1997

Agent Based Systems to Support Multi- disciplinary Probelm Solving Environments

Anupam Joshi

Naren Ramakrishnan

Tzvetan Drashansky

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

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

See next page for additional authors

Report Number:
97-031

AGENT BASED SYSTEMS TO SUPPORT MULTIDISCIPLINARY PROBLEM SOLVING ENVIRONMENTS

Anupam Joshi, Naren Ramakrishnan,
Tzvetan Drashansky, Elias N. Houstis
John R. Rice, Sanjiva Weerawarana
& L.H. Tsoukalas

CSD-TR 97-031
June 1997

Agent Based Systems to Support Multidisciplinary Problem Solving Environments

Anupam Joshi
Department of Computer Engineering & Computer Science
University of Missouri, Columbia, MO 65211
email: joshi@trinetra.cecs.missouri.edu

Naren Ramakrishnan, Tzvetan Drashansky, Elias N. Houstis,
John R. Rice and Sanjiva Weerawarana
Department of Computer Sciences, Purdue University
West Lafayette, IN 47907-1398
email: {naren,ttd,enh,jrr,saw}@cs.purdue.edu

L.H. Tsoukalas
School of Nuclear Engineering
Purdue University, West Lafayette, IN 47907
email: tsoukala@ecn.purdue.edu

Abstract

The predicted growth of computational power and network bandwidth suggests that computational modeling and experimentation will be one of the main tools in big and small science. In this scenario, computational modeling will shift from the current single physical component design to the design of a whole physical system with a large number of components that have different shapes, obey different physical laws and manufacturing constraints, and interact with each other through geometric and physical interfaces. We refer to these multi–experiment based systems as multidisciplinary applications. The realization of the above scenario will require the development of new algorithmic strategies and software for managing the complexity and harvesting the power of the expected HPCC resources; it will require problem solving environment (PSE) technology to support programming-in-the-large and reduce the overhead of HPCC computing. In this paper, we identify the framework for the numerical simulation of multidisciplinary applications and develop the enabling methodologies needed to support and realize this framework in specific applications. The software implementation of this framework is called a multidisciplinary problem solving environment (MPSE). It is assumed that its elements are discipline-specific PSEs. Our MPSE design objective is to allow a) the "natural" specification of multidisciplinary applications and their simulation with interacting PSEs through mathematical and software interfaces across networks of heterogeneous computational resources and b) the automatic selection of

software/hardware resources needed to support the simulation.

1. INTRODUCTION

The new economic realities require the rapid prototyping of manufactured artifacts and rapid solutions to problems with numerous interrelated elements. This, in turn, requires the fast, accurate simulation of physical processes and design optimization using knowledge and computational models from multiple disciplines in science and engineering. These models have been variously described as multi-disciplinary or multiphysics models. Thus, the realization of rapid multidisciplinary problem solving or prototyping is the new grand challenge for computational science and engineering (CSE) [9; 39]. We refer to a software realization of multidisciplinary prototyping throughout as a Multidisciplinary Problem Solving Environment (MPSE).

The aim of this paper is to present a mathematical and software framework for MPSE, including the enabling methodologies and their realization on HPCC platforms that resemble the future National Information Infrastructure (NII). This framework is adaptable and intelligent with respect to end-users and hardware platforms. It uses collaborating software systems and agent based techniques to build demonstration MPSEs for physical modeling. It allows the wholesale reuse of scientific software and provides a natural approach to parallel and distributed problem solving. Finally, it is driven by an intelligent interface providing a high level abstraction of the complexity of the underlying computations and hardware platforms. This functionality is similar to a PSE [41] in that it includes advanced solution methods, automatic or semiautomatic selection of solution methods, and ways to easily incorporate novel solution methods. Moreover, it uses the language of the target class of problems and provide a "natural" interface, so users can use it without specialized knowledge of the underlying computer hardware or software problems [7].

The evolution of the Internet into the global information infrastructure (GII), and the concomitant growth of computational power and network bandwidth suggests that computational modeling and experimentation will continue to grow in importance as a tool for big and small science. Networked scientific computing (NSC) seems to be the next step in the evolution of high performance computing which will enable us to attack mltidisciplinary problems. It allows us to use the high performance communication infrastructure (vBNS, Internet II etc.) to view heterogeneous networked hardware (including resources such as the proposed terraflops machines) and software (e.g., specialized solvers, performance measuring systems) resources as a single "meta computer"[12]. NSC enables scientists to begin to address the class of complex problems that are envisaged in the Accelerated Strategic Computing Initiative (ASCI) from DOE. In this type of problem (e.g., lifecycle simulation of very complex devices), the design process operates at the scale of the whole physical system with a large number of components that have different shapes, obey different physical laws and manufacturing constraints, and interact with each other via geometric and physical interfaces through time. Thus, in this work we have assumed the network computing paradigm to design and implement the proposed MPSE framework.

The research issues addressed here concern the needed infrastructure that allows multidisciplinary applications to use resources and services from many "servers"

spread across the network. The user operates within an environment supporting an abstraction, via an MPSE, of the underlying networked infrastructure as a single meta-computer, and details such as locating the appropriate software and hardware resources for the present problem, changes/updates fixes to the software components, harnessing the network computing power, etc., are handled at the system level with minimal user involvement.

We start by motivating and describing an agent–based approach for designing and building MPSEs together with the problems of resource and solution methodology selection. We then introduce the *SciAgents* system, which provides the solver and mediator agents for our MPSEs. Next, we describe PYTHIA, a multiagent advisory system containing (in a distributed manner) the total knowledge corpus. We show how these various agents can interact with each other to automate the process of solving multiphysics problems. Finally, we consider in detail two case studies using current prototypes that show the applicability and the potential of our MPSE concept and that demonstrate our approach for its implementation.

2. AGENT BASED COMPUTING PARADIGM FOR MPSES

Most physical systems in the real world normally consist of a large number of components where the physical behavior of each component is modeled by a differential equation system with various formulations for the geometry, partial differential equation (PDE), ordinary differential equation (ODE), interface/boundary/linkage and constraint conditions in many different geometric regions. These systems can be modeled as a mathematical network whose nodes represent the physical components in a system or artifact. Each node has a mathematical model of the physics of the component it represents and a solver agent for its analysis. Individual components are chosen so that each node corresponds to a simple PDE or ODE problem defined on a domain with simple geometry.

For the simulation of such systems, one needs an MPSE mathematical/software framework which, (1) is applicable to a wide variety of practical problems, (2) allows for software reuse in order to achieve lower costs and high quality, (3) is suitable for some reasonably fast numerical methods, and (4) can be supported by a distributed or network computing paradigm.

We propose to use the *multi–agent computing framework* to provide run-time support for MPSEs where we replace the multiphysics problem by a set of simple(r) simulation problems on simple geometries which must be solved simultaneously while satisfying a set of interface conditions. These simpler problems may reflect the underlying structure/geometry/physics of the system to be simulated, or may be artifically created by techniques such as domain decomposition. Given a collection of *solvers* for these smaller problems on simple geometries, we view each of them as a *solver agent*, and by introducing *mediator agents* between them, we create a network of collaborating solvers. Each solver deals with one of the subproblems defined earlier. The original multiphysics problem is solved when one has all the equations satisfied on the individual components and these solutions "match properly" on the interfaces between the components. This latter part is the responsibility of the mediator agents. The term "match properly" is defined by the physics if the interface is where the physics changes. For heat flow, for example, this means that temperature is the same on both sides of the interface and that the amount

of heat flowing into one component is the same as the amount flowing out of the other. If the interface is artificial (introduced to make the geometry simple or the work smaller) then "match properly" is defined mathematically and means that the solutions join smoothly (have continuous values and derivatives).

Many agent-based systems have been developed[10; 34; 35; 38; 44] which demonstrate the power of the agent-oriented paradigm. It provides modularity and flexibility, so it is easy to dynamically add or remove agents, to move agents around the computing network, and to organize the user interface. An agent based architecture provides a natural method of decomposing large tasks into self-contained modules, or conversely, of building a system to solve complex problems by a collection of agents, each of which is responsible for small part of the task. Agent-based systems can minimize centralized control.

The agent-based paradigm is useful in scientific computing to handle complex mathematical models in a natural and direct way. It allows *distributed problem solving* [26] which is distinct from merely using distributed computing. The expected behavior of the simple model solvers, computing locally and interacting with the neighboring solvers, naturally take on the behavior of a *local problem solver* agent. The task of mediating interface conditions between adjacent subproblems is given to *mediator* agents and their ability to autonomously pursue their goals can resolve the problems during the solution process without user intervention and converge to the global solution.

Several researchers have addressed the issue of coordinating multi-agent systems. For instance Smith and Davis [36] propose two forms of multi-agent cooperation, task sharing and result sharing. Task sharing essentially involves creating subtasks, and then farming them off to other agents. Result sharing is more data directed. Different agents are solving different tasks, and keep on exchanging partial results to cooperate. They also proposed using "contract nets" to distribute tasks. Wesson et al., showed[43] how many intelligent sensor devices could pool their knowledge to obtain an accurate overall assessment of a situation. The specific task presented in their work involves detecting moving entities, where each "sensor agent" sees only a part of the environment. They reported results using both an hierarchical organization, as well as an "anarchic committee" organization, and found that the latter was as good as, and sometimes better than the former. Cammarata et al. [2] present strategies for cooperation by groups of agents involved in distributed problem solving, and infer a set of requirements on information distribution and organizational policies. They point out that different agents may have different capabilities, limited knowledge and resources, and thus differing appropriateness in solving the problem at hand. Lesser et al. [22] describes the FA/C (functionally accurate, cooperative) architecture in which agents exchange partial and tentative results in order to converge to a solution. Joshi [11] presents a learning technique which enhances the effectiveness of such coordination. It combines neuro-fuzzy techniques [37] with an epistemic utility criterion.

3. THE RESOURCE SELECTION PARADIGM FOR MPSES

In an MPSE environment, the solver and mediator agents form a potentially large pool of computational objects spread across the network. Moreover, there are many possible choices for their instantiation, for example the past few decades has seen

a huge amount of sophisticated code being developed to solve specific, homogeneous problems. Mediators today are almost nonexistant and a large number will have to be created to allow disparate solvers to interact. Clearly, expecting the user to be aware of all the potentially useful solvers on the network is not realistic. Nor is a user likely to know all the hardware choices available to solve the problem. This problem is an obvious generalization of the *algorithm selection* problem formulated by Rice [32], we call it the *resource selection* problem in the context of MPSEs. We propose the use of *advisory agents* that accept a problem definition and some performance/success criteria from the user, and that then suggest software components and hardware resources that can be deployed to solve this problem. This is very similar to the idea of *recommender systems* that is being proposed for harnessing distributed information resources. While the recommender problem has been identified for networked information resources and initial research done[31], the resource selection problem remains largely ignored for harnessing networked computational resources. Note that the problem is different from invoking a known method remotely on some object, a problem where many distributed object oriented techniques are being developed and proposed. To appreciate the need for advisory agents, consider the present day approximation to "networked" scientific computing. Several software libraries for scientific computing are available, such as Netlib, Lapack/ScaLapack, etc. There are even some attempts to make such systems accessible over the web, such as Web //ELLPACK [from Purdue, http://pellpack.cs.purdue.edu/] and NetSolve [from UTK/ORNL, http://www.cs.utk.edu/netsolve/]. The GAMS [1] system helps users to identify and locate the right class of software for their problem. However, the user has to select the specific routine most appropriate for the given problem, download the software along with its installation and use instructions, install the software, compile (and possibly port) it, and then learn how to invoke it appropriately. Clearly this is a non-trivial task even for a single piece of software, and it can be enormously complex when multiple software components need to be used. Using networked resources today can be viewed as the modern day equivalent of programming ENIAC, which required direct manipulation of connecting wires. Systems are needed to abstract away the detail of the underlying networked system from the user and allow interaction with this system in the application domain. This is where MPSEs with inherent "intelligence" come in. We posit that multiagent systems, consisting of broad class of solver, mediator, and advisory agents can be used to create MPSEs with the desired characteristics.

We will often use PDEs as our example domain in which to describe the ideas of solver, mediator and advisory agents. We will also use this domain for our prototype software to validate our ideas. The numerical solution of PDEs depends on many factors including the nature of the operator, the mathematical behavior of its coefficients and its exact solution, the type of boundary and initial conditions, and the geometry of the space domains of definition. Most numerical solvers for PDEs normally require a number of parameters from the user, in order to obtain a solution within a specified error level while satisfying certain resource (e.g., memory and time) constraints. The problem of selecting a solver and its parameters for a given PDE problem to satisfy the user's computational objectives is difficult and of great importance. The user must also select a machine from among the many avail-

able on the network, including parallel machines. Depending on the mathematical characteristics of the PDE models, there are "thousands" of numerical methods to apply, since very often there are several choices of parameters or methods at each of the several phases of the solution. It is unrealistic to expect that engineers and scientists will or should have the deep expertise to make "intelligent" combinations of selections of methods, their parameters, and computational resources that will satisfy their objectives.

The PYTHIA [42] project at Purdue has focussed on creating a knowledge based system that selects scientific algorithms to achieve desired tasks in computing. It determines a near-optimal strategy (i.e., a solution method and its parameters) for solving a given problem within user specified resource (i.e., limits on execution time and memory usage) and accuracy requirements (i.e., level of error). While the ideas behind PYTHIA are quite general, our current implementations operate in conjunction with systems that solve (elliptic) partial differential equations (PDEs), such as the ELLPACK and //ELLPACK PSEs developed at Purdue. The methodology of PYTHIA is to gather performance information about PDE solvers on standardized test problems and use this data plus feature information about PDE problems to determine good algorithms to solve the PDEs. The efficacy of this approach is dependent on the breadth and diversity of the method and problem sets used to create the performance evaluation information.

We now briefly describe some attempts at developing intelligent systems for assisting in various aspects of the PDE solution process. In [33], Rice describes an abstract model for the algorithm selection problem, which is the problem of determining a selection (or mapping) from the problem feature space to the algorithm space. Using this abstract model Rice describes an experimental methodology for applying this abstract model in the performance evaluation of numerical software. In [24], Moore et al. describe a strategy for the automatic solution of PDEs at a different level. They are concerned with the problem of determining (automatically) a geometry discretization that leads to a solution guaranteed to be within a prescribed accuracy. In [4; 5], Dyksen and Gritter describe a rule based expert system for selecting solution methods for elliptic PDE problems based on problem characteristics. This work differs significantly from our approach, which uses only performance data as the basis of the algorithm selection methodology. While these rules help some, we argue that using problem characteristics solely is not sufficient because the performance of a solver depends on quantities which cannot be measured symbolically and a priori. Further, software performance depends not only on the algorithms used, but on their implementations as well. In [19], Kamel et. al. describe an expert system called ODEXPERT for selecting numerical solvers for initial value ordinary differential equation (ODE) systems. ODEXPERT uses textual parsing to determine some properties of the ODEs and performs some automatic tests (e.g., a stiffness test) to determine others. Once all the properties are known, it uses its knowledge base information about available ODE solution methods (represented as a set of rules) to recommend a certain method. After a method has been determined, it selects a particular implementation of that method based on other criteria and then generates source code (Fortran) for the user. If necessary, symbolic differentiation is used to generate code for the Jacobian as well. Leake has recently begun some work in the area of using traditional case based reasoning

systems to select appropriate methods for solving sparse linear systems [20]. Our group has also been actively involved in using several techniques, such as neural nets, neuro-fuzzy systems and Bayesian nets ([42; 17; 28; 15; 13; 14]) to address related issues of classifying PDE problems based on their performance characteristics, and then using this classification to predict an appropriate solution method for new problems. We have also formulated the algorithm selection problem as conducting knowledge discovery in domains of computational science [29; 30]. This work shows that such data mining approaches can be used to form relational descriptions of PDE objects which lead to more powerful schemas for resource selection (in terms of both representation and prediction).

4. SCIAGENTS SYSTEM

In this section, we describe in detail the *SciAgents* software architecture, and explain how to use it for complex PDE-based models from MPSEs.

4.1 Software Architecture and User Abstraction

As an application of our MPSE approach, SciAgents employs two major types of computing agents – *solvers* and *mediators*. It interacts with the *recommender* agents as described later. The solver is considered a "black box" by the other agents and it interacts with them using an interagent language for the specific problem. This feature allows all computational decisions for solving one individual subproblem to be taken independently from the decisions in any other subproblem – a major difference from the traditional approaches to multidisciplinary simulations. Each mediator agent is responsible for adjusting an interface between two neighboring subproblems. Since the interface between any two subproblems might be complex in itself, there may be more than one mediator assigned to adjust it, each of them operating on separate piece of the whole interface. Thus the mediators control the data exchange between the solvers working on neighboring subproblems by applying mediating formulas and algorithms to the data coming from and going to the solvers. Different mediators may apply different mediating formulas and algorithms depending on the physical nature of their interfaces. The mediators are also responsible for enforcing global solution strategies and for recognizing (locally) that some goal (like "end of computations") has been achieved.

The solvers and mediators form a network of agents to solve the given global problem. A schematic view of the functional architecture of a SciAgents MPSE containing an example network is given in Figure 1. The computations (and the major data exchange) are concentrated in the network of solver (PSE) and mediator agents. The solver agents communicate with the recommender agents (as consultants) through queries to obtain "advice" on computation parameters. The user interacts with the system through the global and local user interfaces which send queries and receive replies from the various agents. The intelligent controller and the MPSE constructor can be integrated into a single "agent" which controls the global state of the computations and instantiates, queries, and manages (if necessary) the other agents.

We now describe how the user builds ("programs") this network. The agent framework provides a natural abstraction to the user in the problem domain and hides the details of the actual algorithms and software involved in the problem

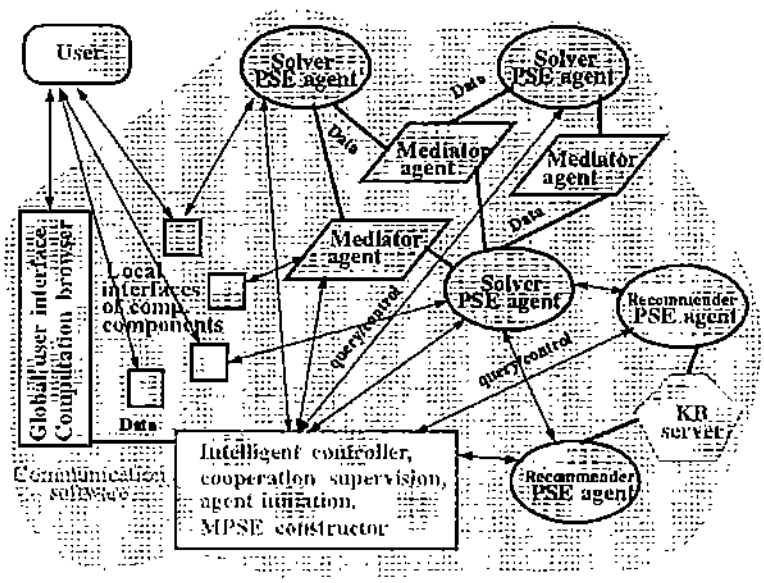

Fig. 1: Functional architecture of a SciAgents solver for an MPSE. The computations (and the major data exchange) are concentrated in the network of solver (PSE) and mediator agents. The solver agents communicate with the recommender ones through queries to obtain "advice" on computation parameters. The user interacts with the system through the global and local user interfaces which send queries and receive replies from the various agents.

solving. The user firsts breaks down the geometry of the composite domain into simple subdomains with simple models to define the subproblems for each subdomain. Then the physical conditions along each interface between the subdomains are identified. All this is done in the terms of the user's problem domain. The user is provided with an *MPSE constructor* (*agent instantiator*) — a process which displays information about the templates and creates active agents of both kinds, capable of computing. Initially, only *templates of agents* — structures that contain information about solver and mediator agents and how to *instantiate* them, are available. Then the user constructs the proper network of computing agents by simply *instantiating* various agents. The user selects solvers that are capable of solving the corresponding subproblems and mediators that are capable of mediating the physical conditions along the specific interfaces, and assigns subproblems and interfaces, respectively, to each of them. The user interacts with the system using a visual programming approach which has proved useful in allowing the non-experts to "program" by manipulating images and objects from their problem domain. In our case, a visual environment is useful for the MPSE constructor, or when the user wants to request some action or data.

Once an agent is instantiated, it takes over the communication with the user and with its environment (the other agents) and tries to acquire all necessary information for its task. Each PSE (solver agent) retains its own interface and can interact with the user. It is convenient to think of the user as another agent in these interactions. The user defines each subproblem independently, interacting with the corresponding solver agent through its user interface and similarly interacting with the mediators to specify the physical conditions holding along the various interfaces.

The agents *actively* exchange partial solutions and data with other agents without outside control and management. In other words, each solver agent can request the necessary domain and problem related data from the user and decide what to do with it (should it, for instance, start the computations or should it wait for other agents to contact it?). After each mediator agent has been supplied with the connectivity and mediating data by the user, it contacts the corresponding solver agents and requests the information it needs. This information includes the geometry of the interface, the functional capabilities of the solvers with respect to providing the necessary data for adjusting the interface, visualization capabilities, etc. All this is done without user involvement. By instantiating the individual agents (concentrating on the individual subdomains and interfaces) the user builds the highly interconnected and interoperable network that is tailored to solve the particular multiphysics problem, by *cooperation* between individual agents.

The user's high-level view of the MPSE architecture is shown in Figure 2. The global communication medium used by all entities in the MPSE is called a *software bus* [40]. The MPSE constructor communicates with the user through the user interface builder and uses the software bus to communicate with the templates in order to instantiate various agents. Agents communicate with each other through the software bus and have their own local user interfaces to interact with the user. The order of instantiating the agents is not important. If a solver agent is instantiated and it does not have all data it needs to compute a local solution (i.e., a mediator agent is missing), then it suspends the computations and waits for some relaxer agent to contact it and to provide the missing values (this is also a way to

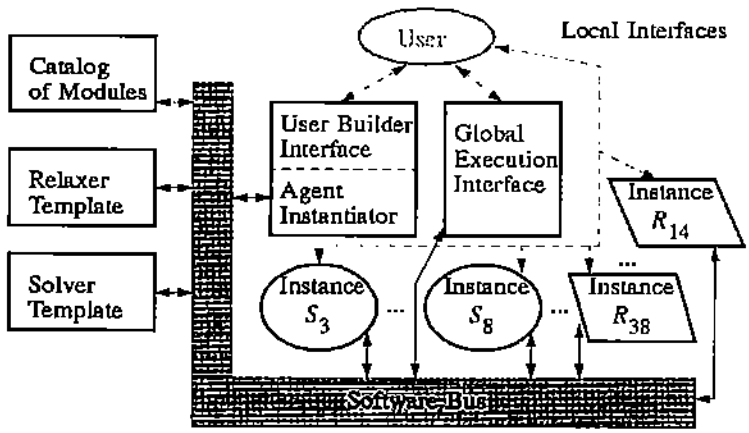

Fig. 2: Software architecture of an MPSE: the user's abstraction. The user initially interacts with the User Interface Builder to define the global composite problem. Later the interaction is with the Global Execution Interface to monitor and control the solution of the problem. Direct interaction with individual solvers and mediators is also possible. The agents communicate with each other using the *software bus*.

"naturally" control the solution process). If a mediator agent is instantiated and a solver agent on either side of its interface is missing, then it suspends its computations and waits for the solver agents with the necessary characteristics (the right subdomain assigned) to appear. This built in synchronization is, we believe, an important advantage of our architecture. It results from each agent adapting to its environment. We go into more detail about inter agent communication later.

Since agent instantiation happens one agent at a time, the data which the user has to provide (domain, interface, problem definition, etc.) is strictly local, and the agents collaborate in building the computing network. The user actually does not even need to know the global model. We can easily imagine a situation when the global problem is very large. Different specialists may only model parts of it. In such a situation, a user may instantiate a few agents and leave the instantiating of the rest of the cooperating agents to colleagues. Naturally, some care has to be taken in order to instantiate all necessary agents for the global solution and not to define contradictory interface conditions or mediation schemes along the "borders" between different users.

The collection of agent interfaces that a user interacts with is the only software the user actually needs to run locally in order to solve her/his problem. Therefore, this architecture abstracts successfully from the user the location of the main computations (the location of the solvers and the mediators) and allows for great flexibility in this direction, including running the MPSE over the Internet and distributing the agents over a WAN.

This user view of the SciAgents architecture is too abstract for an actual implementation where one has to design the internal architecture of each agent and the detailed communication among the agents. We refer the reader to [3] for these important details. We only mention here that the agent architecture utilizes the locality of the communication patterns described before and the fact that whenever

a mediator is active (computing), the corresponding solvers are idle and vice versa. Also, the asynchronicity of the communication and the need of implementing the "pro-active" feature of the agents prompt us to employ many active threads in a single agent (multithreading).

Coordination of the Solution Process We discuss now some important aspects of the cooperation between the agents during the solution process. There are well-defined global mathematical conditions for terminating the computations, for example, reaching a specified accuracy, or impossibility to achieve convergence. In most cases, these global conditions can be "localized" either explicitly or implicitly. For instance, the user may require different accuracy for different subdomains and the computations may be suspended locally if local convergence is achieved.

The local computations are governed by the mediators (the solvers simply solve the PDE problems). The mediator agents collect the errors after each iteration and, when the desired accuracy is obtained, *locally* suspend the computations and report the fact to the intelligent controller. The suspension is done by issuing an instruction to the solvers on both sides of this interface to use the boundary conditions for the interface from the previous iteration in any successive iterations they may perform (the other interfaces of the two subdomains might still not have converged). The solvers continue to report the required data to the submediators and the submediators continue to check whether the local interface conditions are satisfied with the required accuracy. If a solver receives instructions to use the old iteration boundary conditions for all its interfaces, then it stops the iterations. The iterations may be restarted if the interface conditions handled by a given mediator agent are no longer accurately satisfied (even though they once were). In this case, the mediator issues instructions to the two solvers on both sides of its interface to resume solving with new boundary conditions. If the maximum number of iterations is reached, the mediator reports failure to the intelligent controller and suspends the computations. The only global control exercised by the intelligent controller is to terminate all agents in case all mediators report local convergence or one of them reports a failure. The messages used in the interagent communication are given in full detail in [18], we provide a small example in the next section.

The above scheme provides a robust mechanism for cooperation among the computing agents. Using *only* local knowledge, they perform only local computations and communicate only with "neighboring" agents. They *cooperate* in solving a global, complex problem, and none of them exercises centralized control over the computations. The global solution "emerges" in a well-defined mathematical way from the local computations as a result of intelligent decision making done locally and independently by the mediator agents. The agents may change their goals dynamically according to the local status of the solution process – switching between observing results and computing new data.

Other global control policies can be imposed by the user if desired – the system architecture allows this to be done easily by distributing the control policy to all agents involved. Such global policies include continuing the iterations until the all interface conditions are satisfied, and recomputing the solutions for all subdomains if the user changes something (conditions, method, etc.) for any domain.

Software Reuse and Evolution One of the major goals of this MPSE approach is to design a system that allows for low-cost and less time-consuming methods

of building the software to simulate a complex mathematical model of physical processes. This goal cannot be accomplished if the existing rich variety of problem solving software for scientific computing is not used. More precisely, there are many well–tested, powerful, and popular PSEs for solving problems very similar or identical to the subproblems that appear when breaking the global model into "simple" subproblems defined on a single subdomain. These PSEs could easily and accurately solve such a "simple" subproblem. It is, therefore, natural to reuse such PSEs as solver agents. However, our architecture requires the solvers to behave like agents (e.g., understand agent languages, use them to communicate data to other agents), something the existing PSEs in scientific computing do not do.

Our solution to this problem is to provide an *agent wrapper* for PSEs and other software modules, which takes care of the interaction with the other agents and with the other aspects of emulating agent behavior. The wrapper encapsulates the original PSE and is responsible for running it and for the necessary interpretation of parameters and results. This is not simply a "preprocessor" that prepares the PSE's input and a "postprocessor" that interprets the results, since the mediation between subproblems may require communicating intermediate results to the mediators and/or accepting some additional data from them. Designing the wrapper is sometimes complicated by the "closed" nature of extant PSEs — their original design is not flexible or "open" enough to allow access to various parts of the code and the processed data. However, it is our opinion that the PSE developers can design and build such a wrapper for a very small fraction of the time and the cost of designing and building entire new PSE or custom software for every new problem. The wrapper, once written, will enable the reuse of this PSE as a solver agent in different MPSEs, thus amortizing the cost further. As part of the specifications of the wrapper the developers have to consider the mediation schemes involving submodels within the power of the PSE. An additional task is to evaluate the PSE's user interface — since the user defines the local submodel through it, it is important that the interface facilitates the problem definition in user's terms well enough. Our experience with //ELLPACK was that building a wrapper for a substantial (more than a million lines of code), diverse, and non-homogeneous PDE solver could be done efficiently, it required about a thousand lines of code.

5. PYTHIA SYSTEM

We see that the role played by the recommender agents is paramount for the effectiveness of *SciAgents*. When queried by the solver agents, they provide consulting advice on a suitable scheme (and associated computation parameters) to solve a given problem so as to achieve desired performance criteria. An example PDE problem is given in Fig. 3. A prescribed solution strategy could be *"Use the 5–point star algorithm with a* 200×200 *grid on an nCube/2 with 16 processors. Confidence: 0.90"* (Notice that a recommender agent provides a level of confidence in the selected strategy). In essence, the recommender agents serve as knowledge engines that provide domain–specific inference for PDE problems. If any particular recommender agent lacks the expertise to provide this recommendation, it will collaborate with other recommender agents and select the best answer. These agents can also be made to interact directly with the user, via the agent instantiator. Thus PYTHIA is a collaborative, multi–agent system that uses collective knowledge to prescribe a

PROBLEM #28	$(w u_x)_x + (w u_y)_y = 1,$ where $w = \begin{cases} \alpha, & \text{if } 0 \le x, y \le 1 \\ 1, & \text{otherwise.} \end{cases}$
DOMAIN	$[-1,1] \times [-1,1]$
BC	$u = 0$
TRUE	unknown
OPERATOR	Self–adjoint, discontinuous coefficients
RIGHT SIDE	Constant
BOUNDARY CONDITIONS	Dirichlet, homogeneous
SOLUTION	Approximate solutions given for $\alpha = 1$, 10, 100. Strong wave fronts for $\alpha \gg 1$.
PARAMETER	α adjusts size of discontinuity in operator coefficients which introduces large, sharp jumps in solution.

Fig. 3. A problem from the PDE population.

strategy to solve a given problem in scientific computation. The agents themselves are referred to as PYTHIA agents and are implemented by a combination of C language routines, shell scripts and systems such as CLIPS (the C Language Integrated Production System) [8]. The agents communicate using the Knowledge Query and Manipulation Language (KQML) [6], using protocol defined performatives. All PYTHIA agents understand and utilize a private language (PYTHIA–Talk) that describes the meaning (content) of the KQML performatives. This design allows the seamless integration of the recommender agents into the MPSE architecture.

A PYTHIA agent relies heavily on the problem set used in its performance evalution knowledge base so the effectiveness of a recommender agent depends on its 'experience'. For example, one agent's expertise might come from its test base of computational fluid dynamics PDE solvers and problems while a second agent's expertise might be based on heat conduction problems. Our mulit–agent methodology recognizes that there are many, many different kinds of PDE problems and any single recommender agent is likely to be limited by its knowledge base. Thus, the approach taken is to create several different PYTHIA agents, each of which has information about some class(es) of PDE problems and can predict an appropriate solver for a given PDE of those classes. If a PYTHIA agent discovers that it does not have enough confidence in the prediction it is making, it could query all other PYTHIA agents, obtain answers from all of them and use this information to decide which one is "most reasonable". This could entail a huge amount of network traffic and inordinate delays. A better approach is to use the information obtained by the initial broadcast type of queries to infer the most experienced PYTHIA agent for the problem at hand.

Then the research issues are:

(1) Given more than one applicable agent, how does one determine the best agent for a given PDE problem? In other words, what is the mapping from a given problem to a best PYTHIA agent?

(2) Can the notion of best agent be assigned automatically or does it require user input?

(3) How does one learn and adapt to the changing dynamics of the scenario? Agents may spring up into existence, some may go extinct, their abilities may change

dynamically (more problems may be added into their knowledge bases), etc. How do we learn the mapping in this case and update it suitably?

We use a quantitative measure of reasonableness [11; 27], to automatically generate exemplars to learn the mapping from PDE problems to PYTHIA agents. This is needed because the computational scientist cannot be expected to have such information in this dynamic scenario. For example, in response to a query from the user about a particular PDE problem, each PYTHIA agent might suggest a different method with varying levels of confidence in the recommended strategy. Moreover, each of these agents might have different levels of expertise (such as the kind of PDEs it knows about) and different 'training' history. The user, thus, cannot be expected to know which one of them is most suitable for the problem if all these responses are supplied. Our measure of reasonableness allows the automatic 'ranking' of the PYTHIA agents for a particular problem (class). This measure combines two factors, the probability of an agent's prediction q being true, and the predictor's utility. Specifically, the reasonableness of a proposition is defined as follows [21]:

$$r(q) = p(q)U_t(q) + p(\sim q)U_f(q),$$

where $U_t(q)$ denotes the positive utility of accepting q if it is true, $U_f(q)$ denotes the negative utility of accepting q if it is false and $p(q)$ be the probability that q is true.

In the case of PYTHIA, each agent produces a number denoting confidence in its recommendation being correct, so $p(q)$ is trivially available, and $p(\sim q)$ is simply $1 - p(q)$. For the utility, we use the following definition:

$$U_t(q) = -U_f(q) = f(N_e),$$

where f is some squashing function mapping the domain of $(0, \infty)$ to a range of $(0, 1]$, and N_e is the number of exemplars of a given type (that of the problem being considered) that the agent has seen. We chose $f(x) = \frac{2}{1+e^{-x}} - 1$ because it reflects the number ($x = N_e$) of problems of the present type that it has seen.

Having defined our notion of reasonableness, we still need a way to learn a mapping from a PDE problem to the most reasonable PYTHIA agent. We have evaluated standard statistical methods, gradient descent methods, machine learning techniques and other classes of algorithms [16], but it has been our experience that specialized techniques developed for this domain perform better than conventional off–the–shelf approaches [14]. In particular, we have designed a neuro–fuzzy technique that infers efficient mappings, caters to mutually non–exclusive classes (as the PDE problem classes naturally are) and learns the classifications in an on–line manner, see [16]. For the purposes of this paper, it is sufficient to understand that this scheme provides a mapping from a PDE problem to the best available recommender agent and that the mappings can be learnt in an incremental fashion using this reasonableness measure. While this mapping could be done by any of the PYTHIA agents by 'housing' a copy of the learned classification in each of them, we chose to create a central agent, PYTHIA–C whose main task is to perform this mapping. This just serves to demonstrate the learning aspect of the agents as distinct from their other capabilities.

6. CASE STUDIES

6.1 Solving Composite PDE Problems

The main issue is what mediation schemes can be applied to a composite PDE problem — in other words, how to obtain a global solution out of the local solutions produced by the single-domain solvers. To do this, SciAgents uses interface relaxation [3; 23]. Important mathematical questions of the convergence of the method, the behavior of the solution in special cases, etc., are addressed in [25]. Typically, for second order PDEs, there are two physical or mathematical interface conditions involving values and normal derivatives of the solutions on the neighboring subdomains. The interface relaxation technique is as follows.

. *Step 1.* Make initial guesses as boundary conditions to determine the subproblem solutions.

. *Step 2.* Solve the subproblem in each subdomain and obtain a local solution.

. *Step 3.* Use the solution values on the interfaces to evaluate how well the interface conditions are satisfied. Use a *relaxation formula* to compute new values of the boundary conditions.

. *Step 4.* Iterate steps 2 and 3 until convergence.

We now describe the solution of a composite PDE problem using four solvers and five mediators. It models the heat distribution in the walls of a chemical or a nuclear reactor and in the surrounding isolating and cooling structures, see Figure 4. The subdomains are shown, with the solver agents S_i, $i = 0, \ldots, 3$ simulating the local process in each subdomain and the mediators M_j, $j = 0, \ldots, 4$ mediating the interface piece they are written on. The unknown function is T and the exterior boundary conditions are shown next to the corresponding boundary pieces. The reactor keeps the inside temperature of its wall at 1000 degrees and the outside walls of the cooling structures are kept at, more or less, room temperature. The boundary conditions along the x and y axes reflect the symmetry of the construction. We denote by Γ_{ik} the k-th boundary piece of the i-th subdomain. The differential operators L_i, $i = 1, 2, 3$ are

$$\begin{aligned} L_1 &= T_{xx} + T_{yy} + \alpha_1 T - \beta_2(x^2 + y^2 - 2) \\ L_2 &= T_{xx} + T_{yy} + \alpha_2 T \\ L_3 &= T_{xx} + T_{yy} - \gamma_3(T_x + T_y) + \alpha_3 T \end{aligned} \tag{1}$$

The parameters are: $\alpha_1 = 0.2$, $\alpha_2 = 0.4$, $\alpha_3 = 0.3$, $\beta_2 = -60$, $\gamma_3 = 10$. We denote by Ω_i the subdomain associated with S_i, $i = 0, \ldots, 3$. We use as interface conditions the continuity of temperature and heat flow across the subdomain interfaces. Note that even though the interface between Ω_0 and Ω_1, Ω_2, and Ω_3 looks like a single curve from the point of view of Ω_0, it is divided into three pieces Γ_{02}, Γ_{03} and Γ_{04}, so that the mediators M_0, M_1, and M_2 can each be assigned a single piece to mediate. The time we spent from writing down the problem on paper to getting a contour plot of the solution on the screen was 5 hours (this includes some manual calculations and adjusting the relaxation formulas for better convergence).

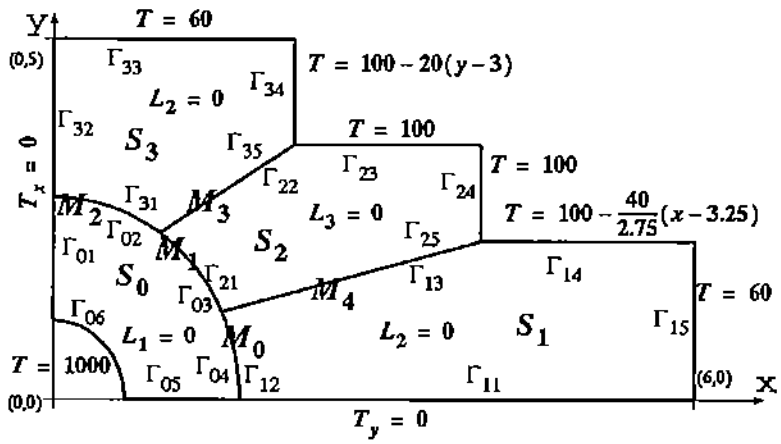

Fig. 4: A sketch of a composite PDE problem modeling the heat distribution in the walls of a chemical or a nuclear reactor and in the surrounding isolating and cooling structures. The subdomains are shown, with the solver agents S_i, $i = 0, \ldots, 3$ simulating the local process in each subdomain and the mediators M_j, $j = 0, \ldots, 4$ mediating the interface piece they are written on. The unknown function is T and the exterior boundary conditions are shown next to the corresponding boundaries. We denote by Γ_{ik} the k-th boundary piece of the i-th subdomain.

A user begins solving this problem by drawing Figure 4. The sketch identifies the subdomains (the solvers), the mediators, each boundary piece in every subdomain, and the endpoints of the interfaces. The sketch is necessary since the currently implemented version of *SciAgents* requires input as a script file. However, we believe that (with the possible exception of the boundary piece identifiers) such a sketch will be necessary even with the best imaginable graphical user interface. We only expect the user to annotate this initial sketch.

After making the sketch the user constructs the *SciAgents* input file and starts *SciAgents*. This starts the global controller (containing the agent instantiator) and it instantiates the agents on the appropriate machines and builds the network of four solvers and five mediators that is to solve the problem. After that, the "computing" thread of the global controller starts a shell-like interface with two major commands: `pause` and `tolerance` for control and steering the computations. The `pause` prompts the controller to issue messages to all agents to save their current state and to exit. The `tolerance` command changes dynamically the tolerance of a given mediator or of all mediators.

After the initial exchange of data to check that all agents are ready, the user sees four copies of the //ELLPACK user interface (see Figure 5). All four subproblems are defined (see Figure 6 for a snapshot during this process) and selecting a discretizer, linear solver, etc., in one subdomain does not lead to any requirement or necessity about selections in the neighboring subdomains. If a subdomain is huge, one may choose to use a 32-node Intel Paragon for it, while the neighboring tiny subdomain may be simulated on the same host where the wrapper is running. There are only two requirements for global synchronization of the local definitions: each subdomain geometry has to be input in terms of the global coordinate system (hence the need of the coordinates of the boundary pieces in the sketch), and for

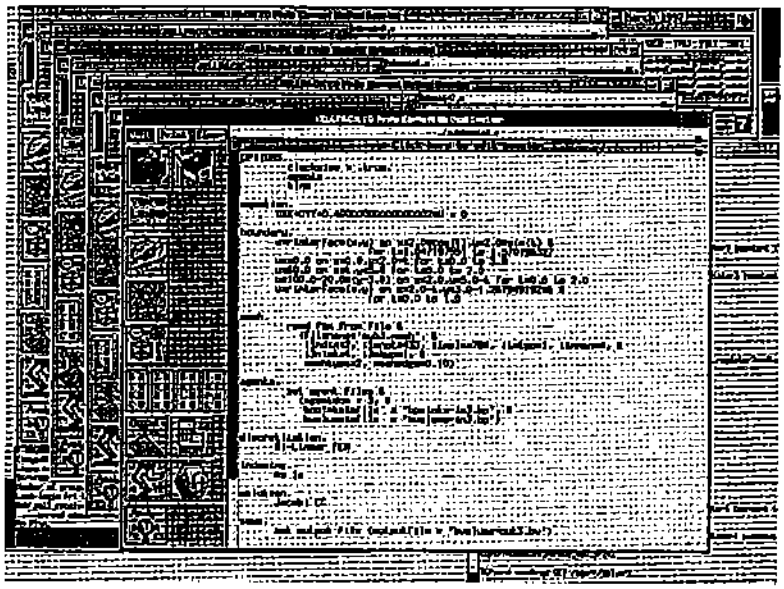

Fig. 5: Four copies of the //ELLPACK interface are presented to the user for defining the four PDE subproblems.

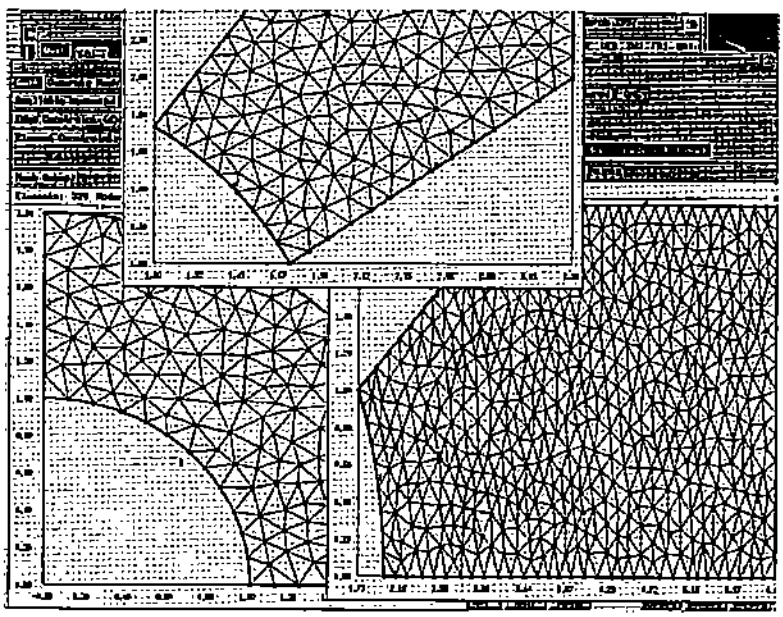

Fig. 6: A snapshot of the display during the subproblem definition process. Parts of three //ELLPACK domain tools containing three of the subdomain geometries and finite element meshes are visible. The user can discretize each subdomain completely independently from the others. For example, the densities of the above meshes are different.

each interface piece, the right-hand side of the boundary conditions has to be the function `rinterface(x,y)`. It is the user's responsibility to make sure that the relaxation formulas used for each interface piece correspond to the left-hand sides of the boundary conditions entered in the two solver's user interfaces. For the example, the boundary condition used at all interfaces is `T = rinterface(x,y)` and the relaxation formula is (U is the solution on the "left" side, V is the solution on the "right" side; U_n is the normal derivative; f is a factor given below; the formula is always applied pointwise for each point from any solver's grid/mesh on the interface):

$$U^{new} = V^{new} = \frac{1}{2}(U^{old} + V^{old}) - f \times (U_n^{old} - V_n^{old}) \tag{2}$$

The form of the factor f is $f = \frac{|U^{old}|+|V^{old}|}{(|U_n^{old}|+|V_n^{old}|)f_0}$ which scales the relaxation properly (and avoids dependencies on the choice of the coordinate system) and regulates the rate of change of the boundary conditions along the interface from iteration to iteration by changing f_0. It is sometimes hard to predict the "optimal", or even the acceptable, values of f_0.

The user input results in writing the script for the actual future runs. The user exits the //ELLPACK interface which prompts the wrapper to collect the initial data and to send them to the mediators. They compute initial right-hand sides of the boundary conditions. After the mediators provide all necessary boundary conditions, the wrapper runs the script which, in turn, runs the executable(s). When the iteration is completed the wrapper takes over again and extracts all required data from the computed solution and sends it to the mediators, waiting for the new boundary conditions from them. Thus, at the next iteration, no new compilation and user actions are necessary, since the same script (and executable(s)) is run by the wrapper.

For this example, we had to change the factor f_0 twice before the process began to converge, especially for mediators M_3 and M_4. This seems to be due to the natural singularity that occurs at the reentrant corners of the global domain which affects the stability of the convergence.

When a mediator observes convergence (the change of the boundary conditions for the next iteration is smaller than the tolerance), it reports this to the global controller, and after all mediators report convergence, the global controller issues a message to all agents to stop. In this case we had convergence after 53 iterations. Figure 7 shows a combined picture of all four subdomain solutions. Note that all contour lines match when crossing from one subdomain to another, there are even a few which go through three subdomains, and one going through all four subdomains. This is solid evidence that the interface relaxation technique works in this problem.

To experiment with the applicability of *SciAgents* to more difficult problems we solved several variations of the above example replacing L_1, L_2, and L_3 with nonlinear operators (exhibiting different nonlinearity for different L_i). Since //ELLPACK uses a Newton iterative procedure to solve a nonlinear problem, the global solution process becomes a multi-level iteration where one *SciAgents* step involves a complete Newton iteration in it. Also, while one can plausibly handle the linear ex-

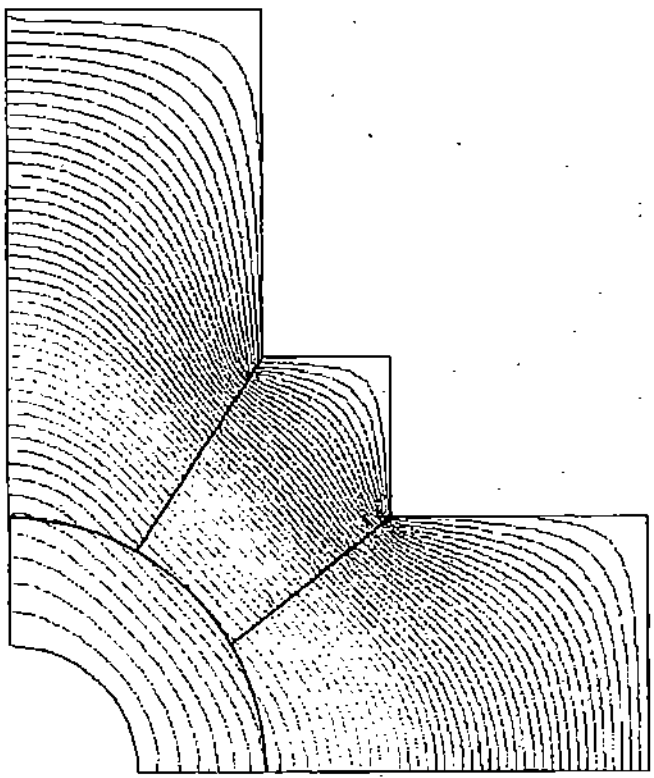

Fig. 7: A combined picture of all subdomain solutions of the example problem in Equation (1). The global solution corresponds to the physical intuition about the behavior of the modeled real-world system. All contour lines match when crossing from one subdomain to another, there are even a few which go through three subdomains and one going through all four subdomains.

ample above by considering a single PDE with discontinuous coefficients on a single domain, this approach is not feasible for nonlinear problems. Using *SciAgents* we were able to solve all of several such problems we tried (the increased complexity is reflected in a two to three-fold increase of the number of iterations necessary for the convergence to the global solution).

6.2 Intelligent PDE Computing with PYTHIA

In this section, we describe how PYTHIA can be used to determine reasonable strategies for PDE problem solving. In our prototype implementation, our PYTHIA agents' expertise stems from the following classes of PDEs (we also list the number of samples in each class from our study that involves about 167 PDE problems):

(1) SINGULAR: PDE problems whose solutions have at least one singularity (6 exemplars).

(2) ANALYTIC: PDE problems whose solutions are analytic (35 exemplars).

(3) OSCILLATORY: PDE problems whose solutions oscillate (34 exemplars).

(4) BOUNDARY–LAYER: Problems with a boundary layer in their solutions (32 exemplars).

(5) BOUNDARY–CONDITIONS–MIXED: Problems that have mixed boundary conditions in their solutions (74 exemplars).

(6) SPECIAL: Problems that do not belong to the above classes (10 problems).

Note that these classes are not mutually-exclusive, so their total membership is 191 problems. In other words, there are different PYTHIA agents, each of which can recommend a solver for a PDE belonging to its representative class(es) of problems. Also, a problem can belong to more than one class simultaneously (a given PDE can *both be* analytic *and have* mixed boundary conditions). Detecting the presence of such mutually non–exclusive classes is critical to selecting a good solver for the PDE.

To test our ideas, we made five experiments, with 2, 3, 4, 5 and 6 PYTHIA agents respectively. In each experiment, each PYTHIA agent knows about a certain class(es) of PDE problems. For example, with 6 PYTHIA agents, each agent knows about one of the above classes of PDEs. In the '3–agent' experiment, agent 1 knows about problem classes 1 and 2, agent 2 knows about classes 3 and 4 and the third agent knows about classes 5 and 6. The population of 167 PDE problems was split into two parts: a large set of 111 problems and a smaller set of 56 problems. We conducted two sets of experiments: In each scenario, we first trained our technique on the larger set of {problem, agent} pairs (using the notion of reasonableness defined earlier) and tested our learning on the smaller set of 56 exemplars. In the second experiment, the roles of these two sets were reversed. We also compared our technique with two very popular gradient descent techniques for training feedforward neural networks, namely, Vanilla (Plain) Backpropagation (BProp) and Resilient Propagation (RProp). Fig. 8 summarizes the results.

It can be easily seen that our method consistently outperforms BProp and RProp on learning the mapping from problems to agents. Also, performance on the larger training set was expectedly better than that on the smaller training set. Moreover, our algorithm operates in an on–line mode; new data do not require retraining

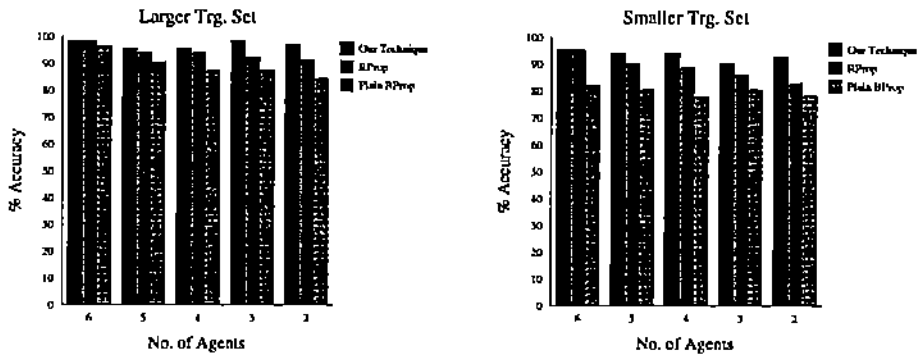

Fig. 8: Performance of learning algorithms. The graph on the left depicts the results with the larger training set and the one on the right shows the results with the smaller training set. In each case, recommendation accuracy figures for the 5 experiments (with 2, 3, 4, 5 and 6 agents) are presented for all the three learning algorithms considered in this paper.

on the old. Our technique was also tested for this ability; for the larger training set, we incrementally trained our algorithm on the 111 PDEs and the accuracy figures on the test set were found to rise steadily to the figures shown in Fig. 8. In the collaborative networked scenario of an MPSE, where the resources change dynamically, this feature of our neuro–fuzzy system enables us to automatically infer the capabilities of multiple PYTHIA agents. If the capabilities of agent 1 were to change, for example, in the 6–agent scenario, then our network could infer the new mappings without losing the information already learnt. This feature is absent in most other methods of classification such as BProp and RProp in which the dimensionality of the network is fixed and it is imperative that the old data be kept around if these networks are to update their learning with new data.

The PYTHIA project web pages at *http://www.cs.purdue.edu/research/cse/pythia* provide information about this collaborative PYTHIA methodology and facilities to invoke it remotely. At the outset, there is a facility to provide feature information about a PDE problem. In particular, there are forms that guide the user in providing information about the operator, function, domain geometry and boundary conditions. Once these details are given, the information is submitted to the central PYTHIA agent, PYTHIA–C, that performs further processing. As mentioned before, it first classifies the given PDE problem into categories of problems as described above. Having classified the problem into one or more of these classes, the PDE is taken to an appropriate PYTHIA agent for this class of problems, which in turn predicts an optimal strategy and reports back to the user.

6.3 Learning and Adaptation in MultiAgent Systems

The above experiment can be visualized as an example where the central agent PYTHIA–C is in a *learning mode*, cycles through the training set, and learns mappings from the given PDEs to appropriate agents. From this point on PYTHIA–C is in the *stable mode*. It will only ask the best agent to answer a particular question. If PYTHIA–C finds a PYTHIA agent's recommendation unacceptable, it will ask the next best agent, until all agents are exhausted. This is facilitated by our neuro–

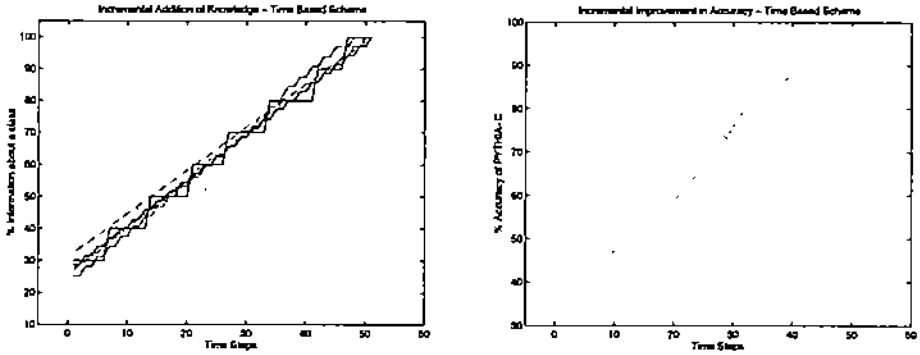

Fig. 9: Results with the time based scheme for 6 agents using the larger training set. The graphs on the left show the systematic increase in the abilities for each of the agents individually and the one on the right shows the corresponding improvement in accuracy of the central agent, PYTHIA-C

fuzzy learning algorithm. By varying an acceptance threshold in the algorithm, we can get an enumeration of "not so good" agents for a problem type. If PYTHIA-C determines no plausible solution exists among its agents or itself, then PYTHIA-C gives the answer that "is best". When giving such an answer, the user is notified of PYTHIA-C's lack of confidence.

While this scheme serves most purposes, an issue still pending is the mode of switching between the learning and stable modes. PYTHIA-C switches from learning to stable mode after an *a priori* fixed number of problems (this was 111 in our first set of experiments, for example). The timing of the reverse switch back to learning is a more interesting problem; we report on three different methods.

Time based: This simple approach is where PYTHIA-C reverts to learning after a fixed time period. At such points, PYTHIA-C cycles through its training set, queries other agents, gets back answers, determines reasonableness values and finally learns new mappings for the PDE problems. Figs. 9 depicts the results with the six-agent case and the time based approach using the larger training set. Initially, each agent starts up with approximately 1/3 of their total knowledge base and this knowledge steadily increases with time. At periodic time intervals, PYTHIA-C switches to learning mode and cycles through the larger training set with each of the agents in the experiment. The performance is then measured with the smaller training set. As can be seen, the accuracy figure steadily improves for each of the six individual agents to the accuracy observed in the previous static experiment. PYTHIA-C's accuracy improves from 40.85% to 98.20% in this experiment.

We conducted another experiment with this method, one more realistic for multi-agent systems. We begin the experiment with no 'known' agents, i.e., PYTHIA-C initially does not know about the existence of any agents or their capabilities. Then, each agent is introduced into the experiment with a small initial knowledge base and then their knowledge base is slowly increased. For example, Agent 1 comes into the setup with a small knowledge base and announces its existence to PYTHIA-C which creates a class for Agent 1. It then reverts to learning mode (though wasteful) and learns mappings from PDE problems to agents (in this case, there is only one agent). After some time, Agent 3 comes into the experiment and this process is

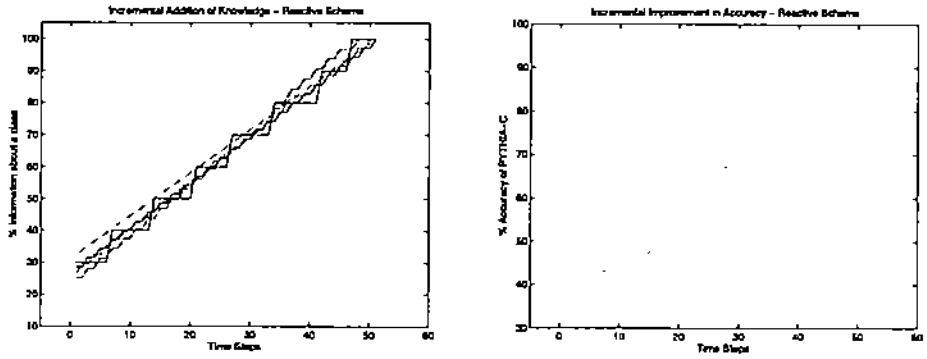

Fig. 10: Results with the reactive method for 6 agents using the larger training set. The graphs on the left show the systematic increase in the abilities for each of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA–C

repeated. This is repeated until all six agents are introduced. While the addition of new agents and associated classes is taking place, the abilities of existing agents (like Agent 1) also increase simultaneously. Thus, these events happen in parallel; i.e., addition of new agents and additions to the knowledge base of existing agents. Because our neuro–fuzzy scheme has the ability to introduce new classes on the fly, PYTHIA–C can handle this situation well. The accuracy figures converge to the values previously obtained.

Reactive: In this method, a PYTHIA agent notifies PYTHIA–C whenever its confidence for some class of problems has changed significantly. PYTHIA–C reverts to learning when it next receives a query about this class of problems. Each agent started with the same initial knowledge base as before and this is slowly increased. As the agents indicate the resulting increase in confidence to PYTHIA–C, it reverts to learning mode from time to time. The accuracy figures for PYTHIA–C approach the same values as before; they follow a monotonic pattern, but a more slowly increasing pattern, see Fig. 10.

Time based reactive: This is a combination of the two methods above where PYTHIA–C sends out a "has anyone's abilities changed significantly" message at fixed time intervals, and switches to learning if it receives a positive response. Each agent, starts with the same knowledge base and this is slowly increased. Fig. 11 shows that the accuracy figures for PYTHIA–C are again a monotonic increasing and rising slightly faster than for the reactive method.

Our experiments with the three methods show that they enable the central agent PYTHIA–C to keep track of the dynamic capabilities (in our case, the knowledge base) of other agents. These methods also enable PYTHIA–C to handle situations where agents appear and disappear over time.

REFERENCES

[1] R.F. Boisvert. The NIST Guide to Available Mathematical Software. `http://math.nist.gov/gams/`, 1996.

[2] S. Cammarata et al. Strategies of Cooperation in Distributed Problem Solving. In Bond and Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 102–105. Morgan

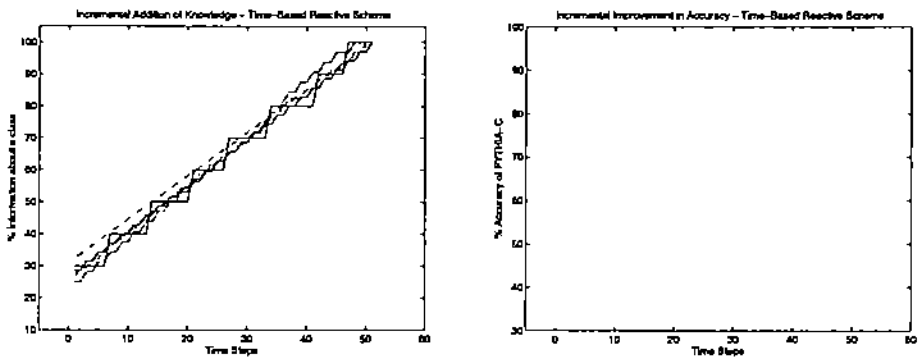

Fig. 11: Results with the time based reactive method for 6 agents using the larger training set. The graphs on the left show the systematic increase in the abilities for each of the PYTHIA agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C.

Kaufmann, 1988.

[3] T. T. Drashansky. *An Agent-Based Approach to Building Multidisciplinary Problem Solving Environments*. PhD thesis, Dept. Comp. Sci., Purdue University, December 1996.

[4] Wayne R. Dyksen and Carl R. Gritter. Elliptic Expert: An Expert System for Elliptic Partial Differential Equations. *Mathematics and Computers in Simulation*, 31:333–343, 1989.

[5] Wayne R. Dyksen and Carl R. Gritter. Scientific Computing and the Algorithm Selection Problem. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Expert Systems for Scientific Computing*, pages 19–31. North–Holland, 1992.

[6] R. Fritzson et. al. KQML- A Language and Protocol for Knowledge and Information Exchange. In *Proc. 13th Intl. Distributed Artificial Intelligence Workshop*, July 1994.

[7] E. Gallopoulos, E. N. Houstis, and J. R. Rice. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Computational Science and Engineering*, 1(2):11–23, 1994.

[8] J. C. Giarratano. *CLIPS User's Guide, Version 5.1*. NASA Lyndon B. Johnson Space Center, 1991.

[9] S.S. Grimajl. The First ICASE/LARC Industry Roundtable: Session Proceedings. Technical Report ICASE Interim Report 26, ICASE, NASA Langley Research Center, Hampton, VA, 1995.

[10] B. Hayes-Roth et al. Guardian. A Prototype Intelligent Agent for Intensive-care Monitoring. *Artif. Intell. Med*, 4(2):165–185, 1992.

[11] A. Joshi. To Learn or Not to Learn ... In G. Weiss and S. Sin, editors, *Adaptation and Learning in Multiagent Systems*, volume 1042 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 1996.

[12] A. Joshi, T. Drashanksy, J.R. Rice, S. Weerawarana, and E.N. Houstis. Multiagent Simulation of Complex Heterogeneous Models in Scientific Computing. *IMACS Math. Comp. Simulation*, 1997. (accepted for publication), http://www.cs.purdue.edu/research/cse/agents.

[13] A. Joshi, N. Ramakrishnan, J.R. Rice, and E. Houstis. A Neuro-Fuzzy Approach to Agglomorative Clustering. In *Proc. IEEE Intl. Conf. on Neural Networks*, volume 2, pages 1028–1033. IEEE Press, July 1996.

[14] A. Joshi, N. Ramakrishnan, J.R. Rice, and E. Houstis. On Neurobiological, Neuro–Fuzzy, Machine Learning and Statistical Pattern Recognition Techniques. *IEEE Trans. Neural Networks*, 8(1):18–31, 1996.

[15] A Joshi, S. Weerawarana, and E. N. Houstis. The Use of Neural Networks to Support Intelligent Scientific Computing. In *Proc. IEEE Intl. Conf. Neural Networks*. IEEE, IEEE Press, July 1994.

[16] A. Joshi, S. Weerawarana, E. N. Houstis, J. R. Rice, and N. Ramakrishnan. Neuro-Fuzzy Sup-

port for Problem Solving Environments. *IEEE Computational Science and Engineering*, 3:44–56, 1996.

[17] A. Joshi, S. Weerawarana, N. Ramakrishnan, E.N. Houstis, and J.R. Rice. Neuro-Fuzzy Support for PSEs: A Step Toward the Automated Solution of PDEs. *Special Joint Issue of IEEE Computer & IEEE Computational Sci ence and Engineering*, vol.3(1):pp.44–56, 1996.

[18] A. Joshi et al. On Learning and Adaptation in Multiaget Systems: A Scientific Computing Perspective. Technical Report TR-95-040, Dept. Comp. Sci., Purdue University, 1995.

[19] M. S. Kamel, K. S. Ma, and W. H. Enright. ODEXPERT: An Expert System to Select Numerical Solvers for Initial Value ODE Systems. *ACM Trans. Math. Software*, 19:44–62, 1993.

[20] D. Leake. Case-based selection of Problem Solving Methods for Scientific Computation. http://www.cs.indiana.edu/hyplan/leake/cbmatrix.html, 1996.

[21] K. Lehrer. *Theory of Knowledge*. Westview Press, Boulder, CO, USA, 1990.

[22] V. R. Lesser. A Retrospective View of FA/C Distributed Problem Solving. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1347–1363, 1991.

[23] S. McFaddin and J. Rice. Collaborating PDE Solvers. *Appl. Num. Math*, 10:279–295, 1992.

[24] Peter K. Moore, Can Ozturan, and Joseph E. Flaherty. Towards the Automatic Numerical Solution of Partial Differential Equations. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Intelligent Mathematical Software Systems*, pages 15–22. North–Holland, 1990.

[25] Mo Mu and J. R. Rice. Modeling with Collaborating PDE Solvers — Theory and Practice. *Computing Systems in Engineering*, 6:87–95, 1995.

[26] T. Oates et al. Cooperative Information Gathering: A Distributed Problem Solving Approach. Technical Report TR-94-66, Computer Science, University of Massachusetts, Amherst, 1994.

[27] N. Ramakrishnan, A. Joshi, E.N. Houstis, and J.R. Rice. Neuro-Fuzzy Approaches to Collaborative Scientific Computing. In *Proc. IEEE Intl. Conf. on Neural Networks*. IEEE Press, 1997. to appear.

[28] N. Ramakrishnan, A. Joshi, S. Weerawarana, E.N. Houstis, and J.R. Rice. Neuro-Fuzzy Systems for Intelligent Scientific Computing. In *Proc. Artificial Neural Networks in Engineering ANNIE '95*, pages 279–284, 1995.

[29] N. Ramakrishnan and J.R. Rice. GAUSS: An Automatic Algorithm Selection System for Quadrature. Technical Report TR-96-048, Dept. Computer Sciences, Purdue University, 1996.

[30] N. Ramakrishnan, J.R. Rice, and E.N. Houstis. Knowledge Discovery in Computational Science: A Case Study in Algorithm Selection. Technical Report TR-96-081, Dept. Computer Sciences, Purdue University, 1996.

[31] P. Resnick and H. Varian. Recommender Systems. *Comm. ACM*, 40(3):56–58, March 1997.

[32] J.R. Rice. The Algorithm Selection Problem. *Advances in Computers*, vol.15:pp.65–118, 1976.

[33] J.R. Rice. Methodology for the algorithm selection problem. In L. Fosdick, editor, *Performance Evaluation of Numerical Software*, pages 301–307. North Holland, 1979.

[34] J. C. Schlimmer and L. A. Hermens. Software Agents: Completing Patternsand Constructing User Interfaces. *Journal of Artificial Intelligence Research*, 1(61-89), 1993.

[35] Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[36] R. G. Smith and R. Davis. Frameworks for Cooperation in Distributed Problem Solving. In Bond and Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 61–70. Morgan Kaufmann, 1988.

[37] L.H. Tsoukalas and R.E. Uhrig. *Fuzzy and Neural Approaches in Engineering*. John Wiley and Sons, Inc., Third Avenue, New York, 1997.

[38] L. Z. Varga et. al. Integrating Intelligent Systems into a Cooperating Community for Electricity Distribution Management. *International Journal of Expert Systems with Applications*, 7(4), 1994.

[39] R.G. Voigt. Requirements for Multidisciplinary Design of Aerospace Vehicles on High Performance Computers. Technical Report ICASE Report No. 89-70, ICASE, NASA Langley

Research Center, Hampton, VA, 1989.

[40] S. Weerawarana. *Problem Solving Environments for Partial Differential Equation Based Systems*. PhD thesis, Dept. Comp. Sci., Purdue University, 1994.

[41] S. Weerawarana, E. N. Houstis, J. R. Rice, A. C. Catlin, C. L. Crabill, C. C. Chui, and S. Markus. PDELab: An Object-Oriented Framework for Building Problem Solving Environments for PDE Based Applications. In *Proc. Second Annual Object-Oriented Numerics Conference*, pages 79–92, Rogue–Wave Software, Corvallis, OR, 1994.

[42] S. Weerawarana, E.N. Houstis, J.R. Rice, A. Joshi, and Houstis C.E. PYTHIA: A knowledge based system to select scientific algorithms. *ACM Trans. Math. Software*, 22(4):447–468, 1996.

[43] R. Wesson et al. Network Structures for Distributed Situation Assessment. In Bond and Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 71–89. Morgan Kaufmann, 1988.

[44] M. Wooldridge and N. Jennings. Intelligent Agents: Theory and Practice. (submitted to Knowledge Engineering Review).