1994

Processing PDE Interface Conditions II

Tzvetan Drashansky

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
94-066

PROCESSING PDE INTERFACE CONDITIONS -- II

Tzvetan Drashansky
John R. Rice

Purdue University
Computer Sciences Department
West Lafayette, IN 47907

CSD-TR-94-066
September 1994

PROCESSING PDE INTERFACE CONDITIONS — II

Tzvetan Drashansky and John R. Rice
Department of Computer Sciences
Purdue University
West Lafayette IN 47907

Contents

1. INTRODUCTION

This report is a continuation of [8]; it is intended to gather some ideas and information and to provide extensions of the previous material.

2. INTERFACE RELAXATION METHODS FOR PDES

Over the past decade a multitude of domain decomposition methods have been introduced for solving PDEs. Many of these have used the idea of solving the PDE (or a discretized approximation

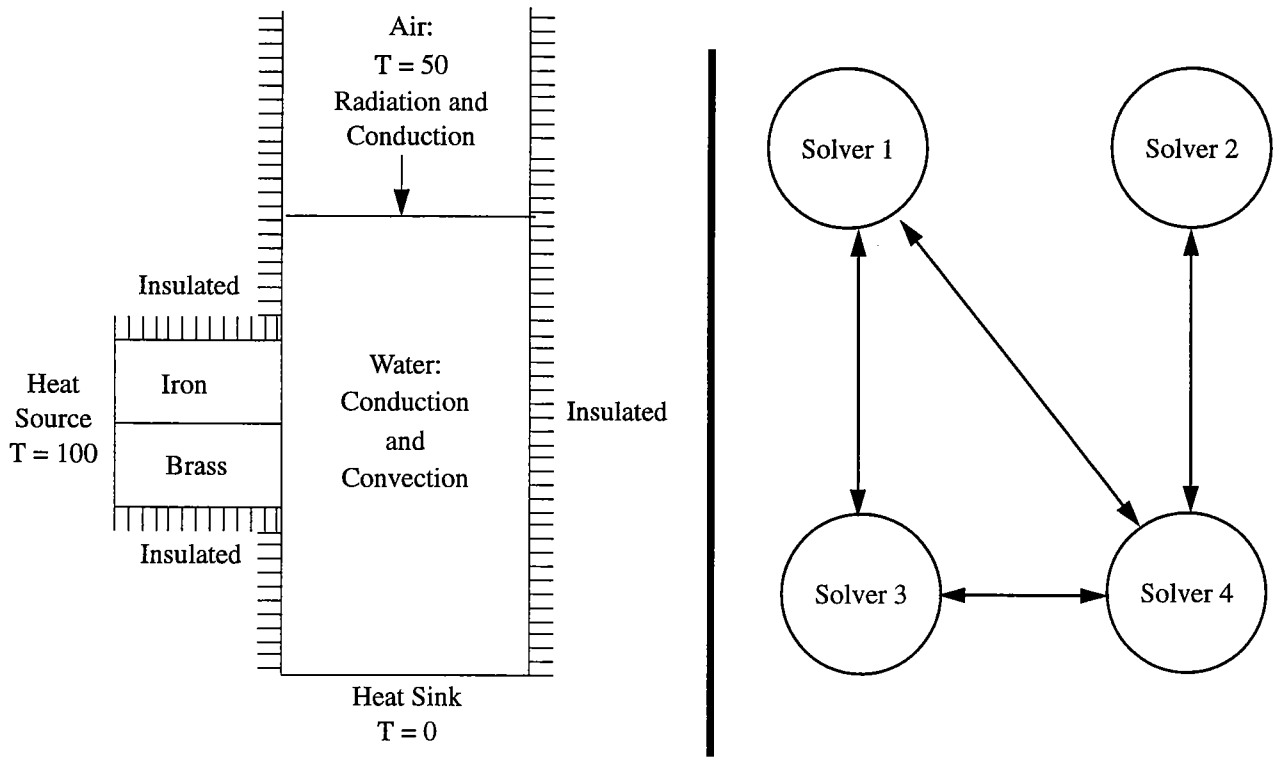

Figure 1.: A simple heat flow problem (left). The PDE model of this physical phenomenon can be solved by a network (right) of four solvers.

of it) on the individual subdomains and then performing an iteration to satisfy the conditions that must hold on the interfaces between subdomains. These are called *interface relaxation methods* and are discussed in [6] and the references cited there. While the method is conceptually simple, its mathematical analysis is very challenging and has been carried out in full only for rather simple model problems. Experiments strongly suggest that the method has the potential to be practical for solving a very wide class of PDE applications. In this section we present a general statement of the problem to be solved and of the method. The intention is to provide a simple framework for discussing the method and no mathematical analysis is made. The discussion is taken from [6].

Figure 1 shows the schematic (left) of a PDE application and the structure of its decomposition (right) into four subdomains. The connections between the subdomains are the interface conditions that must be satisfied. In this example, these conditions are obtained from knowledge of the physics; in other cases the subdomain decomposition is created artificially, the interface conditions are derived from mathematics, and there may be several alternatives for them. We denote the domains by Ω_i and Γ_{ij} is the interface between Ω_i and Ω_j; that is, it is their common boundary piece. On each subdomain the solution U_i satisfied a physical law represented by a PDE L_i. Thus we have

(1) $$L_i U_i = f_i \text{ in } \Omega_i \text{ for } i = 1, 2, \ldots$$

Along each interface there are conditions to be satisfied. Typically, for second order PDEs, there are two physical or mathematical conditions involving values and normal derivatives of the solutions on the neighboring subdomains. Thus on each interface Γ_{ij} relationships of the following form must hold:

$$g_{ij}^{(k)}(U_i, \frac{\partial U_i}{\partial n}) = g_{ji}^{(k)}(U_j, \frac{\partial U_j}{\partial n}) \qquad k = 1, 2, \text{ all } i, j \tag{2}$$

Common examples of these conditions are

$$U_i = U_j \qquad \text{(continuity of solutions)}$$

$$\frac{\partial U_i}{\partial n} = \frac{\partial U_j}{\partial n} \qquad \text{(continuity of slopes)}$$

$$\frac{\partial(a_1(x,y)U_i)}{\partial n} + a_2(x,y) = \frac{\partial(b_1(x,y)U_j)}{\partial n} + b_2(x,y) \qquad \text{(flux continuity)}$$

$$a_1(x,y)U_i + a_2(x,y)\frac{\partial U_i}{\partial n} + a_3(x,y) = b_1(x,y)U_j + b_2(x,y)\frac{\partial U_j}{\partial n} + b_3(x,y)$$
(mixed interface conditions)

The mathematical objective is to satisfy equations (1) and (2).

The *interface relaxation method* iteratively solves the PDE problems (1) independently and then adjusts values along the interfaces to better satisfy the interface conditions (2). There are many boundary conditions that can be used to determine a solution for $Lu = f$ and thus there are many ways to carry out the iteration. The following two simple examples illustrate the variety of possibilities.

Alternating Dirichlet/Neumann [7]. Guess at Dirichlet values U_i along all the interfaces. Then solve $LU_i = f_i$ on all the domains independently with these Neumann conditions. Take these solutions U_i and compute their normal derivatives along all the interfaces. These normals will not agree everywhere (unless the PDE problem is solved), so average them along the interfaces and take these average normals as Neumann boundary conditions along all the interfaces. Then solve $LU_i = f_i$ on all the domains independently with these Neumann conditions. The solutions U_i computed will not agree everywhere along the interfaces (unless the PDE problem is solved), so average them along the interfaces and take the averages as Dirichlet boundary conditions. Repeat the iteration until converged.

Smoothing [5]. Guess at Dirichlet values along all the interfaces. Then solve $LU_i = f_i$ on all the domains independently. Along a paarticular interface Γ_{ij} consider the solutions U_i and U_j computed on each side. Take the new Dirichlet boundary values to be

$$U_i = U_j - \frac{1}{2}(\frac{\partial U_i}{\partial n} + \frac{\partial U_j}{\partial n})$$

Note that the normals to Γ_{ij} point in opposite directions for the neighboring domains so the "smoothing" term is actually a difference. The same formula is used for U_j and the iteration is containued until convergence.

With these two examples in mind, we can formulate the interface relaxation method mathematically as follows. Let $\mathbf{X} = (U, \frac{\partial U}{\partial n})$ be the vector of values along interface Γ_{ij} of Ω_i (we omit subscripts on $\mathbf{X}$ to simplify the notation).

Step 1. Choose information $\mathbf{X}^{old}$ as boundary conditions to determine the PDE solution in Ω_i.

Step 2. Solve the PDE $LU_i = f_i$ on Ω_i to obtain U_i^{new}.

Step 3. Use the U_i^{new}, U_j^{new} values to evaluate how well the interface conditions (2) are satisfied along Γ_{ij}. Use a *relaxation formula* to compute a new value $\mathbf{X}^{new}$.

Step 4. Iterate steps 1 to 3 until convergence.

It is important to note that there are many variations of this method. In particular,

- The relaxation formulas need not be the same for each iteration, or for each interface, or even for each neighbor of a given interface.
- The values taken to determine the solution in step 1 need not be the same for each iteration, or for each interface, or even for each neighbor of a given interface.

The discussion above is entirely in terms of continuous functions, it assumes that *functions* U_i, $\frac{\partial U_i}{\partial n}$ are produced each time a PDE is solved in a subdomain. This view is the best way to understand the relaxation process involved, but in practice the PDEs are approximately solved on the subdomains in terms of a discrete set of unknowns. It is argued in [8] that every PDE solver has the capability to produce *functions* (not just a discrete set of values) for U_i and $\frac{\partial U_i}{\partial n}$ even if that capability has not been explicitly provided in the software. We believe that every PDE solver should have this capability explicitly provided.

3. STRUCTURE OF A COLLABORATING PDE SOLVERS SYSTEM

The idea of collaborating PDE solvers is: make a domain decomposition of a complicated problem, assign a PDE solver to each subdomain and then have the solvers "collaborate" in satisfying the interface conditions. Interface relaxation is a natural method on which to base the collaboration. This approach has been explored experimentally in [3, 4, 5] and found to be quite effective. The basic component of the structure is the linked pair of solvers shown in Figure 2.
As shown in Figure 1, a particular PDE solver may be linked to several others. The actual user for such a system should have a GUI (graphical user interface) such as McFaddin's RELAX system where the subdomain and solver are represented by the geometric shape of the subdomain and the link is represented by the interface Γ_{ij}, see Figure 3.

The PDE solvers for each subdomain are large software systems designed to solve a single PDE problem. The system software encapsulates these solvers and provides the input (solution values

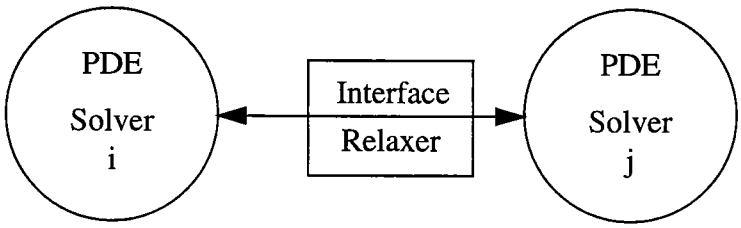

Figure 2.: The basic component of the structure of a linked pair of PDE solvers.

U_i and derivatives $\frac{\partial U_i}{\partial n}$) to the interface relaxers. These, in turn, provide the input (PDE boundary conditions) to the solvers for their next solutions. More specifically, the system should provide:

Solver:

access to solution values and derivatives
error and convergence (size of change) indicators
access to solution plots and results
access to geometry for icon generation
access to the solver user interface

Interface Relaxer:

error and convergence (size of change) indicators
access to the relaxer user interface

General:

a global coordinate system
synchronization policies and control
sequencing policies and control
performance data (errors, changes, compute times)

The interface relaxer is really part of the system as there are no pre-existing modules for this. These are very small modules compared to even a simple PDE solver. Synchronization policies refer to when a solver or relaxer starts a new computation, e.g., a PDE solver could restart whenever any new interface data are available or it could wait until all of it is new. Sequencing policies specify the global ordering of the solver execution. Examples of simple sequencing policies are:

round-robin
skip subdomains which have already converged
user controlled (e.g., single step by clicking)

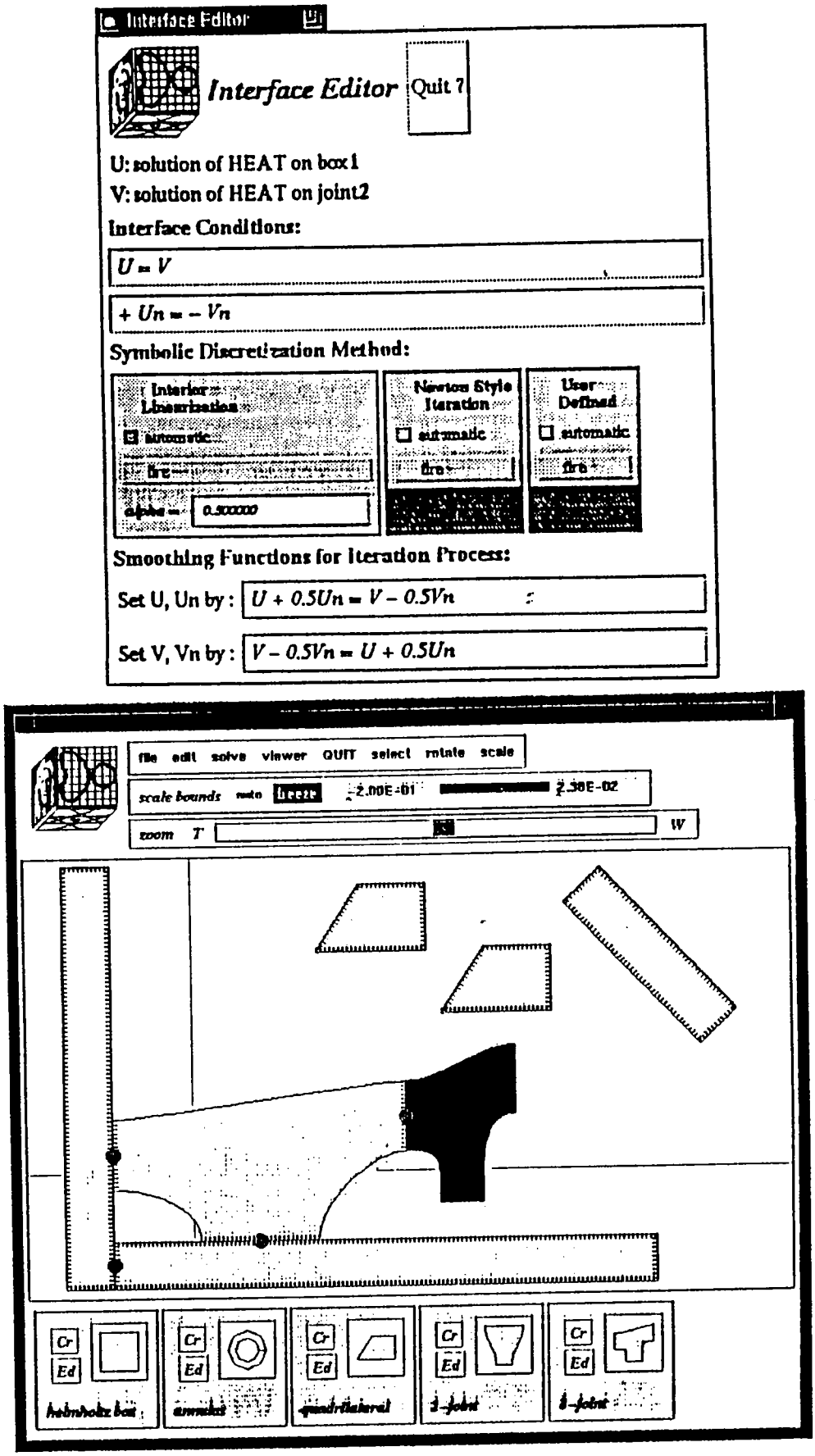

Figure 3.: User interface in the RELAX system. The dots identify interfaces which can be "opened" by clicking. The solvers in the subdomains can be "opened" by clicking in the subdomain.

4. INTERFACE PROCESSING TOOLS

The system of the previous section requires three unrelated tools:

A. *Relaxation Controller.* This tool allows the relaxation formulas to be specified and modified. It also provides information about the performance and history of the relaxation, e.g., values and derivatives along an interface, errors and changes in satisfying the interface conditions.

B. *Message Handler.* This tool collects data from a solver, organizes them and passes them to the interface relaxer. Efficiency is a principal concern both in setting up the data structures for the information exchange initially and in transferring the data (e.g., as for a message passing multiprocessor).

C. *Value/Derivative Handler.* This tool takes data available from the PDE solvers and generates input $\mathbf{X} = (U_i, \frac{\partial U_i}{\partial n})$, etc., for the interface relaxation. There are three cases to consider:

(i) The PDE solver provides accurate values and derivatives at any point on the interface. This handler then has no work.

(ii) The PDE solver provides accurate formulas (in terms of its unknowns) for values and derivatives at any point on the interface. This makes the work for the handler simple. This case is also needed if a direct solution of the PDE is being made as described in [8].

(iii) The PDE solver provides a requested number of points and corresponding solution values near a given interface point. The handler must use these values to obtain accurate estimates of the input for the interface relaxer.

The RELAX system has a window to specify relaxation formulas (see Figure 3). While this window does not provide all the desirable capabilities, it indicates the nature of the relaxation controller.

The message handler tool has a setup phase where it queries the PDE solver for the following information:

- local accuracy of the approximate PDE solutions in terms of a typical h value and method order (1, 2, 3, ...)
- the points on the interface where the solver applies conditions
- the types of data the solver can supply for U, $\frac{\partial U}{\partial n}$ along the interface, i.e.,

  values anywhere

  formulas valid anywhere

  solution values at certain points

  solution and derivative values at certain points

whether the certain points include interface points

- the mode it uses for providing the interface conditions, i.e.,

  simple interfaces conditions (smooth value and slopes)

  mixed interface conditions

  flux continuity conditions

  implicit evaluation

The term "implicit evaluation" means that the solver provides values for $g^{(1)}$ and $g^{(2)}$ (see equation (2)) given values for U_i and $\frac{\partial U_i}{\partial n}$. A form of reverse communication is used where the interface relaxer sends (x, y), U and $\frac{\partial U}{\partial n}$ data to the solver and receives $g^{(1)}$ and $g^{(2)}$ data in return. This seems to be the most convenient way to handle nonlinear interface conditions. Once the set up phase is completed, then the message handler transfers lists of values whose meaning is known to both the PDE solver and the interface relaxer.

The value/derivative handler is straight forward except for case (iii). Then it must solve the following problem:

Given: (x_0, y_0)
(x_i, y_i), $i = 1, 2, \ldots, M$
$U(x_i, y_i)$, $i = 1, 2, \ldots, M$

Compute: accurate values for $U(x_0, y_0)$ and $\frac{\partial U}{\partial n}(x_0, y_0)$.

Here "accurate" means using as high degree polynomials as needed to match the (known) accuracy of the PDE solver.

The straightforward approach is to use polynomial interpolation at the M points $(x_i, y_i), i = 1, 2, \ldots, M$ and then evaluate the interpolant at (x_0, y_0). Note that the number M is restricted because the number of basis functions for various polynomial spaces of interest is limited as follows:

M	*Polynomial Space*
3	degree 1 (linear)
4	tensor product degree 1
6	degree 2 (quadratics)
9	tensor product degree 2
10	degree 3 (cubics)
15	degree 4 (quartics)
16	tensor product degree 3
21	degree 5 (quintics)
25	tensor product degree 4

The straightforward approach is flawed because polynomial interpolation at M points in two variables is not a well posed problem, see [1]. Furthermore, the situation is made more delicate by the

fact that all the points tend to be on one side of (x_0, y_0) (see Figure 2 of [8]) and the $U(x_i, y_i)$ values on the interface Γ_{ij} might be unreliable.

Thus we propose to investigate the use of the Hodie method instead of ordinary polynomial interpolation, see [2]. The problem above is then restated as follows:

Given:	(x_0, y_0)		
	(x_i, y_i), $U(x_i, y_i)$	$i = 1, 2, \ldots, m$	the interpolation points
	(x_i, y_i), G_i	$i = m+1, \ldots, M$	the auxiliary points
Determine:	A polynomial of P maximum degree (from one of the sets defined above) so that (with $P_i = P(x_i, y_i)$)		
	$P_i = U(x_i, y_i)$	$i = 1, 2, \ldots, m^*$	
	$g(P_i, \frac{\partial P_i}{\partial n}) = G_i$	$i = m+1, \ldots, M^*$	

where g is an interface condition function, see equation (2), and G_i is the value of the opposite side. Methods to solve this problem are given in the next section.

This polynomial interpolates the computed solutions at some of the interpolation points and satisfies the interface conditions at the auxiliary points. Most of the time we have $M^* = M$.

The process of generating U_i and $\frac{\partial U_i}{\partial n}$ estimates at an interface point is illustrated in Figure 4. The handler is to provide these values at X (the square point) using quadratic polynomials. It normally requires six values to determine the quadratic so it uses the six closest points to X in the right domain. Three cases are illustrated. Case (A) is where the solver can only provide values in the interior of the domain. Note that the distribution of points suggests that some loss of accuracy is likely since the interpolating quadratic is evaluated somewhat away from the six points. Case (B) is where the solver can provide values on the interface. The six points are now much closer to X and more accuracy is expected. Case (C) uses two points labeled A (the triangles) twice, once to interpolate a value and once to either interpolate a derivative or satisfy an interface condition. Now the six points (points A are counted twice) surround X closely.

Examples of difficult configurations for quadratic polynomials that may arise are illustrated in Figure 5 with three cases. In case (A) all six closest points are on a straight line and there are only three degrees of freedom for quadratics. Even so, the quadratic polynomial determined by the three points closest to X should be very satisfactory for estimating values and normal derivatives at X. In case (B) there is a loss of one degree of freedom. On five of these points one can interpolate by $a + bx + cx^2 + dy + ey^2$ but the xy component of the complete quadratic cannot be determined from the sixth point. Thus the interpolant is only accurate to first order. In case (C) the situation is similar to case (A) in that there are only three degrees of freedom. Now, however, the polynomial interpolant is completely useless. It provides only zero order accuracy in estimating U_i at X and it provides no information at all about $\frac{\partial U_i}{\partial n}$ at X.

5. HODIE METHODS TO APPROXIMATE PDE SOLUTION AND DERIVATIVE VALUES

We present in more detail general methods to obtain accurate approximations of all necessary function and derivative values (along the interfaces) which can be applied by the value/derivative handler regardless of the relative positions of the points supplied by the solvers and which provides the best possible accuracy in any given case. The methods use variations and extensions of the interpolation scheme for multidimensional spaces desribed in [1].

5.1. The Interpolation Problem

The interpolation problem is stated in Section 4 (where we want $m^* = m$ and $M^* = M$). Since we may want to apply also the direct method discussed in [8] and, for added flexibility and reusability of the calculations, we would like to find the polynomial $P(x, y)$ in a form that does not depend on the specific data ($U_i = U(x_i, y_i)$ and G_i) of the interpolation (problem) points. More precisely, we want to find polynomials $c_i(x, y)$, $i = 1, \ldots, m$, and $C_i(x, y)$, $i = m+1, \ldots, M$, such that

$$P(x,y) = \sum_{i=1}^{m} U_i c_i(x,y) + \sum_{i=m+1}^{M} C_i(x,y) G_i \tag{3}$$

The functions $c_i(x, y)$ and $C_i(x, y)$ are the coefficients of the interpolation data, the dual basis for the interpolation problem. Having $P(x, y)$ in the above form also allows us to obtain approximations of the derivatives of the solution at the required points. For example, an approximation of $\frac{\partial U(x,y)}{\partial x}$ is

$$\frac{\partial P(x,y)}{\partial x} = \sum_{i=1}^{m} U_i \frac{\partial c_i(x,y)}{\partial x} + \sum_{i=m+1}^{M} G_i \frac{\partial C_i(x,y)}{\partial x}$$

Therefore we consider the problem of how to obtain the coefficient polynomials $c_i(x, y)$ and $C_i(x, y)$.

First, one can easily see that if we find polynomials $c_i(x, y)$ to satisfy the conditions

$$\begin{gathered} c_i(x_j, y_j) = \delta_{ij}, \ i, j = 1, \ldots, m \\ g(c_i, \tfrac{\partial c_i}{\partial \ldots}, \ldots)(x_j, y_j) = 0, \ i = 1, \ldots, m, \ j = m+1, \ldots, M \end{gathered} \tag{4}$$

then we have a set of coefficient functions. For $C_i(x, y)$, one can analogously satisfy the conditions

$$\begin{gathered} C_i(x_j, y_j) = 0, \ i = m+1, \ldots, M, \ j = 1, \ldots, m \\ g(C_i, \tfrac{\partial C_i}{\partial \ldots}, \ldots)|_{(x_j, y_j)} = \delta_{ij}, \ i = m+1, \ldots, M, \ j = m+1, \ldots, M \end{gathered} \tag{5}$$

Thus we have M conditions for each of the polynomials $c_i(x, y)$, $i = 1, \ldots, m$, and $C_i(x, y)$, $i = m+1, \ldots, M$.

Second, we can write any polynomial $p(x, y)$ of appropriate degree in the form

$$p(x,y) = \sum_{i=1}^{M} \alpha_i \phi_i(x,y) \tag{6}$$

where α_i, $i = 1,\ldots,M$ are constants and $\phi_i(x,y)$, $i = 1,\ldots,M$ are some properly chosen set of basis functions. We can express the polynomials $c_i(x,y)$, $i = 1,\ldots,m$, and $C_i(x,y)$, $i = m+1,\ldots,M$ in the form of (6) with coefficients a_{ij}, $i = 1,\ldots,m$, $j = m+1,\ldots,M$. The problem to find the polynomials c_i and C_i becomes the problem to solve the linear system

$$KA = I \tag{7}$$

where $A = (a_{ij})$ and $K = (k_{ij})$ with $k_{ij} = \phi_i(x_j,y_j)$, $i = 1,\ldots,m$, $j = 1,\ldots,M$; $k_{ij} = g(\phi_i, \frac{\partial\phi_i}{\partial\ldots},\ldots)|_{(x_j,y_j)}$, $i = m+1,\ldots,M$, $j = 1,\ldots,M$. I is the $M \times M$ unit matrix.

5.2. The First Method

If the linear system (7) is non-singular and well conditioned then one merely solves it directly. The discussion and examples in Section 4 show that there can be combinations of basis functions and point configurations where (7) is singular or nearly so. To handle this difficulty we consider using more than M basis functions and applying the techniques of [1] to always assure a solution to the interpolations equations (4) and (5).

We calculate a "wider" matrix $K' = (k_{ij})$, $i = 1,\ldots,M$, $j = 1,\ldots,N$ where $N > M$. Our experiments with nearly singular cases suggest that $N = 2M$ usually suffices. In other words, we choose a set of N basis functions $\phi_i(x,y)$, $i = 1,\ldots,N$ from which the algorithm will select the proper set of M basis functions. This set can be the usual combinations of the powers of x and y: $1, x, y, x^2, y^2, xy, x^3, y^3, x^2y, xy^2, \ldots$ with origin in the area of interest and properly scaled in order to get better accuracy. For example, x^3 may become $(x-x_0)^3/h^3$ or $(x-x_{-1})(x-x_0)(x-x_{+1})/h^3$ for some appropriately chosen values (close to the points (x_i,y_i), $i = 1,\ldots,M$) of x_0, x_{-1}, x_{+1} and h. Another alternative is to find two sets of one-dimensional orthogonal polynomials on carefully selected intervals (for x and y) and to use their products as basis functions instead of the powers of x and y.

Then we apply the following algorithm (see the pseudocode below). We do Gaussian elimination of K' with pivoting. If at some point we can not choose a pivot with the absolute value larger than a predefined tolerance constant τ (representing round-off), then we throw away the corresponding column of K' and reduce the number of columns in K' by one. When we have M pivots on the diagonal with absolute value larger than τ we stop and declare the matrix consisting of the original values of those M columns to be the matrix K which we will use to solve the systems (7). The choice of the tolerance τ is important.

In the pseudocode below, K' is an $M \times N$ matrix, `columns` is a set containing the (still) valid columns of K', K is the coefficient matrix of the system (7) that we want to obtain.

INITIALIZATION
columns $= \{1 \ldots M\}$
Calculate $K' = k_{ij}$

FACTORIZATION For $\ell = 1$ to M
    Loop until a pivot is chosen or $|columns < 1$
        $i = columns_\ell$
        Choose a pivot p among k_{ji}, $j \geq i$
        If $p < \tau$ then
            $columns = columns - \{i\}$
        end if
    end loop
    If $|columns| < \ell$ then
        Stop and report failure
      else
        Eliminate k_{ji}, $j > i$
        Perform operations only on those columns in K' that are still in *columns*
    end if
end for
$K = k_{ij}$, $i = 1, \ldots, M$, $j = columns_1, \ldots, columns_M$
Return K

Of course, we can use the factorization above to solve (7) instead of forming another matrix K. The distinction between K' and K is done only for clarity. After solving the systems (7) we have the polynomials c_i and C_i and we may obtain $P(x, y)$ in the form (3).

We note several features of this method. We calculate the coefficient functions (polynomials) only, so if the grid points are not changed from one iteration to the next then the value/derivative handler has to solve the systems (7) only once. After each iteration, the polynomial $P(x, y)$ can be evaluated with the new solution values at the grid points. This method works for most commonly used grids and meshes. It fails only when N is too small and, in principle, we can dynamically increase N (at the "Stop" of the pseudocode) and the matrix K' so as to always obtain an interpolant. Of course, in extreme cases (Figure 5(c), for instance) the interpolant might not provide good accuracy. Our experiments with the method show that it gives the expected accuracy from the given number of points, even though to prove that mathematically is hard.

5.3. The Second Method

The method in the previous subsection has an important disadvantage. In the case of Figure 5(c) the method chooses higher and higher degree basis functions with respect to y but that does not contribute to the accuracy of the approximation of the point X. In other words, the method does not provide an indication when it fails. To obtain such an indication and to reduce the degree of the interpolation polynomial in cases when it is unnecessarily high, we propose the following variation of the first method.

We take the square matrix K and apply the idea of the previous subsection but with respect to the points first (to the conditions for $c_i(x, y)$ and $C_i(x, y)$) rather than with respect to the basis functions. More precisely, we do Gaussian elimination of K with *row* pivoting instead of column

pivoting (and we still eliminate the elements in the corresponding column). In other words, when we search the ith pivot, we search among the elements k_{ij}, $j = i, \ldots, M$ (instead of among the elements k_{ji}, $j = i, \ldots M$) and we eliminate k_{ji}, $j = i+1, \ldots, M$. Thus, if K is singular or almost singular, we encounter a *row* (instead of a column) of elements less than τ in absolute value. Then we throw away that row which is equivalent to discarding a point (a condition for $c_i(x, y)$ and $C_i(x, y)$) following the idea from the previous subsection. We reduce the number of rows in K, obtaining a matrix K'' with M' rows, $M' < M$, and M columns. We stop the process when we have found pivots for all M' rows of K''. Then we use K'' as the input matrix K' for the algorithm given above in order to find M' basis functions for the polynomials $c_i(x, y)$ and $C_i(x, y)$. The resulting polynomial $P(x, y)$ has the same accuracy as if we were using all initial points (conditions). Of course, it is still of no use in cases like Figure 5(c) but we are now able to detect them (and to try to fix the problem, for example, by requesting more points from the solver).

6. PARALLELISM FOR DIRECT SOLVERS

In this section we present a modification of the solution algorithm described in [8], that allows exploiting more parallelism and has more symmetry between the actions of the solvers on the two domains (i.e., more evenly distributed work). To explain our approach, we use the same problem and the same approximation and discretization schemes as in [8]. The main difference is that now we assume that both solvers have the correspondence between the row indices of the shared terms in the two domains (as opposed to the case before, when only the right domain solver was required to have them). One can easily alter the discretization algorithms to achieve that by adding the mirror image of the steps performed in [8] in order to supply the right domain solver with the correspondence of the row indices. Figure 6 presents schematically the solvers' configurations and the mathematical structures after the discretization has been performed.

	u PDE		*v PDE*
(8)	$A\mathbf{u} = \mathbf{f}$		
(9)	$M_1\mathbf{u} = \mathbf{m}_1$	$=$	$M_2\mathbf{v} + \mathbf{m}_2$
(10)	$N_1\mathbf{u} = \mathbf{n}_1$	$=$	$N_2\mathbf{v} + \mathbf{n}_2$
(11)			$B\mathbf{v} = \mathbf{g}$

As we pointed out in [8], the solver v for the right domain can not use those terms in M_2 and N_2 whose corresponding terms in M_1 and N_1 have been used to eliminate some equations in the left domain (i.e., have contained pivot elements). For example, if the term with index 37 in the left domain has a corresponding non-zero term in the right domain with index 56, and if the element $a_{37,37}$ has been used as a pivot during the factorization of the left domain, then the right domain solver cannot use at all its term with index 56 (since it has at least one non-zero coefficient of unknowns from the left domain). In other words, after a pivot has been chosen on the left, the corresponding right domain term cannot be used by the right domain solver. The main idea of the present algorithm is to start simultaneous factorizations of both domains using the above fact in

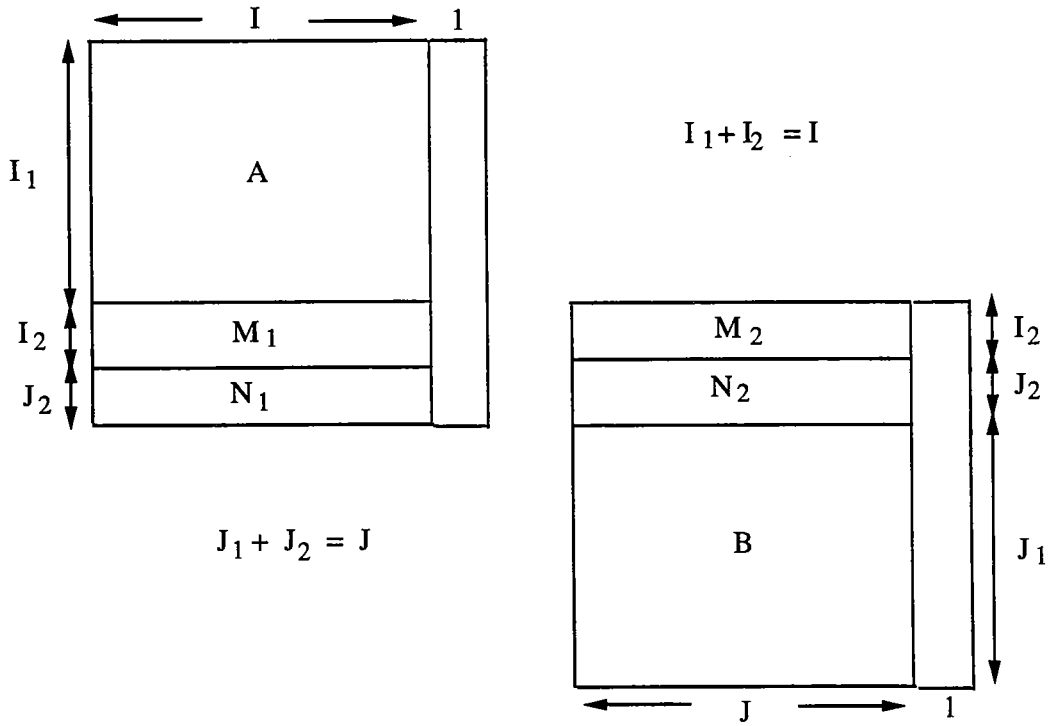

Figure 6.: The sizes of the matrices in the linear system after discretization of the PDEs and interface conditions. The blocks, top to bottom, correspond to equations (3) to (6).

both directions. Of course, we need some synchronization between the solvers so that they do not choose one and the same pivot terms. The factorization proceeds by repeating the following step (please see also the pseudo code below). Both solvers choose a pivot and send its index to the other solver for confirmation. If shared memory is available, a flag corresponding to each index could be used for simplicity and better efficiency. After different pivots have been chosen and agreed on, each solver proceeds with the elimination of the elements in the corresponding column. Note that the terms used as pivots by the other solver must not be eliminated (so that the factorization process in the other domain is not affected), much like the approach in [8] when we could not eliminate in the right domain the terms used as pivots by the left domain. The messages of the type *eliminate q,r,m* can be grouped in a large message which is sent after the elimination is done. Each solver processes the message received from the other solver (performs the operations on the corresponding terms in its domain).

When the factorization of a domain is finished, each solver sends a message so that the other solver will know not to send its pivot indexes for confirmation and not to wait for further *eliminate* messages. Of course, the algorithm continues processing the *eliminate* messages sent by the other solver.

After the matrices for the domains have been factored, the solvers exchange the values of the right hand sides of the pivoting equations. In this algorithm this is easy to do since each solver keeps a list of the other solver's pivots in order to choose its own pivots correctly. When the right-hand sides are received (the messages of type *send index, value* could also be grouped in one larger message), the solvers do not need to communicate anymore since they can proceed with the back substitution.

In the pseudocode for this algorithm given below, P denotes the set of indexes of the pivots (initially empty), PO denotes the set of pivots of the other solver (initially empty), R denotes the set of indexes of the non-pivot terms (initially the whole matrix), I denotes the number of unknowns in the domain, J denotes the number of rows in the domain matrix. The pseudocode describes the actions of either of the two solvers; the actions which are identical or very similar to the original algorithm presented in [7] are not given in detail.

INITIALIZATION
Do the initialization described in [8] and also
$P = \{\}$
$PO = \{\}$
$R = \{1 \ldots J\}$
Determine who changes the pivot in case of collision

FACTORIZATION
For $k = 1$ to I
    Choose a pivot p from R
    If the other solver has not finished yet
        Receive p_{other} (the index corresponding to the index chosen as a pivot
        by the other solver)
        If *collision* and so determined initially
            Change the pivot p
        end if
        If $p_{other} \neq 0$ (i.e., there is a corresponding term to the pivot in this domain)
            $PO = PO + \{p_{other}\}$
            $R = R - \{p_{other}\}$
        end if
    end if
    $P = P + \{p\}$
    $R = R - \{p\}$
    Eliminate the elements in the column p in R
        (send *eliminate* message)s)
    If the other solver has not finished yet
        Receive *eliminate* or *end factorization* message(s)
        Process the message(s)
    end if
end k loop
Send *end factorization* message
While receiving *eliminate* messages
    Process the messages
end while

EXCHANGE OF RIGHT-HAND SIDE TERMS
For $k = 1$ to $|PO|$
    Evaluate term PO_k in v
    send PO_k, v
end k loop
While receiving *send* messages
    Augment the corresponding right-hand sides
end while

BACK SUBSTITUTION
do back substitution as usual; no communication required

The procedure of exchanging the pivots requires some additional discussion. In order to avoid deadlock and to introduce the necessary determinism, the solvers decide which one of them is going to change the pivot if there is a pivot collision, i.e., they choose the corresponding parts of one and the same equation as their pivots at some point of the factorization. That can be done easily by assigning distinct priority numbers (e.g., at random) to solvers. The solver with the larger number gets to change the pivot in case of collision. Once the relation is established, the solvers need to exchange a maximum of three messages per pivot-choosing procedure — to inform the other solver about their own pivot (two messages) and one additional message from the solver who changes its pivot in case of collision. The decision who changes the pivot does not affect the numerical results from the solution process. The parallel algorithm can be applied to more than two solvers, each working on different (but adjacent) domains, so, we have to consider the possibility that the choice of the pivots between one pair of solvers affects the choice of pivots between another pair of solvers. It is clear that such dependence may occur only if there is an equation that has parts in at least three domains (matrices). We claim that such an event indicates some problem in the discretization schemes of the solvers. Indeed, equations with parts in more than one domain involve grid/mesh points that are very close to the interface (if we assume that the grid/mesh is chosen in a way that provides reasonable accuracy, i.e., is dense enough). Then in order to obtain an equation with parts in at least three domains, we need at least two interfaces (domain boundaries) in a very small distance from each other. In other words, the "middle" domain must be very "thin" in neighborhood of the grid points used in the equation. But then the solver in the "middle" domain should use a finer grid since usually such "thin" domains create accuracy and solution problems if the grid is not scaled well to the domain. If there is no equation with parts in more than two domains, then each pair of solvers is independent from the others in their choice of pivots and we can apply the algorithm to more than two solvers simultaneously.

The major advantages of the above algorithm are the degree of parallelism achieved (in the previous algorithms we were not concerned with this aspect of the computation) and its uniformity — every processor uses the code(algorithm) and the algorithm can be easily applied with any number of processors (and domains). The work load of each processor depends on the size of its matrix and, if the sizes are of the same magnitude, then all processors will be busy most of the time and the resource utilization will be high (unlike many domain decomposition algorithms where there is a large part of the computations that is done by a single processor or by a very small number of processors). Even when the sizes of the matrices in different domains differ significantly, this algorithm achieves the maximum possible parallelism — the time to solve all problems is comparable to the time to solve the largest of them.

The potential bottleneck, especially when the number of domains grows, is the exchange of messages necessary to correctly determine the pivots during the factorization. We already pointed out a case when this process can be very efficient — a shared memory multi-processor. Even if the underlying machine is a loosely-coupled multiprocessor, the impact of this message exchange would not be too big if the sizes of the matrices are large (and, hence, the time for the work on column elimination dominates the time spent for message exchange).

7. EFFICIENCY IN MESSAGE PASSING

The modern multi-purpose PDE solvers that can implement parallel algorithms usually run in a message-passing environment. The main reason for that is the relative flexibility of those computers (compared to the SIMD architectures) allowing the users to implement different types of algorithms in a straight-forward manner. An additional attractive feature of the loosely-coupled architectures is the fact that most of the algorithms (and the software) could run with minimal changes on a set of separate (inexpensive) computers connected by a local network with reasonable parameters. Such networks are widely available and, even though a considerable loss in the performance (compared to he multiprocessors) might be observed, their use for computationally intensive tasks provides substantial speedups (over the runtime of a single computer) and better utilization of the computing resources.

The major obstacle in achieving higher speedup and efficiency in all loosely-coupled computing environments is the big difference in the relative speeds between the processors and the communication medium. The low speed of the communication forces programmers to pay serious attention to the design of the message-passing schemes of the parallel algorithms. In this section we discuss some issues related to efficient message passing as part of the interface between the solvers in the implementation of the algorithms and the schemes presented in this report.

A serious difficulty in the design of a good message passing scheme for a given algorithm comes from the well-known tradeoff related to the length and the number of messages. For any communication network, there are some "overhead" operations required before transmitting a message. They include setting up the message (formatting, filling the header, etc.), establishing a connection (if necessary), routing decisions (if necessary), and some more specific operations. The time τ required for these operations usually does not depend heavily on the length of the message. Each system has also a time t_b required to transmit a standard message packet, say one byte. Assume a message contains m bytes, m_h of them in the header and m_d of them containing the (useful) data, $m = m_h + m_d$. Then the time t required to transmit a given message can be expressed (approximately) as

$$t = \tau + m \times t_b \tag{12}$$

For most systems m_h does not depend on the length of the message. Then we may consider a new overhead time τ' where $\tau' = \tau + m_h \times t_b$. Then we can rewrite (12) as

$$t = \tau' + m_d \times t_b \tag{13}$$

In (13) the programmer can control only m_d, τ' is a parameter of the communication network. It is then clear that if we have a certain amount of data to transmit, we would like to form as long a message as possible in order to reduce the overhead penalty. On the other hand, each message usually contains valuable data for the addressee. If we wait until there is "enough" data for a long message then the addressee may be kept waiting unreasonably long. Perhaps if we send a series of short messages, the addressee can work continuously processing the incoming data. The usual way of working out a compromise length of the messages in a parallel program is to find a way to keep the processes busy with local computations (not dependent on the communication) until a

reasonable length message can be produced and transmitted, and then to have the addressee process the message at once (in the ideal case, working until the next message arrives). The most common method of finding enough local work for the processor is to use more coarse-grain parallelism, i.e., to divide the entire job into fewer subtasks, each of appropriately larger size.

That tradeoff is clear in the parallel algorithm presented in Section 6. Each solver performs elimination operations on its matrix once the pivot for each row is chosen. For the elimination of an element in the matrices, there is an *eliminate* message to be sent. There are two feasible ways of composing the messages. First, each *eliminate* message can be sent separately. In this case, the other solver can process its part of the row immediately, but the lengths of the messages are quite small. Second, a single big message per pivot can be formed (during the elimination process, thus reducing τ') and sent at once. It can include all pairs $(factor, element)$ formed in which this pivot is used to eliminate elements. This approach has the added advantage of not repeating the index of the pivot as we would have to do using separate *eliminate* messages. We favor the second solution since one may observe that (if the matrix sizes are comparable) the other solver can perform its local elimination operations and, hence, need not wait without work for the message. In the case of substantial differences in the matrix sizes, the method guarantees that the solver with the largest matrix will not wait. More complicated problems are associated with the negotiation of the pivots since during that procedure none of the solvers can perform useful local work. As we mentioned before, the most efficient solution would use shared data structure containing flags, if such is available. If not, we have to pay the price for the negotiation. It is fairly small in case the solvers choose different pivots in the beginning of the negotiation. The probability of collisions is low, given the different order of the rows in the two domains - the equations in M_1 are placed in the left matrix before the equations in N_1, while those in M_2 (corresponding to M_1) are placed in the right matrix after the ones in N_2. But even if a collision happens, that causes only one additional message to be sent. Note that the algorithm requires relatively small communication throughout the solution process, thus one would expect there to be enough local computation to fill the time between the arrivals of the messages.

We find the message size/frequency tradeoff again when we consider the relaxation scheme described in Sections 2–4. In this case we have to pass the interface and related values to the *Relaxer* and then the boundary values for the next iteration back to the solvers. We do not have much choice in the lengths of the messages here, it is plausible to wait until all the necessary values become available and then send a single message to the relaxer. The relaxer then returns values to the solvers when its short computation is finished. We could, alternatively, consider the distribution of the relaxation computation to the solvers. It is clear that allocating the solvers and the relaxer to different processors, as suggested by Figure 2, is quite inefficient even if we do not take into account message passing — when the solvers work, the relaxer must wait and vice versa. Consider instead a distributed relaxer, see Figure 7. The relaxer is divided into two parts, each one with the information and formulas associated with one of the solvers. Assuming that the relaxation formulas use relatively static information (it changes infrequently during the computation) we can supply the relaxer R_1 with the relaxation formula for S_1 and we can run it on the processor running S_1. The same applies to R_2 and S_2. Then the only communication necessary after each iteration is for S_2 to send its interface and related values to R_1 and for S_1 to send its interface and related

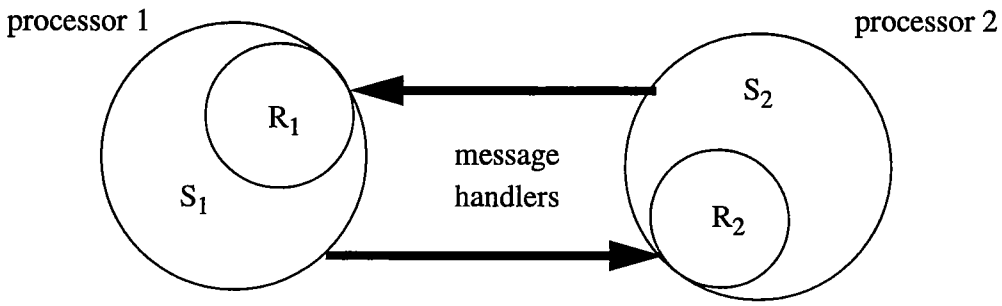

Figure 7.: Schematic of distributing the interface relaxes to the processors responsible for the two PDE solvers.

values to R_2. That distribution accomplishes two goals – it reduces the communication since only one message per iteration must be transmitted in each direction (the values after the relaxation are already in place), and it uses the processors more efficiently.

Another important issue for the relaxation scheme is how we map the topology of the graph (such as in Figure 1) onto the topology of the parallel computer (the local network, for instance, an N-cube topology). For some machines there is considerable advantage from the message passing point of view if solvers connected with an edge on Figure 1 are mapped onto neighboring processors in the parallel machine (two processors are neighbors if they can exchange messages without involving other processors). Of course, this can not be done in the general case but we can consider techniques that can find a mapping close to the optimal one.

8. THE CROSS POINT PROBLEM

The conditions to be satisfied across interfaces are derived from either physical or mathematical considerations. What then are the conditions to be satisfied when more than two subdomains meet (see Figure 8)? These multiple interface points are called *cross points* in [7]. Conditions that are known to hold along the interfaces usually do not "converge" to some common conditions as points on the interfaces converge to a cross point. Discussions of the conditions that should hold at cross points due to physical considerations are rare (we know of none). Visualize, for example, that the domains consist of different materials (air, salt water, oil, sand stone, metal, gravel) and the PDEs model flows of heat or fluids under high pressure. It is plausible that there are no known natural conditions that hold at a cross point and that there are singularities in the PDE solutions at such points.

Many numerical methods naturally involve values at cross points and thus may require some conditions to hold there. If the cross point is artificial (i.e., a domain with a single PDE is subdivided) then mathematical conditions of smoothness can be derived and used. But mathematical analysis [7] of simple model problems suggest greatly reduced convergence rates when cross points are present; some experiments confirm this slow convergence in practice. On the other hand, the experiments of McFaddin with the RELAX system [5] do not show slower convergence from the presence of cross points. The method (5-point star on tensor product grids) in the RELAX system

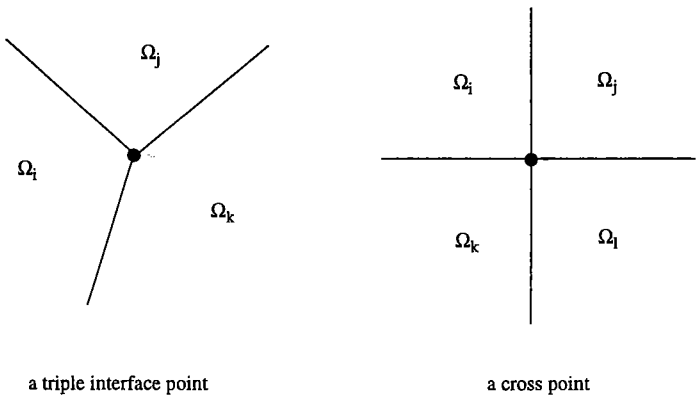

Figure 8.: Cross points are points in two dimensions where three (left), four (right) or more subdomains meet.

never makes any use of information at the cross points even though these points are part of the grids generated.

These skimpy data suggest that the best approach to handling cross points is not to use them at all since

- physical conditions at cross points are indeterminate,
- some analysis and experiments of methods using cross points show poor results [7],
- some experiments with methods ignoring cross points show good results [5].

Of course, it is difficult to make an arbitrary PDE solver automatically omit using a cross point. One of the problems to be solved in this area is the creation of methods to accomplish this.

References

[1] Carl de Boor and A. Ron, Computational aspects of polynomial interpolation in several variables, *Math. Comp.*, **58** (1992), pp. 705–727.

[2] Robert E. Lynch, Hodie approximation of boundary conditions, in *Iterative Methods for Large Linear Systems* (D. Kincaid and L. Hayes, eds.), Academic Press (1990) pp. 135–147.

[3] Scott McFaddin and John R. Rice, RELAX: A platform for software relaxation, in *Expert Systems for Scientific Computing* (Houstis, Rice and Vichnevetsky, eds.) North-Holland, Amsterdam (1992), pp. 125–194.

[4] Scott McFaddin and John R. Rice, Collaborating PDE solvers, *Appl. Num. Math.* **10** (1992) pp. 279–295.

[5] Scott McFaddin, *An object-based problem solving environment for composite partial differential equations*, Ph.D. Thesis, Dept. of Comptuer Sciences, Purdue University, (1992).

[6] Mo Mu and John Rice, Modeling with collaborating PDE solvers — Theory and practice, Tech. Rpt. 94-056, Dept. of Computer Sciences, Purdue University (August, 1994). See also article with the same title in *Contemporary Mathematics*, **XX**, Amer. Math. Soc. (1994) to appear.

[7] A. Quarteroni, F. Pasquarelli, and A. Valli, Heterogeneous domain decomposition: Principles, algorithms, applications, *Fifth Inter. Symp. Domain Decomposition Methods for Partial Differential Equations*, D. Keyes, et. al., eds., SIAM Pubs., Philadelphia, (1992), pp. 129–150.

[8] John R. Rice, Processing PDE interface conditions, Tech. Rpt. 94-041, Dept. of Computer Sciences, Purdue University (June 1994).