

$$-\frac{1}{h^2}(-u_i{}^E + 2u_i - u_i{}^W) - \frac{1}{h^2}(-u_i{}^N + 2u_i - u_i{}^S) = 4f_i$$

where for each u_i, $u_i{}^E$ is the element to the right of u_i, $u_i{}^W$ is to the left of u_i, $u_i{}^N$ is above u_i, and $u_i{}^S$ is below u_i. An analog computer to solve this problem would consist of a grid of one ohm resistors. In finite elements, the protypical example is a triangulation of the region, and the appropriate formulation of the PDE on these elements.

Matching vs. Non-matching grids

When solving problems as in our square-circle example of Figure 8.1, it is necessary to discretize the interior of the regions with either a finite difference style grid or a finite element style mesh. The square may be nicely discretized by covering it with Cartesian graph-paper, while the circle may

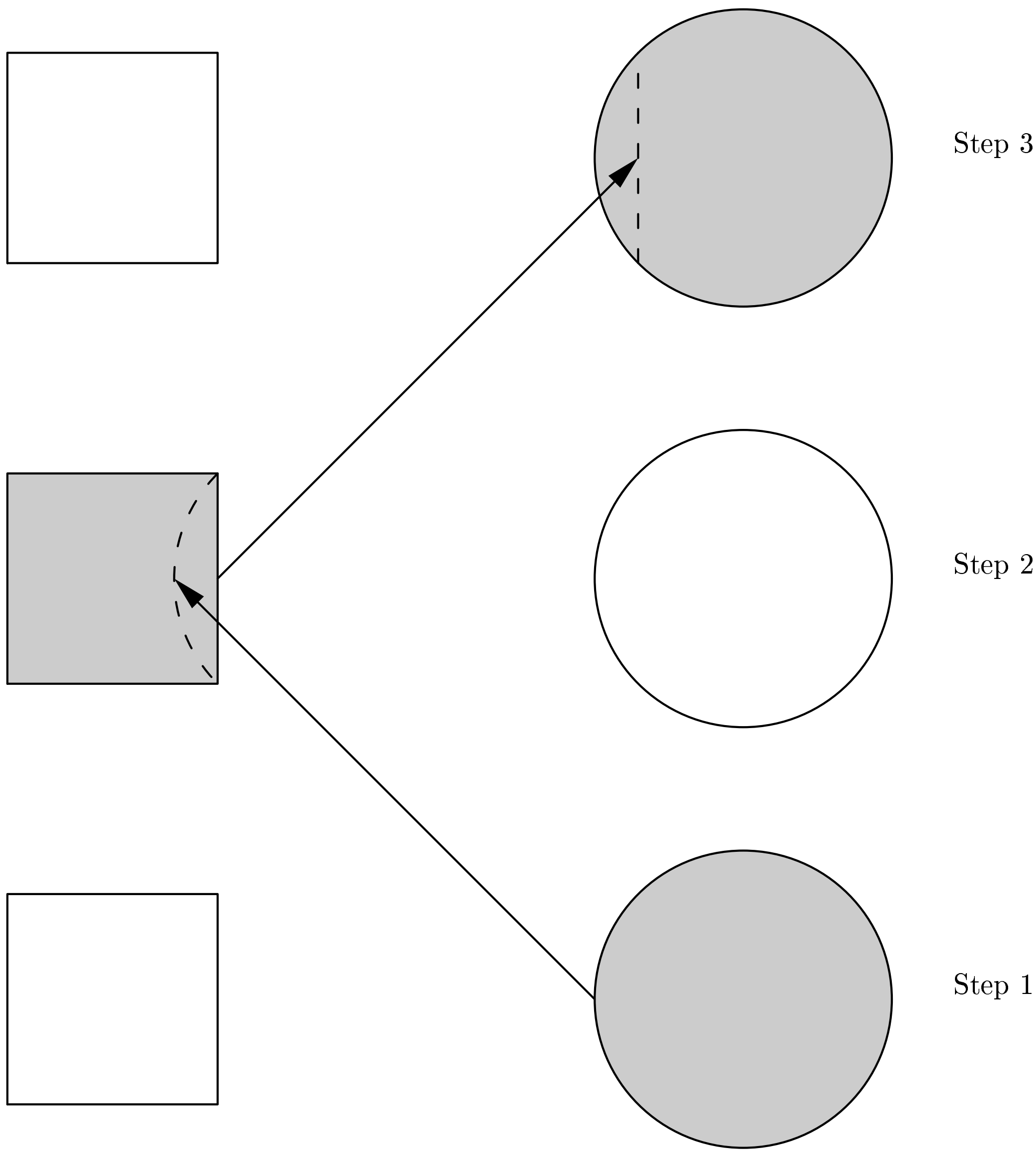


Figure 8.4: Schwarz' alternating procedure

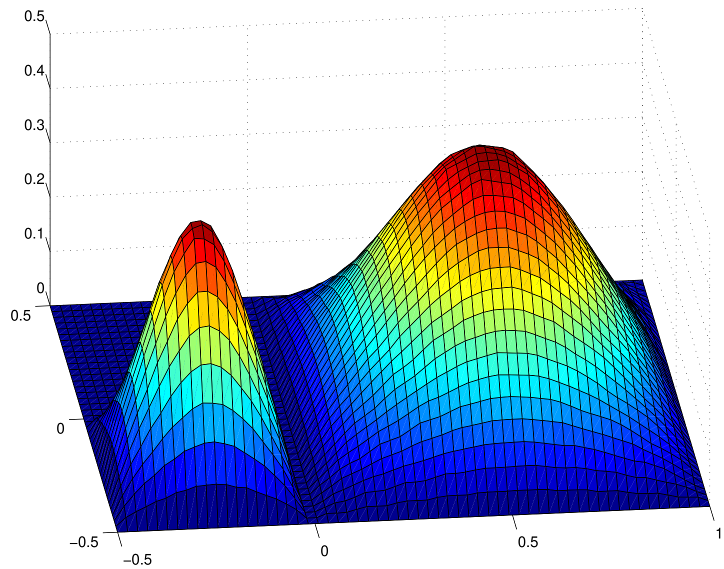


Figure 8.5: Incorrect solution for non-overlaped problem. The result is not differentiable along the boundary between the two regions.

be more conveniently discretized by covering it with polar graph paper. Under such a situation, the grids do not match, and it becomes necessary to transfer points interior to Ω_2 to the boundary of Ω_1 and vice versa. Figure 8.6 shows an example domain with non-mathing grids. Normally, grid values are interpolated for this kind of grid line up pattern.

8.2 Algorithmic Issues

Once the domain is discretized, numerical algorithms must be formulated. There is a definite line drawn between Schwarz (overlapping) and substructuring (non-overlapping) approaches.

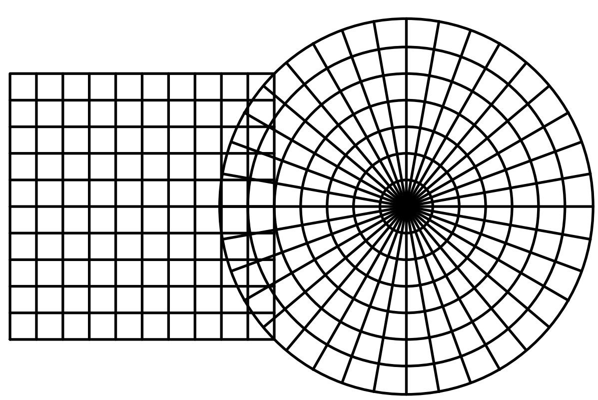


Figure 8.6: Example domain discretized into non-matching grids

8.2.1 Classical Iterations and their block equivalents

Let us review the basic classical methods for solving PDE's
on a discrete domain.

1. Jacobi - At step n, the neighboring values used are from step $n-1$
Using Jacobi to solve the system Au=f requires using repeated applications of the iteration:
$$u_i{}^{(n+1)} = u_i{}^n + \frac{1}{a_i i}[f_i - \sum_{j \neq i} a_{ij} u_j{}^{(n)}] \ \forall \ i$$

2. Gauss-Seidel - Values at step n are used if available, otherwise the values are used from step $n-1$
Gauss-Seidel uses applications the iteration:
$$u_i{}^{(n+1)} = u_i{}^n + \frac{1}{a_{ii}}[f_i - \sum_{j<i} a_{ij} u_j{}^{(n+1)} - \sum_{j>i} a_{ij} u_j{}^{(n)}] \ \forall \ i$$

3. Red Black Ordering - If the grid is a checkerboard, solve all red points in parallel using black values at $n-1$, then solve all black points in parallel using red values at step n For the checkerboard, this corresponds to the pair of iterations:
$$u_i{}^{(n+1)} = u_i{}^n + \frac{1}{a_i i}[f_i - \sum_{j \neq i} a_{ij} u_j{}^{(n)}] \ \forall \ i \ even$$
$$u_i{}^{(n+1)} = u_i{}^n + \frac{1}{a_i i}[f_i - \sum_{j \neq i} a_{ij} u_j{}^{(n+1)}] \ \forall \ i \ odd$$

Analogous *block* methods may be used on a domain that is decomposed into a number of multiple regions. Each region is thought of as an element used to solve the larger problem. This is known as block Jacobi, or block Gauss-Seidel.

1. Block Gauss-Seidel - Solve each region in series using the boundary values at n if available.

2. Block Jacobi - Solve each region on a separate processor in parallel and use boundary values at $n-1$. (Additive scheme)

3. Block coloring scheme - Color the regions so that like colors do not touch and solve all regions with the same color in parallel. (Multiplicative scheme)

The block Gauss-Seidel algorithm is called a multiplicative scheme for reasons to be explained shortly. In a corresponding manner, the block Jacobi scheme is called an additive scheme.

8.2.2 Schwarz approaches: additive vs. multiplicative

A procedure that alternates between solving an equation in Ω_1 and then Ω_2 does not seem to be parallel at the highest level because if processor 1 contains all of Ω_1 and processor 2 contains all of Ω_2 then each processor must wait for the solution of the other processor before it can execute. Figure 8.4 illustrates this procedure. Such approaches are known as multiplicative approaches because of the form of the operator applied to the error. Alternatively, approaches that allow for the solution of subproblems simultaneously are known as additive methods. The latter is illustrated in Figure 8.7. The difference is akin to the difference between Jacobi and Gauss-Seidel.

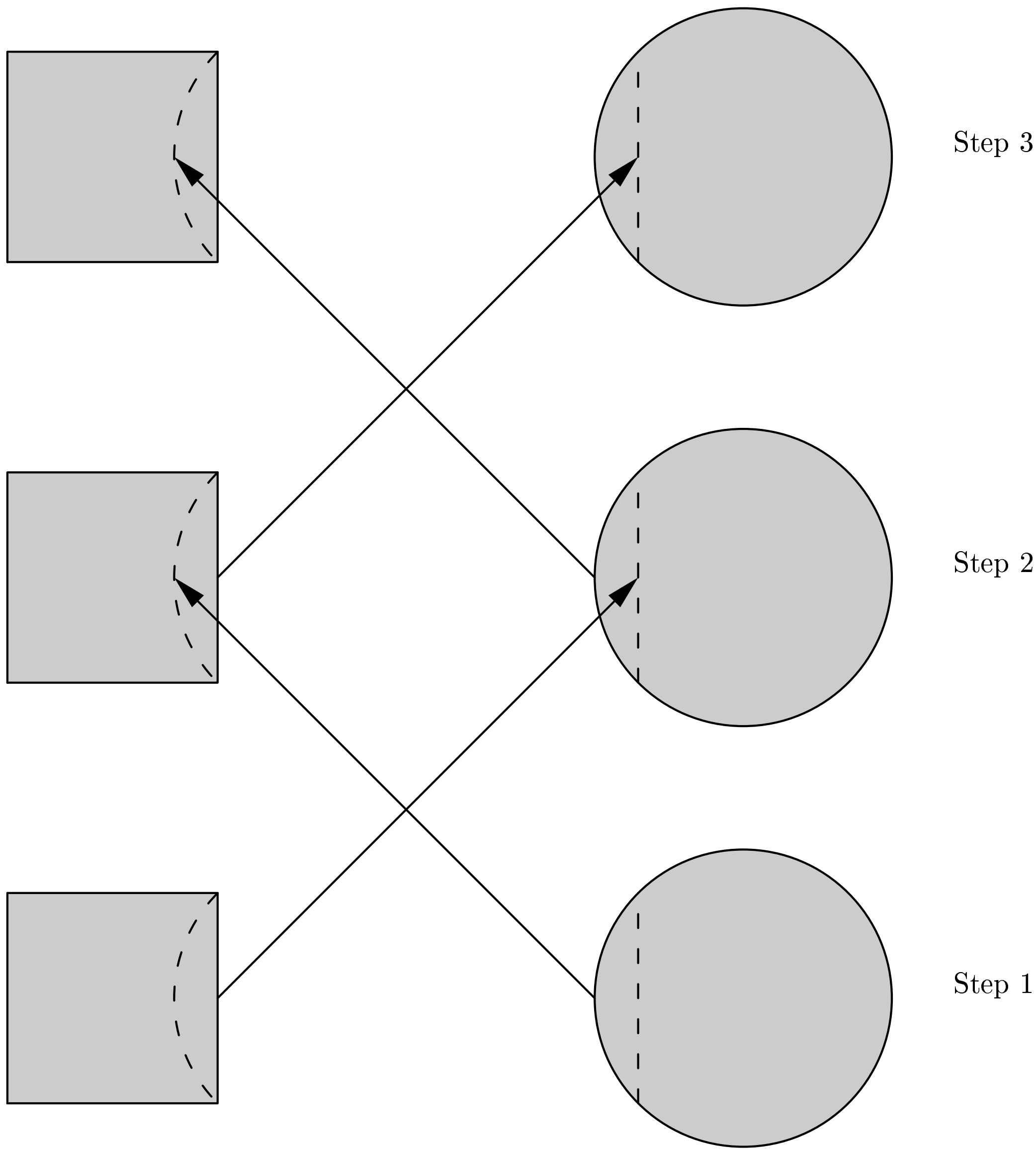


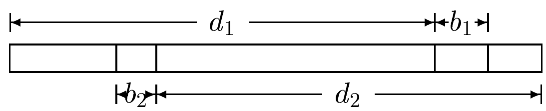
Figure 8.7: Schwarz' alternating procedure (additive)

Overlapping regions: A notational nightmare?

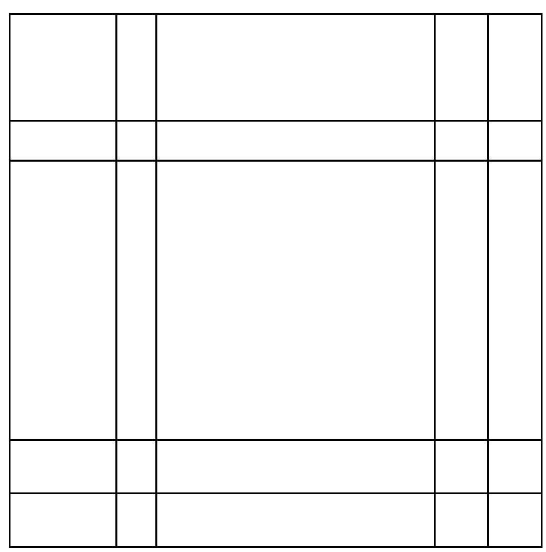
When the grids match it is somewhat more convenient to express the discretized PDE as a simple matrix equation on the gridpoints.

Unfortunately, we have a notational difficulty at this point. It is this difficulty that is probably the single most important reason that domain decomposition techniques are not used as extensively as they can be. Even in the two domain case, the difficulty is related to the fact that we have domains 1 and 2 that overlap each other and have internal and external boundaries. By setting the boundary to 0 we can eliminate any worry of external boundaries. I believe there is only one reasonable way to keep the notation manageable. We will use subscripts to denote subsets of indices. d_1 and d_2 will represent those nodes in domain 1 and domain 2 respectively. b_1 and b_2 will represent those notes in the boundary of 1 and 2 respectively that are not external to the entire domain.

Therefore u_{d_1} denotes the subvector of u consisting of those elements interior to domain 1, while A_{u_1,b_1} is the rectangular subarray of A that map the interior of domain 1 to the internal boundary of domain 1. If we were to write u^T as a row vector, the components might break up as follows (the overlap region is unusually large for emphasis:)



Correspondingly, the matrix A (which of course would never be written down) has the form



The reader should find A_{b_1,b_1} etc., on this picture. To further simplify notation, we write 1 and 2 for d_1 and d_2,1_b and 2_b for b_1 and b_2, and also use only a single index for a diagonal block of a matrix (i.e. $A_1 = A_{11}$).

Now that we have leisurely explained our notation, we may return to the algebra. Numerical analysts like to turn problems that may seem new into ones that they are already familiar with. By carefully writing down the equations for the procedure that we have described so far, it is possible to relate the classical domain decomposition method to an iteration known as *Richardson iteration.* Richardson iteration solves $Au = f$ by computing $u^{k+1} = u^k + M(f - Au^k)$, where M is a "good" approximation to A^{-1}. (Notice that if $M = A^{-1}$, the iteration converges in one step.)

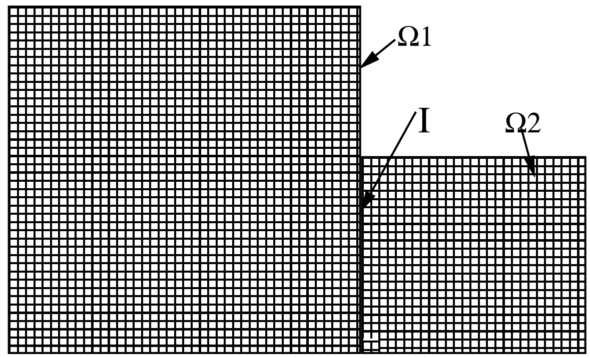


Figure 8.8: Problem Domain

The iteration that we described before may be written algebraically as

$$A_1 u_1^{k+1/2} + A_{1,1_b} u_{1_b}^k = f_1$$

$$A_2 u_2^{k+1} + A_{2,2_b} u_{2_b}^{k+1/2} = f_2$$

Notice that values of $u^{k+1/2}$ updated by the first equation, specifically the values on the boundary of the second region, are used in the second equation.

With a few algebraic manipulations, we have

$$u_1^{k+1/2} = u_1^{k-1} + A_1^{-1}(f - Au^{k-1})_1$$

$$u_2^{k+1} = u_2^{k+1/2} + A_2^{-1}(f - Au^{k+1/2})_2$$

This was already obviously a Gauss-Seidel like procedure, but those of you familiar with the algebraic form of Gauss-Seidel might be relieved to see the form here.

A roughly equivalent block Jacobi method has the form

$$u_1^{k+1/2} = u_1^{k-1} + A_1^{-1}(f - Au^{k-1})_1$$

$$u_2^k = u_2^{k+1/2} + A_2^{-1}(f - Au^k)_2$$

It is possible to eliminate $u^{k+1/2}$ and obtain

$$u^{k+1} = u^k + (A_1^{-1} + A_2^{-1})(f - Au^k),$$

where the operators are understood to apply to the appropriate part of the vectors. It is here that we see that the procedure we described is a Richardson iteration with operator $M = A_1^{-1} + A_2^{-1}$.

8.2.3 Substructuring Approaches

Figure 8.8 shows an example domain of a problem for a network of resistors or a discretized region in which we wish to solve the Poisson equation, $\bigtriangledown^2 v = g$. We will see that the discrete version of the Steklov-Poincaré operator has its algebraic equivalent in the form of the Schur complement.

In matrix notation, $Av = g$, where

$$A = \begin{pmatrix} A_1 & 0 & A_{1I} \\ 0 & A_2 & A_{2I} \\ A_{I1} & A_{I2} & A_I \end{pmatrix}$$

One of the direct methods to solve the above equation is to use LU or LDU factorization. We will do an analogous procedure with blocks. We can rewrite A as,

$$A = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ A_{I1}A_1^{-1} & A_{I2}A_2^{-1} & I \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S \end{pmatrix} \begin{pmatrix} A_1 & 0 & A_{1I} \\ 0 & A_2 & A_{2I} \\ 0 & 0 & I \end{pmatrix}$$

where,

$$S = A_I - A_{I1}A_1^{-1}A_{1I} - A_{I2}A_2^{-1}A_{2I}$$

We really want A^{-1}

$$A^{-1} = \begin{pmatrix} A_1^{-1} & 0 & -A_1^{-1}A_{1I} \\ 0 & A_2^{-1} & -A_2^{-1}A_{2I} \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ -A_{I1}A_1^{-1} & -A_{I2}A_2^{-1} & I \end{pmatrix} \tag{8.1}$$

Inverting S turns out to be the hardest part.

$$A^{-1} \begin{pmatrix} V_{\Omega 1} \\ V_{\Omega 2} \\ V_{Interface} \end{pmatrix} \begin{matrix} \to Voltages & in & region & \Omega_1 \\ \to Voltages & in & region & \Omega_2 \\ \to Voltages & at & interface & \end{matrix}$$

Let us examine Equation 8.1 in detail.

In the third matrix,
A_1^{-1} - Poisson solve in Ω_1
A_{I1} - is putting the solution onto the interface
A_2^{-1} - Poisson solve in Ω_2
A_{I2} - is putting the solution onto the interface

In the second matrix,
Nothing happening in domain 1 and 2
Complicated stuff at the interface.

In the first matrix we have,
A_1^{-1} - Poisson solve in Ω_1
A_2^{-1} - Poisson solve in Ω_2
$A_1^{-1}A_{1I}$ and $A_2^{-1}A_{1I}$- Transferring solution to interfaces

In the above example we had a simple 2D region with neat squares but in reality we might have to solve on complicated 3D regions which have to be divided into tetrahedra with 2D regions at the interfaces. The above concepts still hold.

Getting to S^{-1},

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ c/a & 1 \end{pmatrix} \begin{pmatrix} a & b \\ 0 & d - bc/a \end{pmatrix}$$

where, $d - bc/a$ is the Schur complement of d.

In Block form

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ CA^{-1} & 1 \end{pmatrix} \begin{pmatrix} A & B \\ 0 & D - CA^{-1}B \end{pmatrix}$$

We have

$$S = A_I - A_{I1}A_1^{-1}A_{1I} - A_{I2}A_2^{-1}A_{2I}$$

Arbitrarily break A_I as

$$A_I = A_I^1 + A_I^2$$

Think of A as

$$\begin{pmatrix} A_1 & 0 & A_{1I} \\ 0 & 0 & 0 \\ A_{I1} & 0 & A_I^1 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & A_2 & A_{2I} \\ 0 & A_{I2} & A_I^2 \end{pmatrix}$$

Schur Complements are

$$S^1 = A_I^1 - A_{I1}A_1^{-1}A_{1I}$$

$$S^2 = A_I^2 - A_{I2}A_2^{-1}A_{2I}$$

and

$$S = S^1 + S^2$$

$A_1^{-1} \to$ Poisson solve on Ω_1
$A_2^{-1} \to$ Poisson solve on Ω_2
A_{I1} $\Omega_1 \to I$
A_{21} $\Omega_2 \to I$
A_{1I} $I \to \Omega_1$
A_{2I} $I \to \Omega_2$

Sv - Multiplying by the Schur Complement involves 2 Poisson solves and some cheap transferring.

$S^{-1}v$ should be solved using Krylov methods. People have recommended the use of S_1^{-1} or S_2^{-1} or $(S_1^{-1} + S_2^{-1})$ as a preconditioner

8.2.4 Accellerants

Domain Decomposition as a Preconditioner

It seems wasteful to solve subproblems extremely accurately during the early stages of the algorithm when the boundary data is likely to be fairly inaccurate. Therefore it makes sense to run a few steps of an iterative solver as a preconditioner for the solution to the entire problem.

In a modern approach to the solution of the entire problem, a step or two of block Jacobi would be used as a preconditioner in a Krylov based scheme. It is important at this point not to lose track what operations may take place at each level. To solve the subdomain problems, one might use multigrid, FFT, or preconditioned conjugate gradient, but one may choose to do this approximately during the early iterations. The solution of the subdomain problems itself may serve as a preconditioner to the solution of the global problem which may be solved using some Krylov based scheme.

The modern approach is to use a step of block Jacobi or block Gauss-Seidel as a preconditioner for use in a Krylov space based subsolver. There is not too much point in solving the subproblems exactly on the smaller domains (since the boundary data is wrong) just an approximate solution suffices → **domain decomposition preconditioning**

Krylov Methods - Methods to solve linear systems : **A**u=g . Examples have names such as the Conjugate Gradient Method, GMRES (Generalized Minimum Residual), BCG (Bi Conjugate Gradient), QMR (Quasi Minimum Residual), CGS (Conjugate Gradient Squared). For this lecture, one can think of these methods in terms of a black-box. What is needed is a subroutine that given u computes Au. This is a matrix-vector multiply in the abstract sense, but of course it is not a dense matrix-vector product in the sense one practices in undergraduate linear algebra. The other needed ingredient is a subroutine to *approximately* solve the system. This is known as a preconditioner. To be useful this subroutine must roughly solve the problem quickly.

Course (Hierarchical/Multilevel) Techniques

These modern approaches are designed to greatly speed convergence by solving the problem on different sized grids with the goal of communicating information between subdomains more efficiently. Here the "domain" is a course grid. Mathematically, it is as easy to consider a contiguous domain consisting of neighboring points, as it is is to consider a course grid covering the whole region.

Up until now, we saw that subdividing a problem did not directly yield the final answer, rather it simplified or allowed us to change our approach in tackling the resulting subproblems with existing methods. It still required that individual subregions be composited at each level of refinement to establish valid conditions at the interface of shared boundaries.

Multilevel approaches solve the problem using a coarse grid over each sub-region, gradually accommodating higher resolution grids as results on shared boundaries become available. Ideally for a well balanced multi-level method, no more work is performed at each level of the hierarchy than is appropriate for the accuracy at hand.

In general a hierarchical or multi-level method is built from an understanding of the difference between the damping of low frequency and high components of the error. Roughly speaking one can kill of low frequency components of the error on the course grid, and higher frequency errors on the fine grid.

Perhaps this is akin to the Fast Multipole Method where p poles that are "well-separated" from a given point could be considered as clusters, and those nearby are evaluated more precisely on a finer grid.

8.3 Theoretical Issues

This section is not yet written. The rough content is the mathematical formulation that identifies subdomains with projection operators.

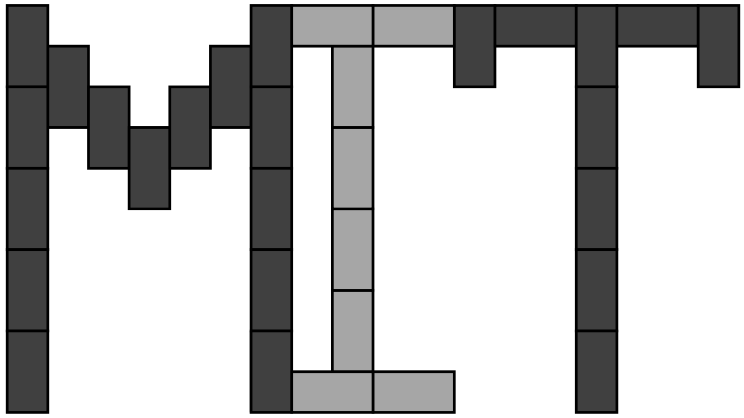


Figure 8.9: MIT domain

8.4 A Domain Decomposition Assignment: Decomposing MIT

Perhaps we have given you the impression that entirely new codes must be written for parallel computers, and furthermore that parallel algorithms only work well on regular grids. We now show you that this is not so.

You are about to solve Poisson's equation on our MIT domain:

Notice that the letters MIT have been decomposed into 32 rectangles – this is just the right number for solving

$$\frac{\partial^2 u}{dx^2} + \frac{\partial^2 u}{dy^2} = \rho(x, y)$$

on a 32 processor machine.

To solve the Poisson equation on the individual rectangles, we will use a FISHPACK library routine. (I know the French would cringe, but there really is a library called FISHPACK for solving the Poisson equation.) The code is old enough (from the 70's) but in fact it is too often used to really call it dusty.

As a side point, this exercise highlights the ease of grabbing kernel routines off the network these days. High quality numerical software is out there (bad stuff too). One good way to find it is via the World Wide Web, at http://www.netlib.org. The software you will need for this problem is found at http://www.netlib.org/netlib/fishpack/hwscrt.f.

All of the rectangles on the MIT picture have sides in the ratio 2 to 1; some are horizontal while others are vertical. We have arbitrarily numbered the rectangles accoding to scheme below, you might wish to write the numbers in the picture on the first page.

```
4                10 21 21 22 22 23 24 24 25 26 26 27
4  5           9 10    20       23       25       27
3  5  6     8  9 11    20                28
3     6  7  8    11    19                28
2        7       12    19                29
2                12    18                29
1                13    18                30
1                13    17                30
0                14    17                31
0                14 15 15 16 16          31
```

In our file neighbor.data which you can take from `~edelman/summer94/friday` we have encoded information about neighbors and connections. You will see numbers such as

```
 1  0  0  0  0  0
 4  0  0  0  0  0
 0  0  0  0
 0
```

This contains information about the 0th rectangle. The first line says that it has a neighbor 1. The 4 means that the neighbor meets the rectangle on top. (1 would be the bottom, 6 would be the lower right.) We starred out a few entries towards the bottom. Figure out what they should be.

In the actual code (solver.f), a few lines were question marked out for the message passing. Figure out how the code works and fill in the appropriate lines. The program may be compiled with the makefile.