MULTIAGENT SYSTEM SUPPORT OF NETWORKED SCIENTIFIC COMPUTING

ANUPAM JOSHI
University of Missouri
NAREN RAMAKRISHNAN AND ELIAS N. HOUSTIS
Purdue University

A prototype multiagent "recommender system" is part of ongoing work to create multidisciplinary problem-solving environments that will enhance the use of networked resources in scientific computing.

Rapid advances in high-performance computing and the growth of the Internet are heralding a paradigm shift in scientific computing toward networked environments. Scientists can now view highly specialized, distributed hardware and software as a single metacomputer that supports research on complex problems whose scope entails life-cycle simulation. The design and simulation for these problems involve hundreds of components functioning under detailed physical laws and different operational or behavioral constraints over time.

The payoff for solving these problems is potentially huge, yet the state of the art in networked scientific computing has progressed very slowly. Application scientists and engineers who wish to use networked systems outside a laboratory environment must still handle low-level distribution and networking issues to a degree that seriously inhibits their research. They must also identify the best resources for solving a given problem from specialized scientific software servers, libraries, repositories, and problem-solving environments distributed across the Internet—a requirement that is not realistic, especially given continual changes in the hardware and software resources available.

For networked resources to support the scientific computing process, they must become as simple to use as networked information resources like the Web are today. In this article, we explore the use of multiagent systems to implement high-level multidisciplinary problem-solving environments that harness the power of internetworked computational resources for solving scientific computing problems. We begin with an overview of the resources available to advance the state of the art in networked scientific computing. This is followed by an overview of our multiagent systems approach to harnessing these resources in a manner transparent to the user.

1089-7801/98/$10.00 ©1998 IEEE

The balance of the article focuses on one component of the system, recommender agents. These agents have a distributed knowledge base of performance data from previously solved problems, which they use to suggest resources for a new problem. We conclude with experimental data that shows the effectiveness of a prototype implementation of this approach in a networked environment.

ONLINE RESOURCES AND PSEs

There are several software libraries ("solvers") available for use in scientific computing that are network accessible. **Netlib** and Lapack/ScaLapack are two. There are also some efforts to make solvers accessible over the Web, such as **Web//Ellpack** and **Net//Ellpack** from Purdue University, and **NetSolve** from the University of Tennessee–Knoxville and the Oak Ridge National Laboratory. In addition, some online resources help users identify and locate the right class of software for a problem, such as the **GAMS**, the US National Institute of Standards and Technology's Guide to Available Mathematical Software.

However, at this time users must still identify the specific software most appropriate for a problem; download the software and its installation and use instructions; install, compile, and possibly port it; and learn how to invoke it appropriately. These tasks are nontrivial even for a single piece of software, and much more complicated when multiple software components are involved.

To help with these difficulties, the scientific computing community has proposed implementing *problem-solving environments*.[1] A PSE is a computer system that gives the user a high-level abstraction of underlying computational facilities for solving a target class of problems. PSEs use the language of the target problem and provide a "natural" interface that accepts high-level problem descriptions. PSEs give users access to advanced solution methods, automatic or semiautomatic tools for selecting solution methods, and techniques to easily incorporate novel methods.

The idea of **MPSEs** (multidisciplinary PSEs) has been proposed to provide similar support for domain experts using networked services to devise scientific computing applications.[2,3] We have developed an approach to MPSEs that views them as *multiagent systems*. The basic idea is to replace a complex simulation problem by a set of simpler simulation problems defined on elementary geometries. Each of these problems is mapped to a solver agent (as described in the next section) and solved simultaneously, along with a set of interface conditions. The simpler problems may reflect the underlying structure of the system to be simulated; they can also be artificial creations based on techniques such as domain decomposition.[4]

A MULTIAGENT MPSE SYSTEM

Figure 1 illustrates the software organization of our

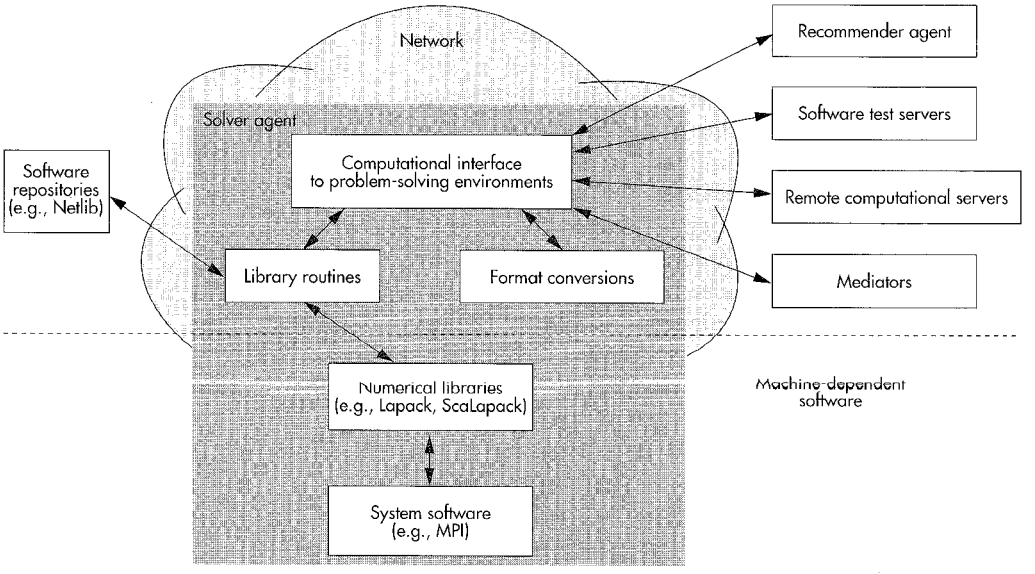

Figure 1. Software organization for a multiagent systems approach to networked scientific computing. The solver agent provides a network interface to a problem-solving environment, and connects to other resources and agents for performance evaluation, resource selection, software retrieval, and remote computation.

approach. We have created a prototype system[5] that integrates three kinds of interacting agents: solvers, mediators, and recommenders. The prototype uses Ellpack[6] as the solver agent, special mediators that implement interface relaxation techniques,[4,7-8] and the Pythia[9] knowledge-based system to implement recommender agents. Ellpack solvers can consult a Pythia agent to automatically determine and invoke the right software for a given problem (the system does not yet automatically select the right hardware).

Our approach supports *distributed problem-solving*, as distinguished from just distributed computing. Although the agent-based paradigm has not been widely used in scientific computing, we believe the ability of agents to autonomously pursue their goals without user intervention can resolve many difficulties associated with the development of networked scientific computing. The software modules that solve the simpler problems, computing locally and interacting with neighboring solvers, effectively translate into the behavior of local problem-solving entities. We view each one as a *solver agent*. By introducing *mediator agents* between them, we can create a network of collaborating solvers to solve the global problem.

Each solver agent deals with one subproblem, and can run on a separate machine across the network. The original problem is solved when all equations on the individual components are satisfied and the solutions "match properly" on the interfaces between the components. The mediator agents make this determination. The match is defined by physics if the interface is where the physics changes; for example, with heat flow, it means that temperature is the same on both sides of the interface and the amount of heat flowing into one component is the same as the amount flowing out of the other. If the interface is artificial, then a proper match is defined mathematically and means that the solutions have continuous values and derivatives across the interface.

The solver and mediator agents form a potentially large pool of software spread across the network. In the case of solvers, a significant amount of sophisticated problem-specific code has already been developed over the past few decades. While mediators are almost nonexistent today, a large number will have to be created to support the interaction of disparate solvers. Expecting users to know all the potential networked solvers for a problem or to understand all the hardware choices is clearly not realistic—the users are application scientists and engineers, not computer scientists. Therefore, we use *recommender agents* to automatically suggest networked software components and hardware resources that can be deployed to solve a defined scientific computing problem according to performance criteria specified by the user.

In our approach, the selection of optimal resources to solve a scientific computing problem is called the *resource-selection* problem. This is a generalization of the algorithm-selection problem formulated by Rice.[10] The recommender agents we employ are conceptually similar to the recommender systems proposed by Resnick and Varian[11] for harnessing distributed information resources. The agents accept a problem definition from the user, including performance criteria, and suggest software and hardware resources that can be deployed to solve it.

Solver agents interact with their neighbors in pursuing autonomous goals.

Note that the task here is different from invoking a known method remotely on some object, for which a host of distributed object-oriented techniques such as Java RMI, CORBA, and DCOM are becoming commercially available. Similar academic approaches geared toward scientific computing can be seen in the **Legion** and **Globus** systems. Our approach is at a higher level of abstraction, and an implementation can use the facilities provided by such tools.

Details about solver and mediator agents are documented elsewhere.[3-5] This article focuses on the recommender aspects of a multiagent MPSE system.

PYTHIA: THE CORE OF RECOMMENDER AGENTS

The core of each recommender agent is Pythia,[9] an intelligent system we developed to automatically select algorithms in well-defined scientific domains. Pythia attempts to determine an optimal strategy (that is, a solution method and its parameters) for solving a given problem within user-specified resource limits and accuracy requirements.

Pythia agents are implemented by a combination of C language routines, shell scripts, and rule-based systems such as CLIPS[12] (the C Language

Integrated Production System). The agents communicate through the Knowledge Query and Manipulation Language,[13] using protocol-defined performatives. All Pythia agents understand and use a private language, Pythia-Talk.

There is also an agent name server (ANS) that determines the URL associated with a particular Pythia agent. We use the TCP transport protocol to access individual Pythia agents. In other words, agent URLs have the format tcp://hostname:port-number. When a new Pythia agent comes into existence, its URL can be dynamically registered with the ANS by an appropriate KQML API call.

Pythia agents use case-based reasoning to recommend solution methods.

While the techniques involved are general, our current implementations of Pythia operate in conjunction with systems that solve partial differential equations (PDEs) and problems in numerical quadrature. 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. There are many software numerical solvers available for PDEs. The solvers normally require the user to specify several parameters to obtain a solution within a specified error level while satisfying resource constraints such as system memory and time.

These specifications are difficult in themselves and complicated by the heterogeneity of machines, including parallel machines, available across the network to solve PDE-based problems. Furthermore, the mathematical characteristics of PDE models can make thousands of numerical methods applicable, since several choices of parameters or methods may be required at each phase of the solution. On the other hand, the numerical solution must satisfy several objectives, primarily involving error and hardware resource requirements.

A Pythia agent accepts the description of an elliptic PDE problem as input, and identifies the methods appropriate to solve it. Its strategy is similar to that believed to underlie human problem-solving skills—that is, it compares new problems to ones it has seen before and uses its knowledge about the performance characteristics of similar problems to evolve a strategy for solving the current one.

The artificial intelligence literature defines this strategy as case-based reasoning. The goal of the reasoning process then is to recommend a solution method and applicable parameters that can be used to solve the user's problem within the given computational and performance objectives. This goal is achieved by

- analyzing the PDE problem and identifying its characteristics;
- identifying the set of similar problems from previously solved problems,.
- extracting all information available about this set of problems and the applicable solvers, and selecting the best method; and
- using performance information from this method to predict its behavior for the new problem.

One way to decide the closest match to a new problem is to directly compare it with all previously seen problems. Such lookup can be done efficiently using specialized data structures such as multidimensional search trees. Alternatively, problems can be assigned to meaningful classes, which are then used to find similar problems. This implies a two-stage process for finding the closest match, which we follow in our work.

Classifying problems into subsets and determining which subset a particular problem belongs to can be implemented in two basic ways. A *deterministic* method defines a class of problems by the centroid of the characteristic vectors of all class members. A class membership is defined as being within a certain radius of the centroid in the characteristic space. A new problem is assigned to the class whose centroid is closest to it. Distance between characteristics determines which particular member of a class is closest to the new problem.

While some classes offer a completely deterministic (and simple) way to determine class membership, others do not. Instead, they require a *derivative* method that determines the class structure from specific samples. To support it, we have developed appropriate architectures and experimented with a variety of "intelligent" techniques—symbolic, fuzzy, connectionist, and hybrid approaches—to determine what kind of a learning system works well in this domain. Pythia also provides a confidence factor between 0 and 1 for the method.

The following is an example typical output for a PDE problem:

> Use the 5-point star algorithm with a 200 x 200 grid on an NCube/2 using 16 processors: Confidence: 0.85.

As Pythia encounters new problems and "predicts" strategies for solving them, it adds information about these problems and their solutions to its knowledge base.

Details about the intelligent mechanisms that support Pythia functions are presented elsewhere.[9,14]

COLLABORATIVE PYTHIA

The underlying model of computation for Pythia is quite simple—essentially a static paradigm dealing with a single agent. In a networked scenario, several intelligent agents will exist across the Internet, each one possibly offering a specialized knowledge base about problems. For example, there are many different types of PDEs. Most scientists tend to use only a limited number of them, so any single Pythia agent they used will have a performance knowledge base limited to such problems. Moreover, agents' capabilities change as their knowledge bases change (for example, when they get more performance data) and agents can appear and disappear over time.

A mechanism that allowed Pythia agents to collaborate over the network could increase the range of problems and solutions. In a collaborative scenario, if an agent discovers that it does not know about a particular problem or does not have enough confidence in the prediction it is making, it could query other Pythia agents and obtain answers from them—that is, suggestions on what resources to use to solve a given problem. The agent could then pick up the "best" suggestion and follow it.

However, this "broadcast mode" can entail a huge amount of traffic on the system. A better approach is to use the information obtained from the initial queries to learn/infer a mapping from a problem to a Pythia agent deemed mostly likely to suggest good solution resources for it.

This mapping would allow the first agent to direct subsequent queries more effectively. However, because the environment is dynamic—with the numbers and abilities of individual agents changing over time—the mapping technique would have to be able to learn online. In other words, to retain its efficiency, the mapping technique must assimilate new data without going over previously learned information. (Many popular "learning" algorithms fail to meet this criterion.)

We have developed a neuro-fuzzy method of learning suitable for this purpose. The method uses neural techniques to model complex relationships and classify concepts into predefined classes. It uses

A MEASURE OF REASONABLENESS

We use a quantitative measure of reasonableness[1] to automatically generate exemplars to learn the mapping from a problem to an agent. To do this in an unsupervised manner, we combine two factors, one denoting the probability of a proposition q being true, and the other denoting its utility. Specifically, we follow Lehrer[2] to define the reasonableness of a proposition as follows:

$$R(q) = p(q)U_t(q) + p(\sim q)U_f(q)$$

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 $p(q)$ denotes 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:

$$U_t(q) = -U_f(q) = f(N_\varepsilon)$$

where f is some squashing function mapping the domain of $(0, \infty)$ to a range of $(0, 1]$, and N_ε is the number of exemplars of a given type (that of the problem being considered) that the agent has seen.

As the measure of an agent's utility, we chose $f(x) = (2/(1+\varepsilon)^{-x}) - 1$ because it reflects the number of problems of the present type that the agent has seen. The value of a utility function is to measure the amount of knowledge that an agent appears to have. The more problems of a certain kind in an agent's knowledge base, the more appropriate will be its predictions for new problems of the same type. Hence, we have designed the utility to be a function of N_ε. This formulation is not unique, though.

For example, assume that the probabilities of two hypotheses being true are identically p (where $p \leq 0.5$), and the positive utilities are U_{t1} and U_{t2} (with $U_{t1} \geq U_{t2}$). In such a case, the second hypothesis will be assigned a greater reasonableness than the first, in spite of the fact that the first hypothesis has a greater utility. This problem arises because multiplying an inequality by a negative quantity reverses its direction. Therefore, we check for this occurrence and invert the signs of U_t and U_f to keep our notion of reasonableness consistent. Other measures of utility are certainly possible.

REFERENCES

1. A. Joshi, To Learn or Not To Learn . . . ," in *Adaptation and Learning in Multiagent Systems*, Vol. 1,042 of *Lecture Notes in Artificial Intelligence*, G. Weiss and S. Sin, eds,, Springer Verlag, 1996.
2. K. Lehrer, *Theory of Knowledge*, Westview Press, Boulder, Colo., USA, 1990.

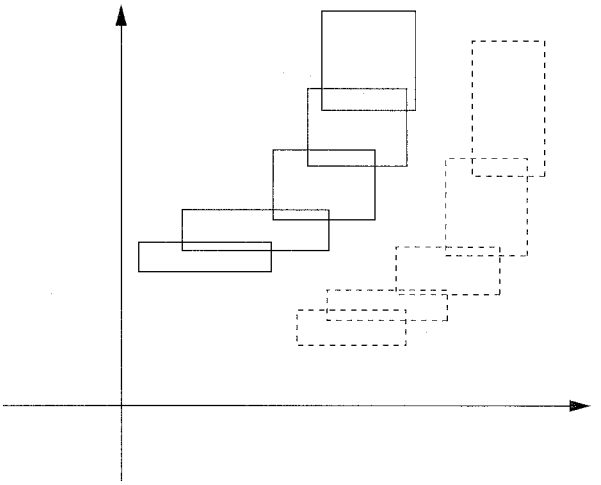

Figure 2. The neuro-fuzzy classification algorithm represents pattern classes by hyperboxes that define an n-dimensional pattern space for class membership. In this domain, there are two classes, depicted by the solid and the dotted hyperboxes.

fuzzy techniques to model uncertainty and represent applications where membership in classes varies in degree (see the sidebar, "A Measure of Reasonableness").

Thus, the goal of the collaborative Pythia system is to use a network of collaborating Pythia agents as recommenders to enable networked scientific computing. A neuro-fuzzy approach enables each agent to learn about the capabilities of other Pythia agents in the network.

Neuro-Fuzzy Approaches

While we have worked on several techniques to allow agents to learn a mapping from problem type to agent (for example, statistical methods, gradient descent methods, machine-learning techniques, and other algorithm classes), we have found that specialized techniques developed for this domain perform better than conventional off-the-shelf approaches.[12] In particular, our neuro-fuzzy technique infers efficient mappings, caters to mutually nonexclusive classes that characterize real-life domains, and learns these classifications online. We provide only the details of this algorithm that are relevant in the current context; complete details are documented elsewhere.[15]

The classification scheme uses fuzzy sets to describe pattern classes.[16] These sets are represented, in turn, by the fuzzy union of several hyperboxes. Such unions of hyperboxes define a region in n-dimensional pattern space that contain patterns with full membership in the class. For example, Figure 2 shows two classes in a domain: one with solid-bounded hyperboxes and one with dotted-line boundaries.

A hyperbox is completely defined by a min-point and max-point, and also has a fuzzy membership function (with respect to these min-max points) associated with it. This function defines class membership for points outside the box, which helps view the hyperbox as a fuzzy set. "Hyperbox fuzzy sets" can be aggregated to form a single fuzzy set class. This provides inherent degree-of-membership information that can be used in decision making.

The resulting structure fits neatly into a three-layer feed-forward neural network assembly. Learning in the network proceeds by placing and adjusting these hyperboxes. Recall in the network consists of calculating the fuzzy union of the membership function values produced from each of the fuzzy set hyperboxes.

Initially, the system starts with an empty set (of hyperboxes). As it "learns" each pattern sample, either an existing hyperbox (of the same class) is expanded to include the new pattern or a new hyperbox is created to represent it. The latter case arises when there is no existing hyperbox of the same class or when an existing hyperbox cannot expand any further because it has reached a limit set on such expansions.

Simpson's original method assumed that the pattern classes underlying the domain are mutually exclusive and that each pattern belongs to exactly one class.[17] But the pattern classes that characterize problems in many real-world domains are frequently not mutually exclusive. Consider the problem of classifying geometric figures. Polygons, squares, and rectangles are not mutually exclusive classes; for example, a square is also a rectangle and polygon. It is possible to apply Simpson's algorithm to this problem by first reorganizing the data into disjoint classes such as "rectangles that are not squares" and "polygons that are not rectangles." However, this strategy does not reflect the natural overlapping characteristics of the underlying base classes. Thus Simpson's algorithm fails to account for a situation where one pattern might belong to several classes. Simpson's method has a parameter called the maximum hyperbox size, which denotes the limit beyond which the hyperbox cannot expand to enclose a new pattern.

We enhanced our scheme to overcome this drawback by allowing hyperboxes to selectively overlap. Specifically, we allow hyperboxes to over-

lap if the problem domain so demands it. This helps in determining nonexclusive classes; it also allows our algorithm to handle "nearby classes."

Consider a scenario where a pattern gets associated with the wrong class, say Class 1, merely because of its proximity to members of Class 1 that were in the training samples rather than to members of its characteristic class (Class 2). This situation can arise from a training set that has a larger incidence of Class 1 than Class 2 patterns or from a nonuniform sampling (since we make no prior assumption on the sampling distribution). In such a case, our scheme offers an additional parameter that gives us the ability to make a soft decision by which we can associate a pattern with more than one class. In essence, this parameter is a threshold on the degree of membership, above which we declare a pattern to be a member of a class.

Another requirement of collaborative Internet-based systems is *clustering*, which automatically groups agents according to some notion of similarity. For example, we can automatically group Agents 1 and 3 as most suited to problems of Type *X* and Agents 2 and 4 as most suited to problems of Type *Y*. Thus, *X* and *Y* are clusters that have been inferred to contain 'samples' that subscribe to a common notion of similarity.

We have proposed a multiresolution scheme, similar to computer and human vision,[17] to partition the data into clusters. The basic idea is to look at the clustering process at differing levels of detail (resolution). For clustering at the base of a multilevel pyramid, we use Simpson's clustering algorithm.[18] This is looking at the data at the highest resolution. Then we operate at different zoom/resolution levels to obtain the final clusters. At each step up the pyramid, we treat the clusters from the level below as points in the higher level. As we go up the hierarchy, therefore, we view the original data with decreasing resolution. This approach has led to encouraging results from clustering real-world data sets,[14] including the Pythia agent data set described here.

In this article, we confine ourselves to the original problem of classifying agents into known, predefined categories, using our neuro-fuzzy algorithm.

Strategies for Learning

An interesting question arises as to where to locate the learning algorithm. In our current implementation, we assume a special agent (called Pythia-C) whose main purpose is to learn this mapping. Pythia-C operates our neuro-fuzzy scheme to model the mapping from a partial differential equation to an appropriate Pythia agent for that problem. We chose such a "master agent" purely for demonstration purposes as it reveals the learning aspect of the agents, as distinguished from their PDE problem-solving capabilities. In practice, this capability could be integrated into every Pythia agent, and any one of them could potentially serve as an appropriate "server" for PDE problems. Thus, while other Pythia agents provide information about PDE problem solving, the purpose of Pythia-C is to direct queries about PDE problems to appropriate Pythia agents.

Pythia agents can collaborate over a network to recommend computing resouces.

Pythia-C can be in either a learning mode or a stable mode. During the learning mode, Pythia-C asks all other known agents for solutions to any problem that is presented to it. It collects all the answers, then chooses the best result as the solution (using the epistemic utility technique described in the sidebar). In effect, Pythia-C uses what has been described as "desperation-based" communication.[19]

For example, if we have six agents in our setup, Pythia-C would ask each one to suggest a solution strategy for a given problem, solicit answers, and assign reasonableness values. It would then recommend the strategy suggested by the agent with the highest reasonableness value. While in this mode, it is also learning the mapping from a problem to the agent that gave the best solution. After this period, Pythia-C has learned a mapping that describes which agent is best for a particular type of problem.

From this point on, Pythia-C switches to what we call *stable mode*. In other words, it will ask only the best agent to answer a particular question. If Pythia-C does not believe an agent has given a plausible solution, it will ask the next-best agent, until all agents are exhausted. This is facilitated by our neuro-fuzzy learning algorithm. By varying the threshold in the defuzzification step, we can get an enumeration of "not so good" agents for a problem type. If Pythia-C determines that no plausible solution exists in itself or among its agents, then Pythia-C will give the answer that was "best" and notify the user of its lack of confidence in the answer.

While this scheme serves most purposes, the switch between learning and stable modes remains an open issue. Pythia-C switches from learning to stable mode after an a priori fixed number of problems have been learned. The timing of the reverse switch from stable to learning mode is a more interesting problem that we chose to attack by three different methodologies.

- *Time based.* Pythia-C reverts to learning mode after a fixed time period in stable mode.
- *Reactive.* A Pythia agent sends a message to Pythia-C whenever its confidence for some class of problems has changed significantly. Pythia-C can then choose to revert to learning mode when it next receives a query about that type of problem.
- *Time-based reactive.* At fixed intervals, Pythia-C sends out a message asking if anyone's capabilities have changed significantly and switching to learning mode if it received a positive response.

EXPERIMENTAL RESULTS

We applied these learning strategies to a collection of networked Pythia agents. Our current implementation deals with finding suitable agents for selecting methods to solve elliptic partial differential equations. It assumes that several agents exist, each with specialized expertise about one or more classes of PDEs. The agent is required to suggest the best solution to the problem in this context.

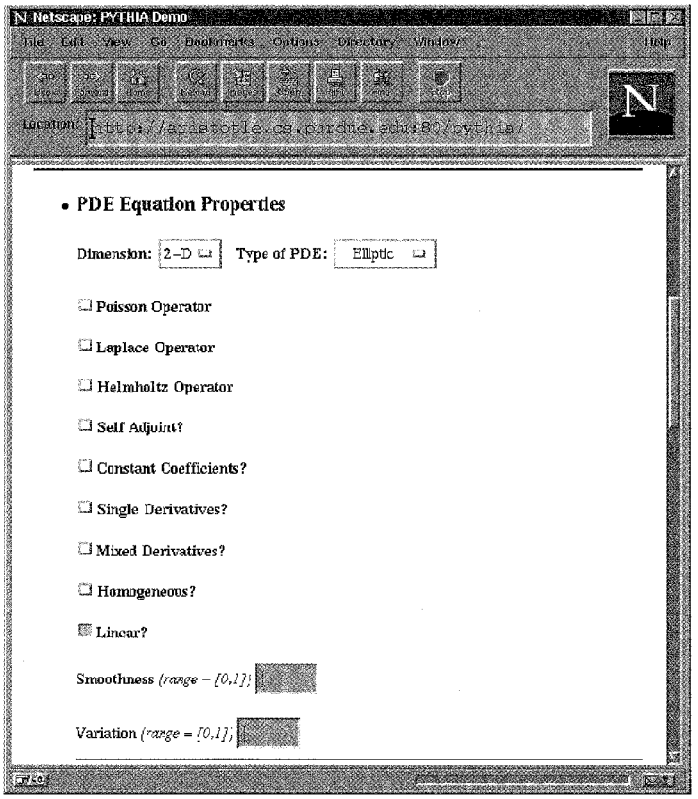

Figure 3. Web session with Pythia-C input screen. The questions address the physical and mathematical details of the PDE problem.

Figure 3 shows an example session where the user specifies the details of the PDE problem. Figure 4 shows the recommendations from the Pythia-C agent.

An experimental version of this system can be accessed online from the Pythia project home page at http://www.cs.purdue.edu/research/cse/pythia.

We have restricted the case study reported here to a representative class of PDEs—namely, linear, second-order PDEs. Our initial problem population consisted of 56 of these PDEs, and we parameterized 42 of them so that the actual problem space consists of more than 250 problems. Many of these PDEs come from real-world problems; others were created to exhibit "interesting" characteristics. Figure 5 presents an example PDE.

We used 167 of the 250 problems in this case study and defined six nonexclusive classes from them:

- Solution-Singular: PDEs with solutions that have at least one singularity (six problems).
- Solution-Analytic: PDEs with solutions that are analytic (35 problems).
- Solution-Oscillatory: PDEs with solutions that show oscillatory behavior (34 problems).
- Solution-Boundary-Layer: PDEs with a boundary layer in their solutions (32 problems).
- Boundary-Conditions-Mixed: PDEs that have mixed boundary conditions in their solutions (74 problems).
- Special: PDEs with solutions that do not fall into any of the above classes (10 problems).

Note that the total number of classified exemplars exceeds 167 because they are not mutually exclusive, and one PDE can therefore belong to more than one class.

Experimental Setup

We divided the set of 167 problems into two parts. The first part contained 111 exemplars (henceforth, we refer to this as the larger training set); the second part contained 56 exemplars (hence, the smaller training set). We created four scenarios with six, five, four, and three Pythia agents respectively. Table 1 presents exact information about these scenarios.

In the scenarios, each Pythia agent "knows" about

one or more classes of PDE problems. For example, in the case of six Pythia agents, each agent knew about one of the six problem classes defined above. Our experiments consisted of two main kinds:

- All the Pythia agents have fixed, a priori knowledge. In other words, each agent's expertise includes all information about the problem classes it is supposed to know about.
- Each agent starts with a small fraction of its representative PDE knowledge base, and the base is refined over time. For example, Agent 2 (whose expertise is "solution-analytic"), starts with information about nine problems and progressively improves with time to the full set of 35 problems that are in this class.

We refer to the first kind as the *static* case, requiring just a one-shot mapping to the knowledge base of the various agents. The second kind, the *dynamic* case, requires the agents to support a switch between learning and stable modes as appropriate. The switch can itself be implemented in a variety of ways; we concentrate here on the time-based, reactive, and time-based reactive methodologies. The experimental setup also includes a "central agent," Pythia-C, as described earlier.

Static Scenario

In the static scenario, the expertise of each agent is fixed to equal the cardinality of the classes. Pythia-C starts off with no knowledge about which agent is suitable for which PDE problem. To determine this information, Pythia-C uses a flooding technique, shooting out queries about each problem to each agent in the setup and soliciting recommendations from them. The following is an example query:

```
(tell   :ontology Recommender
        :language Pythia-Talk
        :reply-with Problem-22750
        :sender Pythia-C
        :receiver Pythia-Agent1
        :content(select_solution_scheme
        (2 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0.5 0.5 1
        0.5 0.5 0 1 0 0 1 1 0 0 0 0.5 0.5 0.5 0.5))
)
```

This message implies that a Pythia-C agent is sending a request to one of the agents in the setup—namely, Pythia-Agent1—asking it to select a solution scheme for a PDE problem with the characteristic vector given by the sequence of numbers. This sequence is interpreted in the context of the Pythia-Talk language and the Recommender ontology. The Pythia-Agent1 is given the Problem-22750 tag to communicate an answer.

Similarly, the following is an example output from Pythia-Agent 1:

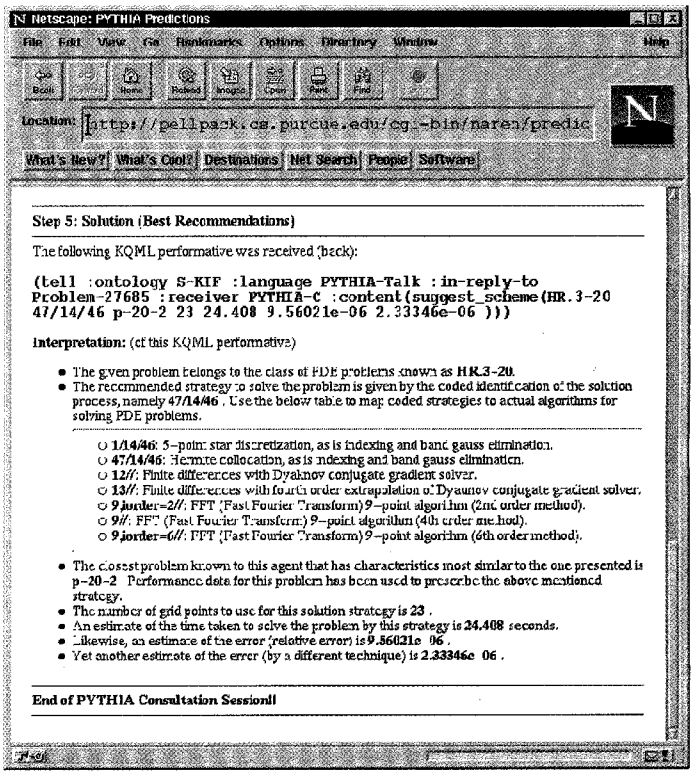

Figure 4. Web session with C-Pythia output screen. The system contacts the appropriate Pythia agent and suggests appropriate software resources and parameters. It also points to the location of the software via the NIST GAMS repository.

Problem #28	$(wu_x)_x + (wu_y)_y = 1$ 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 α = 1, 10, 100. Strong wave fronts for $\alpha >> 1$.
Parameter	α adjusts size of discontinuity in operator co-efficients which introduces large, sharp jumps in solution.

Figure 5. Example problem from the case study population of 167 linear, second-order PDEs.

Table 1. Scenario descriptions in the experimental setup.

Scenario	No. of Agents	Description
1	6	One agent for each class.
2	5	Same as Scenario 1 except that Solution-Boundary-Layer and Boundary Conditions-Mixed classes are the combined expertise of one agent.
3	4	Same as Scenario 2 except that Solution-Singular and Solution-Oscillatory classes are the combined expertise of one agent.
4	3	Same as Scenario 3 except that Solution-Analytic and Special classes are the combined expertise of one agent.

```
(tell   :ontology Recommender
        :language Pythia-Talk
        :in-reply-to Problem-22750
        :receiver Pythia-C
        :sender Pythia-Agent1
        :content(suggest_scheme
        (HR.3-20 47/14/46 p-20-2 23
        24.408 9.56021e-06 2.33346e-06 ))
)
```

In this case, Pythia-Agent1 suggests the solution strategy described in the Pythia-talk sequence beginning HR.3-20.

Each of these recommendations is given a reasonableness value. Pythia-C then chooses the best agent as the one having the highest reasonableness value, as described earlier, and this information forms one exemplar to the Pythia-C setup.

In this way, mappings of the form (problem, agent) are determined for all problems in each of the four agent scenarios.

We conducted two sets of experiments. In the first, we trained Pythia-C on the larger training set of problem-agent pairs and then tested our learning on the smaller training set of exemplars. In effect, the smaller training set formed the test set for this experiment. In the second experiment, the roles of these two sets were reversed.

We also compared our technique with two very popular gradient-descent algorithms for training feedforward neural networks, namely, Vanilla (Plain) backpropagation (BProp)[20] and Resilient Propagation (RProp).[21] Figure 6 summarizes the results.

These results demonstrate that all the learning techniques were adequate in the sense that the Pythia-C agent could find the best agent for each problem class with greater than 90 percent accuracy. Performance using the larger training set was expectedly

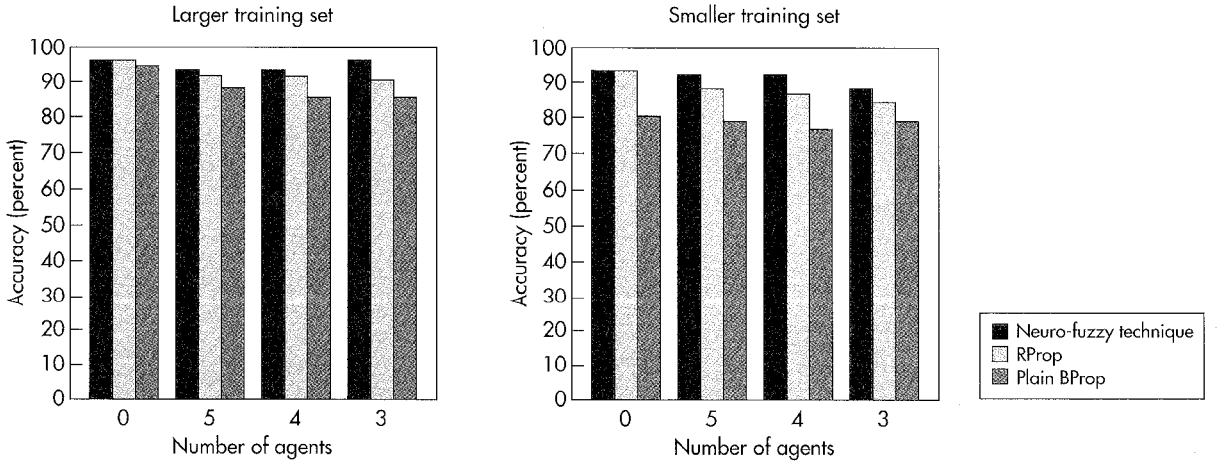

Figure 6. Results of learning. The graph on the left depicts results with the larger training set; on the right, values for the smaller training set. Accuracy figures for the four scenarios are presented for three algorithms: the neuro-fuzzy technique, vanilla back propagation, and resilient propagation.

better than that on the smaller training set.

It can also be seen that our neuro-fuzzy method consistently outperforms BProp and RProp in learning the mapping from problems to agents. Moreover, our algorithm (housed in Pythia-C) operates in an online mode. New data do not require retraining on the old, as they do for many other learning algorithms, including RProp and BProp.

For the larger training set, we incrementally trained our algorithm on the 111 exemplars; the accuracy rose steadily to the values depicted in Figure 6. For example, in the six-agent scenario, we plotted the increase in number of problems seen by Pythia-C and the corresponding decrease of error. Figure 7 shows this incremental increase in the hyperboxes utilized by the neuro-fuzzy technique housed in Pythia-C on the left and the steady decrease in errors on the right. (The hyperboxes are added incrementally to better represent the problem space characteristics.) The number of hyperboxes created for this scenario was 53.

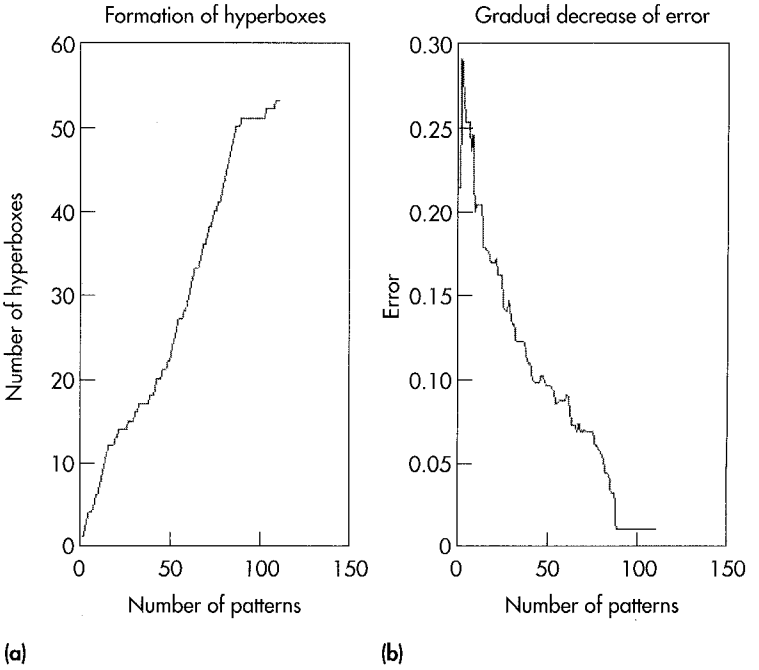

Figure 7. Results of using the neuro-fuzzy technique to train the six-agent scenario online. Results are for the larger training set. The graph on the left shows the incremental formation of hyperboxes; on the right, the corresponding decrease in errors.

Dynamic Scenario

In a collaborative networked scenario, where the resources change dynamically, the neuro-fuzzy technique's online mode of operation lets us automatically infer the capabilities of multiple Pythia agents. In this scenario, the abilities of the agents are assumed to change dynamically and the question is to decide when Pythia-C should switch from a stable mode to a learning mode of operation.

We experimented implementing this switch in time-based, reactive, and time-based reactive schemes.

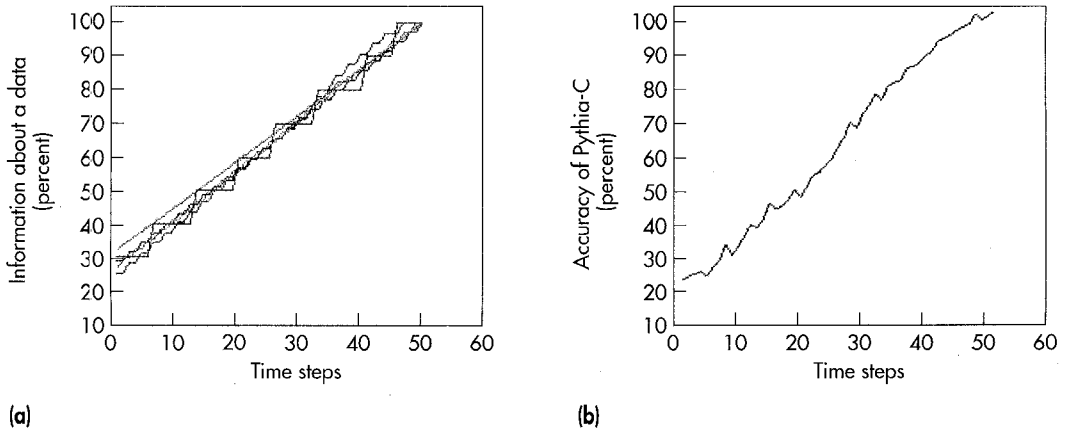

Figure 8. Time-based scheme results for six agents with the larger training set. The graph on the left shows the periodic increase in agent abilities to give correct recommendations; the graph on the right shows the corresponding improvement in accuracy of Pythia-C.

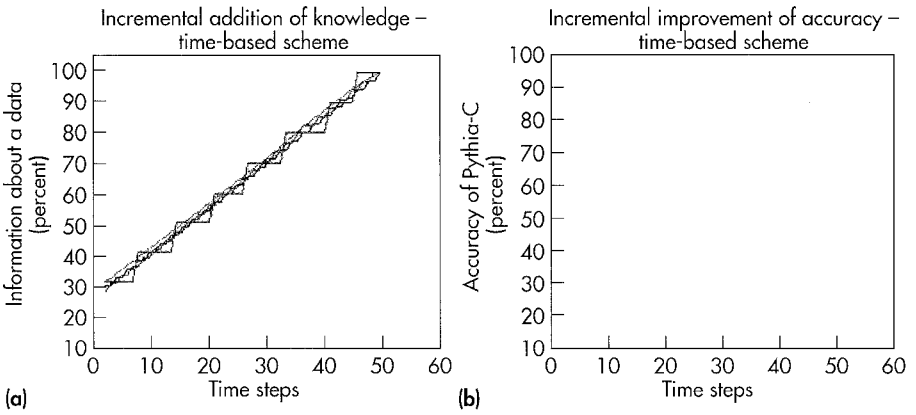

Figure 9. Time-based scheme results for five agents with the larger training set. The graph on the left shows the periodic increase in agent abilities to give correct recommendations; the graph on the right shows the corresponding improvement in accuracy of Pythia-C.

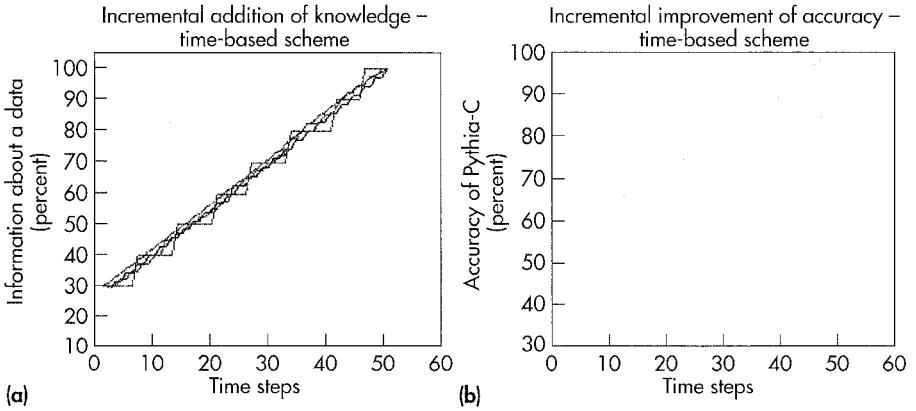

Figure 10. Time-based scheme results for four agents with the larger training set. The graph on the left shows the periodic increase in agent abilities to give correct recommendations; the graph on the right shows the graph on the right shows the corresponding improvement in accuracy of Pythia-C.

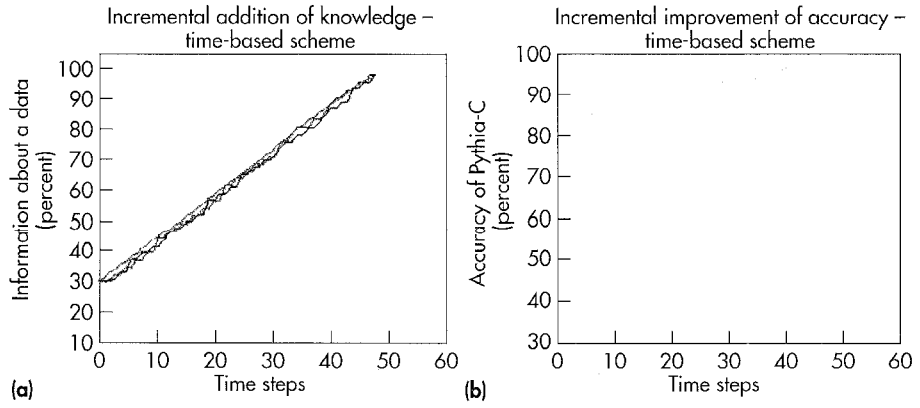

Figure 11. Time-based scheme results for three agents with the larger training set. The graph on the left shows the periodic increase in agent abilities to give correct recommendations. the graph on the right shows the corresponding improvement in accuracy of Pythia-C.

Time-Based. In the time-based scheme, Pythia-C reverts to learning mode at periodic time steps. At such points, Pythia-C cycles through its training set, sends queries to other agents, gets back answers, determines reasonableness values, and finally learns new mappings for the PDE problems. This might involve adding or modifying hyperboxes in the fuzzy min-max network.

Figures 8, 9, 10, and 11 depict results with this scheme for the four agent scenarios on the larger training set. After Pythia-C cycles through the larger training set with each of the agents in the setup, the performance is measured with the smaller training set. (For the sake of brevity, we are not illustrating the figures with the smaller training set. The graphs from this set of experiments are very similar to those with the larger training set except that Pythia-C's learning converges to smaller accuracy figures.)

As Figure 8 through 11 show, the mapping steadily improves for the large training set in all cases to the accuracy observed in the static set of experiments.

Figure 12 charts the accuracy percentages for all the agent scenarios with the time-based reactive scheme. The steepest line is the results for the maximum number of agents (in this case, six). The top line is for three agents. As expected, the scenarios with fewer agents start off at a higher accuracy level and converge to the levels presented earlier in Figure 6. This is because the expertise of the individual agents is combined in the first scenario. Pythia-C's accuracy improves from 40.85 to 98.20 percent in scenario 1, from 59.15 to 98.20 percent in scenario 2, from 59.28 to 98.20 percent in scenario 3, and from 68.86 to 98.20 percent in scenario 4. (Note that these patterns hold true for all three dynamic schemes; the only difference is the rate of progress toward these values).

We conducted yet another series of experiments with the time-based scheme. Rather than having each agent available with approximately a third of its knowledge as a starting situation, we began with no "known" agents in the setup. In other words, Pythia-C did not know about the existence of any agents or their capabilities. Then each agent was introduced into the scenario with a small knowledge base and their abilities were slowly increased.

For example, Agent 1 joins the group with a small knowledge base and announces its existence to Pythia-C. Pythia-C reverts to learning mode and learns mappings from PDE problems to agents (in this case, there is only one agent in the group). After some time, Agent 3 joins the group and this process is repeated. While new agents are being added, the abilities of existing agents (like Agent 1) are also changing. Thus, these two events happen in parallel: New agents are added and the knowledge bases of existing agents are refined. Because our neuro-fuzzy technique can introduce new classes on the fly, Pythia-C handled this situation well, and the accuracy percentages converged to the values obtained for the static case shown in Figure 6.

Reactive. In the reactive scheme, Pythia-C waits for other agents to broadcast information if their confidence for some class of problems has changed significantly. As with the time-based scheme, each agent started the experiments with the same level of ability and its expertise was slowly increased. Because each agent indicates this change of expertise to Pythia-C, the latter reverts to learning mode appropriately. Thus, the accuracy percentages

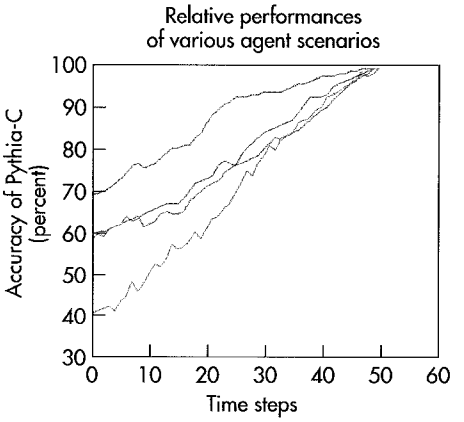

Figure 12. Relative accuracies observed in the four scenarios for the time-based dynamic scheme with the larger training set.

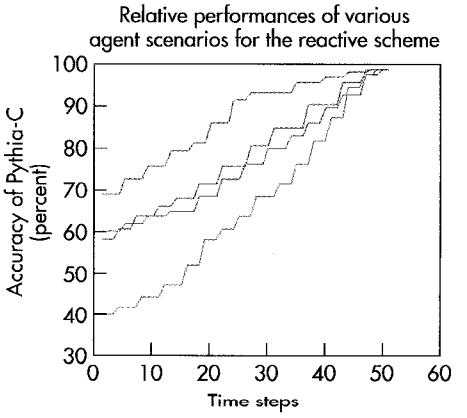

Figure 13. Relative accuracies observed in the four scenarios for the reactive scheme with the larger training set.

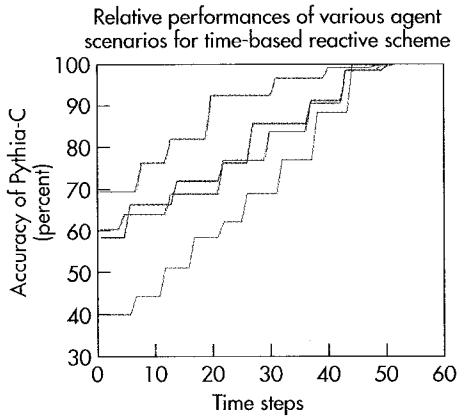

Figure 14. Relative accuracies observed in the four scenarios for the time-based reactive scheme with the larger training set.

approach the same values as in the time-based scheme but follow a more progressive pattern, in tune with the pattern of increase in the abilities of the other Pythia agents.

These curves begin to resemble a "staircase" pattern, as shown in Figure 13.

Time-Based Reactive. In the time-based reactive scheme, we use a combination of the first two approaches. Pythia-C sends out a message at periodic intervals asking if any agent's abilities has changed significantly, and switches to learning mode if it receives an affirmative reply. As in the other two schemes, each agent started with the same level of ability, and its expertise was slowly increased. It was observed that the accuracy percentages follow an even more stable pattern of improvement because Pythia-C waits for other agents to signal change in expertise but only at regular intervals.

Figure 14 summarizes the accuracy curves for all four agent scenarios.

CONCLUSIONS

The results from our experiments with the various schemes show that the techniques we propose enable an agent to keep track of the dynamic capabilities (in this case, the knowledge bases) of other agents. The results also show that our approach can handle situations where agents appear and disappear over time.

Our approach has two major strengths. First, since our basis is a "soft" computing scheme (fuzzy logic), it can handle the uncertainties and imprecisions inherent in this task. For example, when asking the question, "Which agent or agents should be consulted for a problem of type A," the neuro-fuzzy technique naturally incorporates the fact that different agents will be able to "best" answer to a different degree.

Second, our approach provides for "single pass" learning, which means that the learning time is fairly short and can therefore keep up with rapid changes. Even though the initial experiments reported here are with a small set of agents, we see no significant computational burden when the system is scaled up.

At present, we have seen no significant differences between the reactive and time-based approaches in disseminating updates. However, we feel that further experiments with a larger number of agents are needed to establish if one of these schemes is actually better than the other. We are presently conducting such experiments with our schemes in the context of both Pythia and Web Personalization Systems. We note here that while we have used Pythia in particular, and networked scientific computing in general, as the framework in which to develop our multiagent recommender system, the techniques we have developed are applicable to any multiagent system on the network.

In a full implementation, Pythia-C would learn of the entry of a new Pythia agent by querying the ANS. In our experimental studies, however, we used an abstraction of this dynamic network resource querying; Pythia-C is told of a new agent by adding the appropriate entry in its learning input.

In ongoing work, we are adding facilities for collaborators to automatically add Pythia agents (consistent with our specifications) and for Pythia-C to become aware of them via the ANS.

We have also recently coupled Pythia agents with the GAMS index of mathematical software repositories.[1] However, the integration between Ellpack, GAMS, and Pythia is still not perfect. We presently use the Shade/KQML API implementation of KQML from Lockheed, but ongoing efforts will transition this to **JATLite** from Stanford.

The other platform-dependent component of our recommender system is the CLIPS system. However, a Java-based replacement, called JESS (Java expert system shell),[22] is being built for CLIPS. Thus, the entire Pythia system will be Java based and, hence, platform independent.

REFERENCES

1. E. Gallopoulos, E.N. Houstis, and J.R. Rice, "Computer as Thinker/Doer: Problem-Solving Environments for Computational Science," *IEEE Computational Science and Engineering*, Vol. 1, No. 2, 1994, pp. 11-23.
2. E. Hostis et al., *MPSE: Multidisciplinary Problem Solving Environments*, white paper, CIC Forum on America in the

Age of Information, Nat'l Science and Technical Council, Washington, D.C., 1995; available online at http://www.hpcc.gov/cic/forum/CIC_Cover.html.
3. T. Drashansky, A. Joshi, and J.R. Rice, "SciAgents: An Agent Based Environment for Distributed, Cooperative Scientific Computing," *Proc. Seventh Int'l Conf. Tools with Artificial Intelligence*, IEEE Computer Society, Los Alamitos, Calif., 1995, pp. 452-459.
4. M. Mu and J.R. Rice, "Modeling with Collaborating PDE Solvers: Theory and Practice," *Computing Systems in Engineering*, Vol. 6, 1995, pp. 87-95.
5. A. Joshi et al., "Multiagent Simulation of Complex Heterogeneous Models in Scientific Computing," *IMACS Math. Comp. Simulation*, Vol. 44, pp. 43-59, 1997.
6. E.N. Houstis and J.R. Rice, "Parallel ELLPACK: A Development and Problem Solving Environment for High Performance Computing Machine," In *Programming Environments for High Level Scientific Problem Solving*, North Holland, 1992, pp. 229-243.
7. J.R. Ric, "Processing PDE Interface Conditions," Technical Report TR-94-041, Dept. Comp. Sci., Purdue Univ., 1994.
8. T.T. Drashansky and J.R. Rice, "Processing PDE Interface Conditions II," Technical Report TR-94-066, Dept. Comp. Sci., Purdue University, 1994.
9. S. Weerawarana et al., "Pythia: A Knowledge-Based System to Select Scientific Algorithms," *ACM Trans. Math. Software*, Vol. 22, No. 4, 1996, pp. 447-468.
10. J.R. Rice, "Methodology for the Algorithm Selection Problem," In *Performance Evaluation of Numerical Software*, L. Fosdick, ed.,North Holland, 1979, pp. 301-307.
11. P. Resnick and H. Varian, "Recommender Systems," *Comm. ACM*, Vol. 40, No. 3, Mar. 1997, pp. 56-58.
12. J. C. Giarratano, *CLIPS User's Guide, Version 5.1*, NASA Lyndon B. Johnson Space Center, 1991.
13. R. Fritzson et. al., "KQML—A Language and Protocol for Knowledge and Information Exchange," *Proc. 13th Int'l Distributed Artificial Intelligence Workshop*, 1994.
14. A. Joshi et al. "Neuro-Fuzzy Support for PSEs: A Step Toward the Automated Solution of PDEs," *IEEE Computational Science & Engineering*, Vol. 3, No. 1, 1996, pp.44-56.
15. A. Joshi et al., "A Neuro-Fuzzy Approach to Agglomerative Clustering," *Proc. Int'l Conf. Neural Networks*, Vol. 2, IEEE Press, Piscataway, N.J., 1996, pp. 1,028-1,033.
16. P.K. Simpson, "Fuzzy Min-Max Neural Networks—Part I: Classification," *IEEE Trans. Neural Networks*, Vol. 3, Sep. 1992, pp. 776-786.
17. J.M. Jolion, "A Pyramid Framework for Early Vision," Kluwer Int'l Series in Eng. and Computer Science, 1994.
18. P.K. Simpson, "Fuzzy Min-Max Neural Networks—Part II: Clustering," *IEEE Trans. Fuzzy Systems*, Vol. 1, No. 1, Feb. 1993, pp. 32-34.
19. Y. Lashkari, M. Metral, and P. Maes, "Collaborative Interface Agents," *Proc. AAAI 94*, AAAI, 1994.

URLs for this article

GAMS • gams.nist.gov/
Globus • www.globus.org/
JATLite • java.stanford.edu/java_agent/html/
Legion • www.cs.virginia.edu/~legion//
MPSEs • www.crcs.missouri.edu/~joshi/sciag/
Net//Ellpack • pellpack.cs.purdue.edu/netpp/
Netlib • www.netlib.org/
NetSolve • www.cs.utk.edu/netsolve/
Pythia • www.cs.purdue.edu/research/cse/pythia/
Web//Ellpack • pellpack.cs.purdue.edu/

20. D.EE. Rumelhart and J.L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press, Cambridge, Mass., 1986.
21. H. Braun and M. Riedmiller, "Rprop: A Fast and Robust Backpropagation Learning Strategy," *Proc. ACNN*, 1993.
22. M. Watson, *Intelligent Java Applications for the Internet and Intranets*, Morgan Kaufman, San Francisco, 1997.

Anumpam Joshi is an assistant professor of computer engineering and computer science at the University of Missouri. He obtained his PhD from Purdue University in 1993, and his bachelor's degree from IIT-Delhi in 1989. His research interests relate to intelligent networked systems, specifically networked scientific computing, mobile computing, and content-based retrieval from networked multimedia repositories. He is a member of IEEE, IEEE Computer Society, ACM, and UPE.

Naren Ramakrishnan obtained his PhD in computer sciences from Purdue University in 1997. In August 1998, he will join Virginia Polytechnic Institute and State University as an assistant professor of computer science. His research interests include recommender systems, problem solving environments, computational science and data mining. He is a member of IEEE, IEEE Computer Society, ACM, and ACM SIGART.

Elias N. Houstis is a professor of computer science and director of the computational science and engineering program at Purdue University. His research interests include problem-solving environments; parallel, neural and mobile computing; and distributed learning. He received a PhD in mathematics from Purdue University and a BS in mathematics from the University of Athens. He is a member of IMACS, Sigma Xi, and the IFIP WG 2.5 in Numerical Software.

Readers may contact the authors through their Web sites at http://www.cecs.missouri.edu/~joshi/, http://www.cs.purdue.edu/homes/ramakris, and http://www.cs.purdue.edu/people/enh.