Contemporary Mathematics
Volume 180, 1994

Modeling with Collaborating PDE Solvers: Theory and Practice

MO MU AND JOHN R. RICE

ABSTRACT. We consider the problem of modeling very complex physical systems by a network of collaborating PDE solvers. Various aspects of this problem are examined from the points of view of real applications, modern computer science technologies, and their impact on numerical methods. The related methodologies include *network of collaborating software modules*, *object-oriented programming* and *domain decomposition*. We present a domain decomposition approach of collaborating PDE solvers based on interface relaxation. The mathematical properties and application examples are discussed. A software system RELAX is described which is implemented as a platform to test various relaxers and to solve complex problems using this approach. Both theory and practice show that this is a very promising approach for efficiently solving complicated problems on modern computer environments.

1. Introduction

Modeling physical phenomena with scientific computing is an interdisciplinary effort involving engineers, mathematicians and computer scientists. Practical physical systems are often mathematically modeled by complicated partial differential equations (*PDEs*). Their numerical solution requires high performance computers, large software systems and efficient algorithms. The design of numerical PDE algorithms must balance many factors. From the numerical analysis point of view, one often focuses on good approximation, fast convergence, low arithmetic expense, and other mathematical properties. In practical applications, one should be able to handle the complexity and generality of PDE problems. Among the major concerns for software development are software

1991 *Mathematics Subject Classification.* Primary 65N55, 65F10; Secondary 65Y05, 65c20.
The first author was supported in part by the National Science Foundation grant CCR-8619817 and the Hong Kong RGC DAG93/94.SC10. The second author was supported in part by the Air Force Office of Scientific Research grants, 88-0243, F49620-92-J-0069 and the Strategic Defense Initiative through Army Research Office contract DAAL03-86-K-0106.
The final version of this paper will be submitted for publication elsewhere.

productivity, complexity, reusability, maintenance, portability, and other quality issues. Modern software technologies and concepts are needed. In addition, the use of parallel computing leads to issues, such as parallel algorithms, communication cost, and scalability. Obviously, many of these objectives conflict with each other. The principal trade off is programming effort versus execution time efficiency. We examine various aspects of the simulation problem from the practical point of view. These considerations lead to the domain decomposition approach which we call *collaborating PDE solvers*. It aims to solve complex physical problems with the use of modern computer science technologies. It is based on the classical relaxation idea by iteratively solving local problems and adjusting interface conditions. A software system RELAX has been implemented as a platform to support this approach. One can use this system to model complex physical objects, specify mathematical problems and test various interface relaxation schemes. It is shown that this approach is promising in both theory and practice.

2. Collaborating PDE solvers

A physical system in the real world normally consists of a large number of components. They have different shapes, obey different physical laws, and collaborate with each other by adjusting interface conditions. An automobile engine system is such a typical example. Mathematically, it corresponds to a very complicated PDE problem with various formulations for the geometry, PDE, and interface/boundary condition in many different regions. Interface locations and conditions may also vary, as in systems with moving interfaces. One can imagine the great difficulty in creating a software system to model such a complicated real problem. Therefore, one needs an effective software development mechanism which first, is applicable to a wide variety of practical problems, second, allows for the use of advanced software technologies in order to achieve high productivity and quality, and finally, is suitable for some reasonably fast numerical methods.

Notice that most of the physical systems in practical applications can be modeled as a mathematical *network*. Here a network is a directed graph consisting of a set of nodes and edges. If we represent each physical component in a system by a node, then a pair of neighboring components are linked by an edge in the graph, with the edge directions used to indicate the necessary information transmission for the interface adjustment. Each node in the network is then assigned a key to represent the local physical law for the corresponding component. For numerical relaxation one may also assign certain weights to each edge in order to provide detailed control for the interface adjustments, such as for boundary values and their jumps across the interfaces. Moving interfaces are allowed in this network specification.

Usually, individual components are simple enough so that each node corre-

sponds to a simple PDE problem with a single PDE defined on a regular geometry. There exist many standard PDE solvers that are well developed and can be applied to these local node problems. To solve the global problem, we let these local solvers collaborate with each other by invoking an interface controller. It collects boundary values from neighboring subdomains and adjusts interface conditions according to the network specifications. Therefore, the network abstraction of a physical system allows us to build a software system which is a network of collaborating PDE solvers. These networks can be very big for major applications. There are normally about 5 interfaces per subdomain. For a highly accurate weather prediction, for example, one needs 3 billion variables in a simulation with continuous input at 50 million places. This assumes a 3-D adaptive grid, otherwise the computation is much larger. Very optimistically, if one needs a new forecast every 2-3 hours, the answer is 100 gigabytes in size and requires 80 mega-giga FLOPs to compute. Such a network roughly consists of 3,000 subdomains and 15,000 interfaces. An "answer" is a data set that allows the accurate approximate solution to be displayed at any place. It is much smaller than the numerical solution from which it is derived. Another example to consider is a realistic vehicle simulation, where there are perhaps 100 million variables and many different time scales. This problem has very complex geometry and is very non-homogeneous. The answer is 20 gigabytes in size and requires about 10 tera FLOPs to compute. The network has 10,000 subdomains and 35,000 interfaces.

A software network of this type is a natural mapping of a physical system. It simulates how the real world evolves and thus normally produces a reasonable solution. It allows various advanced software technologies to be applied to create a high quality system in a very productive way. For instance, one can apply the networking technology to efficiently integrate a collection of software components into an entire system and to implement a neat and flexible system architecture for the model and its interface connections. This implies the use of the software parts technology that is the natural evolution of the software library idea with the addition of software standards. It allows software reuse for easy software update and evolution, which are extremely important in practice. The real world is so complicated and diverse that we believe there are no monolithic, universal solvers. Without software reuse, it is impractical for anyone to create on his own a large software system for a reasonably complicated application. For example, automobile manufacturers frequently change automobile models. Each change normally results in a new software system. Recreating such a system could easily take several months or years. In contrast, the execution time to perform the required computation might only be a few hours. Notice that such a physical change usually corresponds to replacing, adding, or deleting a few nodes in the network with a corresponding change in interface conditions. These can be modular manipulations on a network that do not affect the majority of the system components.

Object-oriented programming is a powerful software development methodology well suited to such a software system. In this methodology each physical component can be viewed both as a physical object and as a software object. Actions and interactions of objects are clearly defined by the network. Two basic principles of object-oriented programming are data structure abstraction and information hiding for each object. These principles are expressed here by the local solvers and the interface conditions. In addition, this network approach is naturally suitable for parallel computing as it exploits the potential parallelism in physical systems. One can handle issues like data partition, assignment, and load balancing on the physics level by the structure of a given physical system. Synchronization and communication are controlled by the network specification and restricted to interfaces of subdomains, which results in a coarse-grained computational problem. This is suitable for today's most advanced parallel supercomputers, such as the Intel's PARAGON system and the Thinking Machine's CM-5. The network approach also allows high scalability. Finally, this network approach naturally fits into the mathematical domain-decomposition framework with the overall geometry being viewed as automatically partitioned into a collection of subdomains. Note that subdomains and interfaces simply correspond to the network nodes and edges, respectively.

There have been many types of domain decomposition methods proposed over the past decade. Specific references to the literature are given in [**2**] and in other volumes of these proceedings. However, not all of them are suitable for, or directly applicable to, this network framework due to considerations from the practical and software points of view. First, the artificial subdomain overlapping introduced for mathematical convergence purposes obviously violates the basic principles of object-orientation. Each software object should correspond to a natural physical component without knowing part of data structures of other objects or exposing its local data structures to others. Second, it is not proper to apply the algebraic type of domain decomposition methods that first discretize a PDE problem on an entire domain and then partition the discrete system according to the geometric decomposition. In fact, the network framework implies the problem partition on the continuous problem level so that PDE solution techniques in different regions may be totally independent depending on local properties. One may use finite differences for one subdomain, and finite elements or even an analytic solution for another. In addition, the subdomain PDE operators are not necessarily extensible to interfaces so that global discretization is not always applicable. More importantly, the success of most of these methods relies on finding a good preconditioner for the interface Schur complement matrix, which is very difficult to do in practice for a complicated physical system. Another well-known class of methods are motivated by observing that the global solution of a Poisson equation on an entire domain is continuous on interfaces up to its first derivative, i.e., U and $\partial U/\partial n$. In order to match the continuity for both U and $\partial U/\partial n$, one starts with an initial guess for U and $\partial U/\partial n$ on interfaces, takes

them as boundary data to solve a Dirichlet or Neumann boundary value problem on subdomains, then updates interface values using the new solution data, and iterates until convergence. A common approach is the alternating Dirichlet-Neumann algorithm. This is a non-overlapping method. Generally speaking, it converges slowly and may diverge, although preconditioning techniques may be applied to improve convergence. Theoretically, it is rather difficult to understand the convergence mechanism, especially when the so-called cross points are present on interfaces. In addition, interface conditions in practical applications usually appear in more complicated forms. Nevertheless, it is important that this subdomain-iteration based approach best fits into the network framework and is thus promising from the practical point of view. The challenge is then to extend it to general interface conditions and to guarantee its fast convergence.

3. Interface Relaxation

We now present a general subdomain-iteration approach based on the classical relaxation idea.

Let Γ_{ij} be a typical interface, that is, the common boundary piece of two neighboring subdomains Ω_i and Ω_j, i.e., $\Gamma_{ij} = \partial\Omega_i \cap \partial\Omega_j$. Each subdomain obeys a physical law locally. Namely, there is a PDE L_l and function U_l defined in each Ω_l so that

$$L_l U_l = f_l \text{ in } \Omega_l \text{ for } l = i, j. \tag{3.1}$$

Notice that the interface condition on Γ_{ij} can usually be specified in the form

$$g_{ij}(U_i, U_j, \frac{\partial U_i}{\partial n}, \frac{\partial U_j}{\partial n}) = 0. \tag{3.2}$$

In general, the left-hand side of (3.2) may also involve higher order derivatives. Without loss of generality, we only consider first order derivatives. For example, for the continuity conditions of the global solution and its normal derivative, (3.2) takes the form

$$(U_i - U_j)^2 + \left(\frac{\partial U_i}{\partial n} - \frac{\partial U_j}{\partial n}\right)^2 = 0. \tag{3.3}$$

For some physical phenomena we might have different conditions to be satisfied on opposite sides of the interface so that the interface conditions need not be symmetric, i.e., we can have $g_{ij} \neq g_{ji}$. Denote by $BV(U_i, U_j) \equiv \{U_i, U_j, \frac{\partial U_i}{\partial n}, \frac{\partial U_j}{\partial n}\}|_{\Gamma_{ij}}$ the data set of boundary and derivative values of local solutions U_i and U_j on Γ_{ij}. Equation (3.2) can then be viewed as a constraint on $BV(U_i, U_j)$.

We now describe a general relaxation procedure as follows. Suppose that we have an initial guess for BV, denoted by BV^{old}, which satisfies the constraint

(3.2) for all interfaces. For each subdomain, we solve the boundary value problems with the corresponding PDEs in (3.1) and by using part of BV^{old} as the boundary data. With the newly computed local solutions, denoted by U_l^{new} for Ω_l, we evaluate boundary values to get $BV(U_i^{new}, U_j^{new})$ for all Γ_{ij}, which is denoted by BV' for brevity. In general, BV' does not satisfy the constraint (3.2) although part of BV^{old} may be preserved in BV' as the boundary data used in the local solve. The relaxation idea is to further change, i.e., to *relax*, certain components in BV' to obtain a new data set BV^{new} that (better) satisfies the constraint (3.2). This leads to solving equation (3.2) for the corresponding boundary components as the unknowns. The above two-phase procedure, consisting of local PDE solve and constraint relaxation, defines a mapping from BV^{old} to BV^{new}. Iterating this procedure until convergence, we obtain the global solution that satisfies both the local PDEs and interface constraints.

It is easy to create an object-oriented implementation of this relaxation procedure. The actions defined on a subdomain object are: (a) solving a PDE boundary value problem with the provided boundary data in BV from interfaces, and (b) evaluating boundary values of the resulting local solution. The actions defined on an interface object are: (a) collecting boundary values from neighboring subdomains, (b) checking for convergence by examining the interface constraints, (c) relaxing the constraint to update BV, and (d) invoking local solvers for neighboring subdomains.

There are various possible choices for the relaxation, depending on the boundary condition type for each subdomain solve and the way of relaxing the interface constraint. The alternating Dirichlet-Neumann approach is an example. An alternative is to apply a smoothing procedure along an interface, which blends the neighboring solutions to better satisfy the interface constraints along the interface. It is also possible to apply least squares to perform an overdetermined interface constraint relaxation rather than an exact relaxation. As usual, one may derive a multi-step type of relaxer by introducing certain relaxation parameters and taking a weighted average of previous and updated iterates.

We consider the following class of relaxers. First, we consider only *stationary relaxers*, those that use the same relaxation and PDE solution techniques at every iteration. There are non-stationary relaxers of serious interest, such as those that alternate between satisfying Neumann and Dirichlet conditions. Second, we consider only relaxers that use values and derivatives of PDE solutions along interfaces. That is, at each iteration a PDE is solved for U_l in Ω_l and the boundary values of U_l and its derivatives are the input to the relaxers. Discrete versions of the relaxers may involve such values on or near interfaces.

We define this class of relaxers precisely as follows. Let $I(l)$ be the indices of those subdomains that are neighbors of subdomain l. Let the PDE problem that is solved on Ω_l be

$$
\begin{aligned}
& L_l U_l^{new} = f_l \text{ in } \Omega_l, \\
(3.4) \qquad & B_{lj} U_l^{new} = b_{lj} \text{ on } \Gamma_{lj} \text{ for } j \in I(l), \\
& U_l^{new} \text{ satisfies the global boundary conditions on } \partial\Omega,
\end{aligned}
$$

where B_{lj} is a usual boundary condition operator and b_{lj} is defined as part of the relaxer as follows. Let $\vec{X}_{lj}^{old}$ be a vector of values which approximate U_l and its derivatives on Γ_{lj} for $j \in I(l)$. The length of the vector $\vec{X}$ is the number of derivatives of U_l used, it is normally 2 (using values and normal derivatives of U_l). Then a *relaxer* is a procedure that maps U_l^{old}, $\vec{X}_{lj}^{old}$ for $j \in I(l)$, $\vec{X}_{jl}^{old}$ for $j \in I(l)$ into b_{lj}.

Note that this definition of relaxers makes them domain-based and not interface-based. That is, the process of obtaining U_l^{new} is not easily interpreted as applying some independent set of procedures along the interfaces Γ_{lj} for $j \in I(l)$. Indeed, there might be interaction between different b_{lj} where, for example, two interfaces meet.

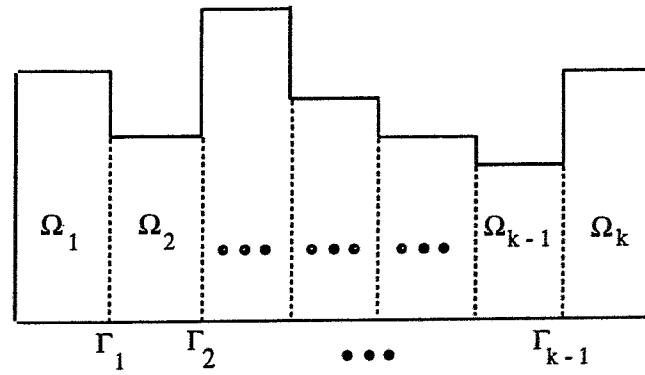

FIGURE 1. A "one dimensional" composite domain Ω.

To be more specific and for the sake of simplicity, let us assume that Ω is as in Fig. 1 and denote $\Gamma_{i,i+1}$ simply by Γ_i. Furthermore, without loss of generality, we assume that the global solution vanishes on $\partial\Omega$, and the interior interface condition is (3.3). Suppose that we impose Dirichlet condition on each Γ_{lj} for $j \in I(l)$ in (3.4). In our simplified notation, the solutions on both sides of

any interface Γ_i have the same boundary values on Γ_i, denoted by X_i, at each iteration. Equation (3.3) is then reduced to

$$\frac{\partial U_i}{\partial n} = \frac{\partial U_{i+1}}{\partial n} \text{ on } \Gamma_i \text{ for } i = 1, 2, \ldots, k-1. \tag{3.5}$$

In principle, one can apply any numerical method, such as finite differences, finite elements, or collocation to solve the local PDE problem (3.4). The corresponding discrete systems can be generally written as

$$\begin{cases} A_l U_l^{new} = f_l + P_{\Omega_l,\Gamma_{l-1}} X_{l-1}^{old} + P_{\Omega_l,\Gamma_l} X_l^{old} \text{ for } l = 1, 2, \ldots, k \\ X_0^{old} \equiv X_k^{old} \equiv 0 \end{cases} \tag{3.6}$$

where the matrices A_l, $P_{\Omega_l,\Gamma_{l-1}}$ and P_{Ω_l,Γ_l} correspond to the discretization of the PDE operator L_l in the interior and next to the boundary pieces Γ_{l-1} and Γ_l of the subdomain Ω_l. We do not distinguish the notations for a continuous function and the vector of its discrete values.

After solving (3.4) to obtain $\{U_l^{new}\}_{l=1}^k$, we want to relax the interface conditions by adjusting the solution values on interfaces to better satisfy (3.5). Let $\{X_i^{new}\}_{i=1}^{k-1}$ denote the new interface values. The normal derivatives of the relaxed solutions on both sides of Γ_i can be approximated by the finite differences involving values on Γ_i and the discretization lines next to Γ_i. Denote these neighboring grid lines by $\Gamma_i^{\pm}$. A discrete approximation to (3.5) is then

$$\frac{X_i^{new} - U_i^{new}|_{\Gamma_i^-}}{h_i^-} = \frac{U_{i+1}^{new}|_{\Gamma_i^+} - X_i^{new}}{h_i^+} \text{ for } i = 1, 2, \ldots, k-1, \tag{3.7}$$

where $h_i^{\pm}$ denote the corresponding spacings between Γ_i and $\Gamma_i^{\pm}$. Solving (3.7) for X_i^{new} we obtain

$$X_i^{new} = \alpha_i^- U_i^{new}|_{\Gamma_i^-} + \alpha_i^+ U_{i+1}^{new}|_{\Gamma_i^+} \text{ for } i = 1, 2, \ldots, k-1, \tag{3.8}$$

with $\alpha_i^- = \frac{h_i^+}{h_i^- + h_i^+}$ and $\alpha_i^+ = \frac{h_i^-}{h_i^- + h_i^+}$. As in general relaxation methods, one can further introduce some relaxation parameters or make use of U_i^{new} and U_{i+1}^{new} values on other grid lines nearby Γ_i or use previous values X_i^{old}, U_i^{old}, etc., in order to accelerate the overall convergence. For example, one can define X_i^{new} by

$$X_i^{new} = \omega X_i^{old} + (1-\omega)(\alpha_i^- U_i^{new}|_{\Gamma_i^-} + \alpha_i^+ U_{i+1}^{new}|_{\Gamma_i^+}), \tag{3.9}$$

where ω is a relaxation parameter. Recall that $X_i^{old} \equiv U_i^{new}|_{\Gamma_i} \equiv U_{i+1}^{new}|_{\Gamma_i}$ in the present case. In general, we see that a *linear relaxer* can be expressed as

$$X_i^{new} = \varphi_i(U_i^{new}, U_{i+1}^{new}) \text{ for } i = 1, 2, \ldots, k-1, \tag{3.10}$$

where φ_i is a linear combination of U_i^{new} or U_{i+1}^{new} restricted to grid lines near to the interface Γ_i with certain weights. The choice of φ_i depends on the interface condition (3.2), the approximation accuracy of the finite differences to the normal derivatives, the relaxation techniques, and so on.

We may combine solving (3.4) for $\{U_l^{new}\}_{l=1}^k$ with (3.10) to obtain the matrix representation of $\{X_i^{new}\}$ in terms of $\{X_i^{old}\}$. The convergence analysis of the relaxation process is then reduced to the standard spectral analysis of the corresponding iteration matrix. We show in [**2**] the convergence of this iteration for the class of relaxers as described above for general elliptic PDE problems and general domain decomposition with cross interface points. Furthermore, under certain model problem assumptions for a rectangular domain as decomposed in Fig. 1, an explicit expression is obtained for the spectrum of the iteration matrix so that the convergence mechanism is fully understood. In addition, the optimal relaxation parameters are also determined. Extensive numerical experiments are reported in [**2**] to support the theoretical convergence analysis.

4. RELAX problem solving environment

In this section, we describe a problem solving environment RELAX [**1**] that is implemented as a platform to support the collaborating PDE solvers approach. RELAX provides both a computational and user interface environment. The computational environment coordinates teams of single-domain PDE solvers, which collaboratively solve mathematical systems called composite PDEs that model complex physical systems. The user interface environment coordinates multiple interactive user interface components, called editors, which display or alter any feature of a composite PDE problem. Editors may be both text-oriented (e.g., equation editors) and graphics-oriented (e.g., solution plotters).

RELAX is implemented using object-oriented programming technology. The system architecture is based upon a set of inter-communication software components. Editors and single-domain PDE solvers are examples of RELAX components – these particular ones are externally supplied (perhaps from libraries or other software systems). RELAX provides a message-passing mechanism for supporting the inter-component communication. It is capable of integrating existing scientific software for PDEs into a broader problem solving environment. It also has the capability of using pre-existing display and interaction components to form a flexible, dynamic user interface.

We briefly outline the function of each type of component of the RELAX architecture and refer to [**1**] for more details:

- *Primitive Objects*: These are externally supplied components which model and solve primitive PDE problems. Primitive objects are responsible

for all aspects of solving a single-domain PDE problem, including the generation of numerical meshes, discretization of the PDE, and solving systems of equations.

- *Editors*: These are externally supplied components which provide an interface between the user and some feature of the system. The editor component is responsible for the complete presentation of the user interface, including all communication with the window system and/or graphics package.
- *Message Dispatcher*: This is a system supplied component which handles all transmission of messages within the system. The message dispatcher also registers targets and can assist editors in locating targets.
- *Composite Problem Platform*: This is a system component which maintains the data structures defining a composite PDE problem. For example, the composite problem platform stores topological information about which primitive objects share geometric interfaces, as well as equations defining the interface conditions along those interfaces. Additionally, the composite problem platform maintains data structures defining a global solution iteration, and is capable of executing such iterations. Finally, the composite problem platform is capable of defining composite PDE problems hierarchically.
- *Object Support Platform*: This is a system component responsible for integrating primitive objects into the system. The object support platform provides the attachment point for primitive objects – recall that they are external components and must be dynamically attached to the running system. The object support platform relays messages between primitive objects and the message dispatcher. Another feature of the object support platform is a *virtual object* mechanism which allows various primitive objects to filter the messages intended for other primitive objects.
- *Editor Support Platform*: This is a system component which provides an attachment point and communication interface for editors. The editor support platform relays messages to and from editors, and is also capable of parameterizing and controlling the message flow, for example, by copying and buffering messages.

With this environment, one can easily describe primitive PDE problems and interface conditions to compose a complex mathematical system, specify local solvers and relaxers to define an iterative procedure, and display the computed solution in various ways. As an application example, we use the RELAX system to solve a physical heat flow problem as shown in Fig. 2. The complex object consists of seven simple subdomains with nine interfaces. The radiation conditions allow heat to leave on part of the boundaries while the temperature U is zero on all the other boundaries. The mounting regions have heat dissipated.

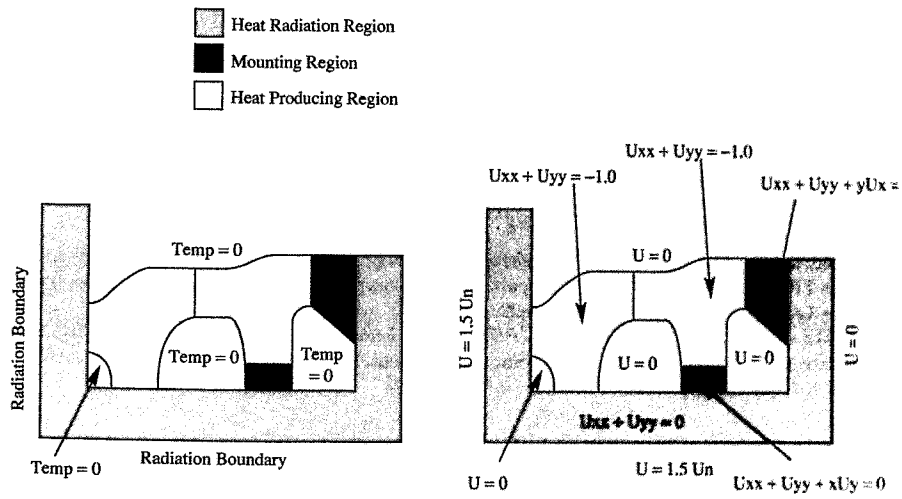

FIGURE 2. A heat flow problem for a complex domain along with the physical and mathematical descriptions.

The interface conditions are continuity of temperature U and its derivative. An approximate solution of 3-digit accuracy is obtained after 15 iterations, where the initial guess is zero and the relaxer used is as described in Section 3 with the relaxation parameter $\omega = 0$.

5. Conclusions

We examine in this paper various aspects of the real world simulation with the emphasis on software productivity and quality. Application of modern software technologies and the impact on numerical PDE methods are considered. We present a general approach for modeling complex physical systems by a network of collaborating PDE solvers. The related methodologies include *networks of collaborating software modules, object-oriented programming* and *domain decomposition*. They lead to a suitable subdomain-iteration based procedure with interface relaxation. Various types of relaxers are discussed. A software system RELAX is described which is implemented as a platform to test various relaxers and to solve complex problems using the network approach. Both theory and practice show that this is a promising approach for solving complicated problems

on modern computer environments.

Acknowledgement. The RELAX environment is implemented by Dr. Scott McFaddin. We would like to thank him very much for providing the implementation details. We also thank Professor David Keyes for his valuable comments and suggestions.

REFERENCES

1. S. McFaddin and J. R. Rice, *Architecture of the RELAX problem solving environment*, CSD-TR-92-081 and CER-92-37, Department of Computer Sciences, Purdue University, West Lafayette, IN47907, October, 1992.
2. M. Mu and J. R. Rice, *Collaborating PDE solvers with interface relaxation*, CSD-TR-93-024 and CER-93-13, Department of Computer Sciences, Purdue University, West Lafayette, IN47907, April, 1993.

DEPARTMENT OF MATHEMATICS, THE HONG KONG UNIVERSITY OF SCIENCE & TECHNOLOGY, CLEAR WATER BAY, KOWLOON, HONG KONG
E-mail address: mamu@usthk.bitnet

COMPUTER SCIENCES DEPARTMENT, PURDUE UNIVERSITY, WEST LAFAYETTE, IN 47907 U.S.A.
E-mail address: rice@cs.purdue.edu