1997

MultiAgent Systems to Support Networked Scientific Computing

Anupam Joshi

N. Ramakrishnan

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

Report Number:
97-021

MULTIAGENT SYSTEMS TO SUPPORT NETWORKED SCIENTIFIC COMPUTING

Anupam Joshi
Narendran Ramakrishnan
Elias N. Houstis

CSD-TR 97-021
March 1997

MultiAgent Systems to support Networked Scientific Computing

Anupam Joshi
Department of Computer Engineering and Computer Science
University of Missouri, Columbia, MO 65211
email: joshi@ece.missouri.edu
and
N. Ramakrishnan, E.N. Houstis
Department of Computer Sciences
Purdue University, West Lafayette, IN 47907-1398
email: {naren,enh}@cs.purdue.edu

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. High Performance Computing & Communication (HPCC) Systems facilitate this scenario. This paper explores the use of *advisory agents* to enable harnessing the power of (inter)networked computational resources (agents) to solve scientific computing problems.

Many hitherto dormant and difficult challenges in applied sciences, such as modeling protein folding or internal combustion engine design, have become feasible to attack using the power of HPCC. 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 the HPCC. It allows us to use the high performance communication infrastructure (vBNS, Internet II etc.) to view heterogeneous networked hardware (including specialized high performance resources such as the proposed terraflops machines) and software (e.g. specialized solvers, databases of material properties, performance measuring systems) resources as a single "meta computer"[11](http://www.cecs.missouri.edu/joshi/sciag/). 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 problems, "lifecycle simulation" is the operative keyword. 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. The scientific computing software of tomorrow, developed with such applications in mind, will use software agent based techniques to build systems from software components which run on heterogeneous, networked platforms. It will allow wholesale reuse of legacy software and provide a natural approach to parallel and distributed problem solving.

Yet, for all its potential payoffs, the state-of-the-art in Scientific Computing systems is woefully inadequate in terms of ease of use. Take, for example, parallel computing. Diane O'Leary in a recent article[22] compared parallel computing of today to the "prehistory" of computing, where computers were used by a select few who understood the details of the architecture and operating system, where programming was complex, and debugging required reading hexadecimal dumps. Computer time had to be reserved, jobs were submitted in batches, and crashes were common. Users were never sure of whether an error was due to a bug in their code or in the system. One can safely argue that if parallel processing is in its prehistory, then networked scientific computing (NSC) is probably in the mesozoic era. Clearly, if the NSC based computational paradigm for the scientific process is to succeed and become ubiquitous, it must provide the simplicity of access similar to the point and click capability of networked information resources like the web.

This means that an infrastructure needs to be developed to allow scientific computing applications to use resources and services from many sources spread across the (inter)network. The system, however, needs to be network and evolution transparent to the user. In other words, the user should be presented an abstraction 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/bug fixes to the software components etc. should be handled at the system level with minimal user involvement. The first part of the problem can be handled by creating *advisory agents* that accept a problem definition and some performance/success criteria from the user, and suggest software components and hardware resources that can be deployed to solve this problem. This is very similar in substance to the idea of *recommender systems* that is being mooted for harnessing distributed information resources.

While the problem has been identified in the networked information resources scenario, and initial research done[24], the problem remains largely ignored for the domain of networked computational resources. Note that the problem is different from invoking a known method remotely on some object, for which a host of distributed OO techniques are being deveoped and proposed. To realize the need for such an advisory system, 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/). Software such as GAMS[1](http://gams.nist.gov/) exists which enables users to identify and locate the right class of software for their problem. However, the user has to identify the software most appropriate for the given problem, download the software and its installation and use instructions, install the software, compile and (possibly port) it, and learn how to invoke it appropriately. Clearly this is a non-trivial task even for a single piece of software, and can be enormously complex when multiple software components need to be used. Using the networked resources of today is the modern day equivalent of programming ENIAC, which required direct manipulation of connecting wires. Research is needed to provide systems which will abstract away the detail of the underlying networked system from the user, who should be able to interact with this system in the application domain. This is where PSEs with inherent "intelligence" come in.

A Problem Solving Environment is a computer system that provides the user with a high level abstraction of the complexity of the underlying computational facilities. The design objectives and architecture of PSEs are described in [29]. It provides all the computational facilities necessary to solve a target class of problems[6]. These facilities include advanced solution methods, automatic or semiautomatic selection of solution methods, and ways to easily incorporate novel solution methods. Moreover, PSEs use the language of the target class of problems and provide a "natural" interface, so users can use them without specialized knowledge of the underlying computer hardware or software. The user can not be expected to be well versed in selecting appropriate numerical, symbolic and parallel systems, along with their associated parameters, that are needed to solve a problem. Nor can s/he be expected to be aware of all possible software components and hardware resources that are available across the network to solve a problem. An important task of a PSE is to accept some "high level" description of the problem from the user, and then automatically locate and select the appropriate computational resources (hardware, software) needed to solve the problem. Clearly, this task requires the use of "intelligent" techniques - it requires knowledge about the problem domain and reasoning strategies. The purpose of our research is to address the issue of intelligence in the general networked scientific computing domain. Specifically, we have developed *advisory systems* for the class of applications that can be described by mathematical models involving Partial Differential Equations(PDEs).

The numerical solution of partial differential equation models 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 numerical solvers (software) for PDEs. These solvers normally require a number of parameters that the user must specify, 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. With the heterogeneity of machines available across the network to the PSE, including parallel machines, an additional decision that has to be

made is regarding on which machine and with what configuration should the problem be solved. 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. On the other hand, the numerical solution must satisfy several objectives, primarily involving error and hardware resource requirements. It is unrealistic to expect that engineers and scientists will or should have the deep expertise to make "intelligent" selections of both methods and their parameters plus the selection of computational resources that will satisfy their computational objectives. Thus, we propose to build a multiagent advisory system to help the user make the optimal, or at least satisficing, selections.

In this paper, we start by describing some related research on "expert systems" for scientific computing, and briefly introduce the PYTHIA project, an intelligent advisory system that performs automatic algorithm selection in well defined scientific domains using hybrid (neuro-fuzzy and bayesian) techniques. We then describe a scenario where several PYTHIA agents exist on the network, each containing a part of the total knowledge corpus. We show how they can interact to share their knowledge, and illustrate how neuro–fuzzy techniques help to identify the most "reasonable" agent to seek advice from about a particular problem. While it is not the focus of this paper, we are at present experimenting with gathering data about the amount of traffic our multiagent systems generate, and how our learning and adaptation schemes can help to reduce it.

2. RELATED WORK

We have remarked earlier on systems such as Netsolve, GAMS, Web //Ellpack etc. which are being created to realize the idea of networked scientific computing. We now briefly describe some attempts at developing intelligent systems for assisting in various aspects of the PDE solution process. In[25], 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 [21], 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. The refinement process is guided by various "refinement indicators" and refinement is affected by one of three mesh enrichment strategies. At the other end of the PDE solution process, expert systems can be used to guide the internals of a linear system solver[17]. This particular expert system applies self-validating methods in an economical manner to systems of linear equations. Other expert systems that assist in the selection of an appropriate linear equation solver for a particular matrix are also currently being developed. In [3; 4], Dyksen and Gritter describe an expert system for selecting solution methods for elliptic PDE problems based on problem characteristics. Problem characteristics are determined by textual parsing or with user interaction and are used to select applicable solvers and to select the best solver. This work differs significantly from our approach, which is based on using performance data for relevant problems as the algorithm selection methodology. Dyksen and Gritter use rules based solely on problem characteristics. We argue that using problem characteristics solely is not sufficient because the characterization of a problem includes many symbolically and a priori immeasurable quantities, and also because practical software performance depends not only on the algorithms used, but on the particular implementations of those algorithms as well. In [16], 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) that the user can use. 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[19]. Our group has also been actively involved in using several techniques, such as neural nets, neuro-fuzzy systems and Bayesian nets ([30; 15; 23; 14; 12; 13]) to address related issues

of classifying PDE problems based on their characteristics, and then using this classification to predict an appropriate solution method for new problems.

3. PYTHIA

PYTHIA[8; 30] is a knowledge based system that helps select scientific algorithms to achieve desired tasks in computation. PYTHIA attempts to solve the problem of determining an 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 techniques involved are general, our current implementations of PYTHIA operate in conjunction with systems that solve (elliptic) partial differential equations (PDEs) [30] and problems in numerical quadrature.

In its current preliminary implementation, a PYTHIA agent accepts as input the description of an elliptic PDE problem, and produces the method(s) appropriate to solve it. Its strategy is similar to that believed to underlie human problem solving skills. There is a wealth of evidence from psychology that suggests that humans compare new problems to ones they have seen before, using some metric of similarity to make that judgment. They use the experience gained in solving "similar" previous problems to evolve a strategy to solve the present one. This same strategy has been defined as case based reasoning in the AI literature. In effect, the strategy of PYTHIA is to compare a given problem to the ones it has seen before, and then use its knowledge about the performance characteristics of prior problems to estimate those of the given one. 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 the following steps

—Analyze the PDE problem and identify its characteristics.

—From the previously solved problems, identify the set of problems similar to the new one.

—Extract all information available about this set of problems and the applicable solvers and select the best method.

—Use performance information of this method to predict its behavior for the new problem.

When deciding on which previously seen problem is the closest to the new one, one could directly compare the new problem with all previously seen problems. Such lookup can be done efficiently using specialized data structures such as multidimensional search trees. Alternatively, one could classify problems into meaningful classes, and use these to find similar problems. This would imply that finding the closest problem would be a two stage process, where one would first find the appropriate class for a new problem, and then look for close problems amongst the members of that class. We will investigate both these strategies in our proposed work. For the case of Elliptic PDEs, we identify five classes of problems that seem to be relevant in choosing a solution method, namely Solution Singular, Solution Analytic, Solution Oscillatory, Solution Boundary Layer, and Boundary Conditions mixed. A sixth class identifies problems which do not belong to any of the above. The classification of problems into subsets and determining which subset a particular problem belongs to can be implemented in several ways. One approach is to use a deterministic membership strategy where a class of problems is represented by the centroid of the characteristic vectors of all the members of that class. Then, class membership is defined as being within a certain radius of the centroid in the characteristic space. To classify a new problem, we compute the distance from it to all centroids. The problem is said to belong to that class whose centroid is closest to it. Distance between characteristics is again used to determine which particular member of a class is closest to the new problem. The technique described above produces classes in a deterministic way and computes a characteristic centroid for each class. Then a new problem is classified by computing its distance from these centroids using some norm. While there are some classes where there is a completely deterministic (and simple) way to determine class membership, for other classes such a priori determination is not possible. We have to determine the class structure based on samples seen thus far. In other words, the class structure has to be learnt given the examples. We will develop appropriate architectures for addressing the learning issue using a variety of "Intelligent" approaches. We propose to experiment with a variety of symbolic, fuzzy, connectionist and hybrid approaches to learning and determine what kind of a learning system works well in this domain. An initial work, describing a moderate sized

comparative study we did was recently published[24]. We also propose to generate performance data for a wider variety of PDEs, and extend the capabilities of the PYTHIA agent in terms of the kinds of problems it can assist a user with.

In this paper, we shall be concerned with the former application domain. The strategy of PYTHIA is to compare a given problem to the ones it has seen before, and then use its knowledge about the performance characteristics of prior problems to estimate those of the given one. Together with a good method to solve a given problem, PYTHIA also provides a factor of confidence that PYTHIA has in the recommended strategy. For example, a typical PYTHIA output for a PDE problem would be: *"Use the 5–point star algorithm with a* 200×200 *grid on an NCube/2 using* 16 *processors: Confidence* – 0.85".

It can be easily seen that the above case–based reasoning strategy does not scale up when the population of problems becomes huge. We, therefore, have a revised two–stage reasoning process wherein PYTHIA first classifies the given problem into one (or more) of pre–defined problem classes, and having thus pruned the search space, uses performance criteria from its database to map from the problem class space to the algorithm space. Due to space limitations, we are unable to dwell on the intelligent mechanisms in the functioning of PYTHIA. The interested reader is referred to [30; 15; 23; 13] for details. As PYTHIA encounters new problems and 'predicts' strategies for solving them, it adds to its Knowledge Base (KB), information about these problems and their solutions and thus, refines its KB.

4. COLLABORATIVE PYTHIA

The underlying model of computation as used by the above mechanism is quite primitive — it is essentially a static paradigm dealing with a single PYTHIA agent. In the networked scenario, several PYTHIA agents exist across the Internet, each (possibly) with a specialized knowledge base about problems. For example, in the domain being discussed here, there are many different types of partial differential equations and most scientists tend to use only a limited kind. As such, any PYTHIA agent they are using will gather information about and be able to answer questions effectively only about a limited range of problems. If there were mechanisms that allowed PYTHIA agents of various application scientists to collaborate, then each agent could share its knowledge to others and potentially solve a broader range of problems. This forms the main focus of our paper. We describe how we make PYTHIA a collaborative multi-agent system which operates in the networked environment.

PYTHIA outputs a strategy for solving a given problem and also a quantitative measure of its confidence in the answer. In our collaborative scenario, if an agent discovers that it does not have "enough" confidence in the prediction it is making, it could query all other PYTHIA agents and obtain answers from all of them. These answers are, as mentioned before, suggestions on what resources to use to solve a given problem. It can then pick up one of the suggestions and follow it. This often entails a huge amount of traffic on the system which is not desirable. A better approach is to use the information obtained by the initial flooding type of queries to learn/infer a mapping from a problem to a PYTHIA server, best able to suggest solution resources for the given problem.

As remarked upon earlier, it is likely that some given PYTHIA agent will know a great deal about a certain type of problem. Thus from the answers received by an agent, we want it to learn a mapping from the type of the problem to the agent which is most likely to have a correct answer. In future, it could direct queries more effectively, rather than using a broadcasting technique to seek answers. Additionally, the situation can be dynamic – the abilities of individual PYTHIA agents can be expected to change over time (as they add more problems to their KB), more PYTHIA agents might come into existence etc. Thus any mapping technique utilized should perform efficient classification and yet have the ability to learn *on–line*. This means that assimilating new data should not entail going over previously learned information, so as to avoid long learning periods. Interestingly, this rather useful feature is absent in many popular 'learning' algorithms mooted by the Artificial Intelligence community.

We have developed a neuro–fuzzy method of learning suitable for this purpose. Neural techniques provide methods to model complex relationships and find application in classifying concepts into predefined classes. Fuzzy techniques provide a more natural means to model uncertainty and are useful to represent applications where we talk of membership of concepts in classes to varying degrees. Our technique combines the

advantages of both and is natural for the application considered here. The reason that a fuzzy system finds application in the PYTHIA scenario is that classification of PDE problems is not crisp. In the domain of PDEs, our problem classes are defined based on the properties possessed by the solutions of the PDEs. For instance, the solution of a given PDE could have a singularity, and also show some oscillatory behavior on the boundaries. Thus the given PDE would have membership (to different extent) in the classes representing "solution-singular" and "solution-oscillatory". A conventional, binary membership function would not model this situation accurately. We feel that such fuzziness is inherent in the learning task whenever agents model complex, real world, scientific problems. The neural part is used to learn such fuzzy functions.

In this paper, we use a quantitative measure of reasonableness [10] to automatically generate exemplars to learn the mapping from a problem to an agent. This is needed because the user cannot be expected to have information about the most reasonable resource(s) for a given problem in such a dynamic scenario. To do this in an unsupervised manner, we combine two factors, one which denotes the probability of a proposition q being true, and the other which denotes its utility. Specifically, the reasonableness of a proposition is defined as follows [20]:

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

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 $f(x) = \frac{2}{1+e^{-x}} - 1$. The reason for choosing this expression as the measure of the utility of an agent is that it should reflect the number of problems of the present type that it has seen. The value of an utility function is to measure the amount of knowledge that an agent appears to have. The more the problems of a certain kind in an agent's KB, the more appropriate will its prediction be for new problems of the same type. Hence, we have designed the utility to be a function of N_e. The above formulation is not unique, though. For example, assume that the probabilities of the 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, then the second hypothesis will be assigned a greater reasonableness than the first one (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.

Thus, the goal of the PYTHIA collaborative system is to use the neuro–fuzzy approach to automatically map the most reasonable agent for a particular problem.

5. NEURO–FUZZY APPROACHES

While we have worked on several techniques (our studies have utilized standard statistical methods, gradient descent methods, machine learning techniques and other classes of algorithms), it has been our experience that specialized techniques developed for this domain perform better than conventional off–the–shelf approaches [13]. In particular, our neuro–fuzzy technique infers efficient mappings, caters to mutually non–exclusive classes which characterize real–life domains and learns these classifications in an on–line manner. For the sake of brevity, we provide only those details of this algorithm here that are relevant in the current context. For a more detailed exposition, we refer the interested reader to [23].

Our neuro–fuzzy classification scheme [23; 15] is based on an algorithm proposed by Simpson [27]. The basic idea is to use fuzzy sets to describe pattern classes. These fuzzy sets are, in turn, represented by the 'fuzzy' union of several hyperboxes. Such hyperboxes define a region in n-dimensional pattern space that contain patterns with full-class membership. A hyperbox is completely defined by a min–point and max–point and also has associated with it a fuzzy membership function (with respect to these min-max points).

This membership function helps to view the hyperbox as a fuzzy set and such "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 each pattern sample is "taught" to the neuro-fuzzy system, either an existing hyperbox (of the same class) is expanded to include the new pattern or a new hyperbox is created to represent the new pattern. The latter case arises when we do not have an already existing hyperbox of the same class or when we have such a hyperbox but which cannot expand any further beyond a limit set on such expansions. Simpson's original method assumes that the pattern classes underlying the domain are mutually exclusive and that each pattern belongs to exactly one class. But the pattern classes that characterize problems in many real world domains are frequently not mutually exclusive. For example, consider the problem of classifying geometric figures into classes such as polygon, square, rectangle etc., Note that these classes are not mutually exclusive (i.e., a square is a square and a rectangle and a polygon). It is possible to apply Simpson's algorithm to this problem by first 'reorganizing' the data into mutually disjoint classes such as 'rectangles that are not squares', 'polygons that are not rectangles', and 'polygons' etc., but 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. Also, the only parameter in Simpson's method is the maximum hyperbox size parameter - this denotes the limit beyond which a hyperbox cannot expand to "enclose" a new pattern.

The way we enhance our scheme to overcome this apparent drawback is to allow hyperboxes to 'selectively' overlap. In other words, we allow hyperboxes to overlap if the problem domain so demands it. This, consequently, aids in the determination of non-exclusive classes. It also allows our algorithm to handle "nearby classes" : Consider the scenario when 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). Such a situation can be caused due to a larger incidence of the Class 1 patterns in the training set than the Class 2 patterns or due to a non uniform sampling, since we make no prior assumption on the sampling distribution. In such a case, an additional parameter in our scheme gives us the ability to make a soft decision by which we can associate a pattern with more than one class.

A more interesting and rather 'natural' requirement in collaborative internet based systems is what is known as clustering to 'automatically' (using some criterion) group entities (in this case, agents) into several 'clusters' based on some notion of similarity. For example, in this paradigm, we would be able to automatically 'group' Agents 1 and 3 as most suited to problems of Type X and Agents 2 and 4 to be 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 multi-resolution scheme, similar to computer vision [9], 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 the multi-level pyramid, we use Simpson's clustering algorithm [28]. 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 at this 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 [12; 13], including the PYTHIA agent data set described here.

In this paper, we shall confine ourselves to the original problem of classifying agents into known, predefined categories and as described earlier, we utilize our neuro-fuzzy algorithm to achieve this purpose.

6. STRATEGIES FOR LEARNING

An interesting question arises as to where to 'house' the neuro-fuzzy algorithm that we just described. In our current implementation, we assume an 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 PDE problem to an

appropriate PYTHIA agent for that problem. We have chosen such a 'master agent' purely for demonstration purposes as this brings out the learning aspect of the agents as being distinct from their PDE problem solving abilities. In practice, this ability could be integrated into every PYTHIA agent and potentially, anyone of them could 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 help in directing queries about PDE problems to appropriate PYTHIA agents.

Further, PYTHIA–C can be in either a "learning" mode (LM) or a "'stable mode" (SM). During the LM, PYTHIA–C asks all other known agents for solutions about a particular type of problem. It collects all the answers, and then chooses the best result as the solution. In effect, PYTHIA–C uses what has been described in [18] as "desperation based" communication. For example, if we have 6 agents in our setup, PYTHIA–C would ask each one of these agents to suggest a solution strategy for a given problem, solicit answers and assign reasonableness values based on the formulation above. It would then recommend the strategy suggested by the agent that had the highest reasonableness value. While in this mode, it is also learning the mapping from a problem to the agent which gave the best solution. The best solution in LM is computed by the epistemic utility formulation described earlier. After this period, PYTHIA–C has learned a mapping describing which agent is best for a particular type of problem. From this point on, PYTHIA–C is in the SM. It now switches to what we label as stable communication. In other words, it will only ask 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 an acceptance threshold in the defuzzification step, 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 will give the answer that "was best". When giving such an answer, the user will be notified of PYTHIA–C's lack of confidence.

While this scheme serves most purposes, an issue still pending is the mode of 'switching' between LM and SM. PYTHIA–C switches from LM to SM after an *a priori* fixed number of problems have been learned. The timing of the reverse switch from SM to LM is a more interesting problem that we chose to attack by three different methodologies.

—**Time based:** This is, by far, the simplest approach in which PYTHIA–C reverts to LM after a fixed time period in SM.

—**Reactive:** In this scheme, a PYTHIA agent sends a message to PYTHIA–C whenever its confidence for some class of problems has changed significantly. PYTHIA–C can then chose to revert to LM when it next receives a query about that type of problem.

—**Time based Reactive:** A combination of the two approaches outlined above would have PYTHIA–C send out a "has anyone's abilities changed significantly" message at fixed time intervals, and switch to LM if it received a positive response.

6.1 System Details of a PYTHIA agent

The PYTHIA agents are implemented by a combination of C language routines, shell scripts and systems such as CLIPS (the C Language Integrated Production System) [7].

The agents communicate using the Knowledge Query and Manipulation Language (KQML) [5], using protocol defined performatives. All PYTHIA agents understand and utilize a private language (PYTHIA–Talk) that describes the meaning (content) of the KQML performatives. Examples of KQML performatives exchanging messages in PYTHIA–Talk format are provided in the next section. There is also an Agent Name Server (ANS) in this scenario whose purpose is to determine the URL (Uniform Resource Locator) associated with a particular PYTHIA agent. We use the TCP transport protocol to access the individual PYTHIA agents. In other words, the URL for the agents would have the format `tcp://hostname:portnumber`. When a new PYTHIA agent comes into existence, the URL for this agent can be dynamically registered with the ANS by an appropriate KQML Application Programming Interface call. PYTHIA–C comes to know of this new entry by querying the ANS. In our experimental studies however, we used an abstraction of this dynamic network resource querying. In other words, 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 PYTHIA specifications) and for PYTHIA–C to become aware of them via the ANS.

7. EXPERIMENTAL RESULTS

In this section, we describe the results of applying the above mentioned ideas to a collection of networked PYTHIA agents. Our current implementation deals with finding suitable agents for selecting methods to solve elliptic partial differential equations (PDEs). In other words, it is envisioned that there will exist several agents, each with specialized expertise about a class(es) of PDEs and it is required to determine the agent that can best suggest a solution to the problem at hand. An experimental version of this system can be accessed on–line by using the *demos* link from the WWW URL of the PYTHIA project: `http://www.cs.purdue.edu/research/cse/pythia`. An example of a session where the user specifies the details of the PDE problem is provided in Fig. 1. The recommendations from the PYTHIA–C agent can be seen in Fig. 2.

For this case study, we have restricted our study to a representative class of PDEs (linear, second–order PDEs) and our initial problem population consisted of fifty–six of such PDEs. Of these, 42 are 'parameterized' which leads to an actual problem space of more than two–hundred and fifty problems. Many of these PDEs have been taken from 'real–world' problems while some have been artificially created so as to exhibit 'interesting' characteristics. In this paper, we utilize 167 of these PDEs. An example of a PDE is given in Fig. 3.

From this population of PDEs, we defined the following six non-exclusive classes:

(1) SOLUTION–SINGULAR: PDEs whose solutions have at least one singularity (6 problems).

(2) SOLUTION–ANALYTIC: PDEs whose solutions are analytic (35 problems).

(3) SOLUTION–OSCILLATORY: PDEs whose solutions oscillate (34 problems).

(4) SOLUTION–BOUNDARY–LAYER: PDEs with a boundary layer in their solutions (32 problems).

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

(6) SPECIAL: PDEs whose solutions do not fall into any of the above classes (10 problems).

As can be easily seen from the size of these classes (the total number of exemplars in the above classification is more than 167), they are not 'mutually–exclusive': one PDE can belong to more than one class 'simultaneously'.

7.1 Experimental Setup

This set of 167 problems was divided into two parts – the first part containing 111 exemplars (henceforth, we refer to this as the larger training set) and the second part containing 56 exemplars (hereinafter referred to as the smaller training set). We created four scenarios with 6, 5, 4 and 3 PYTHIA agents respectively. In each case, each PYTHIA agent 'knows' about a certain class(or classes) of PDE problems. For example, in the case where there were 6 PYTHIA agents, each agent knew about one class of problems as described above. Exact information about these scenarios is provided in the table below:

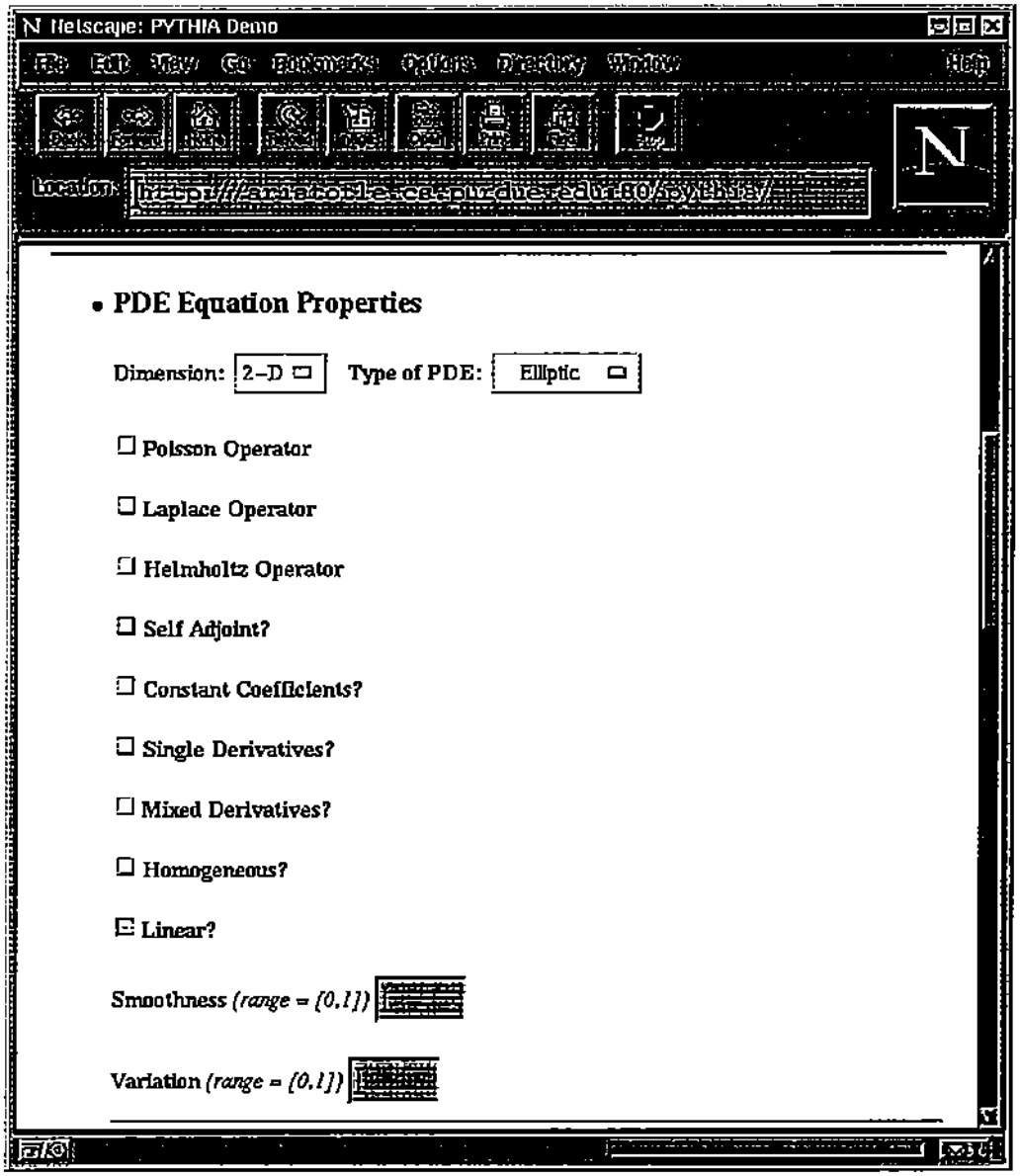

Fig. 1. Web Session with C-PYTHIA, input

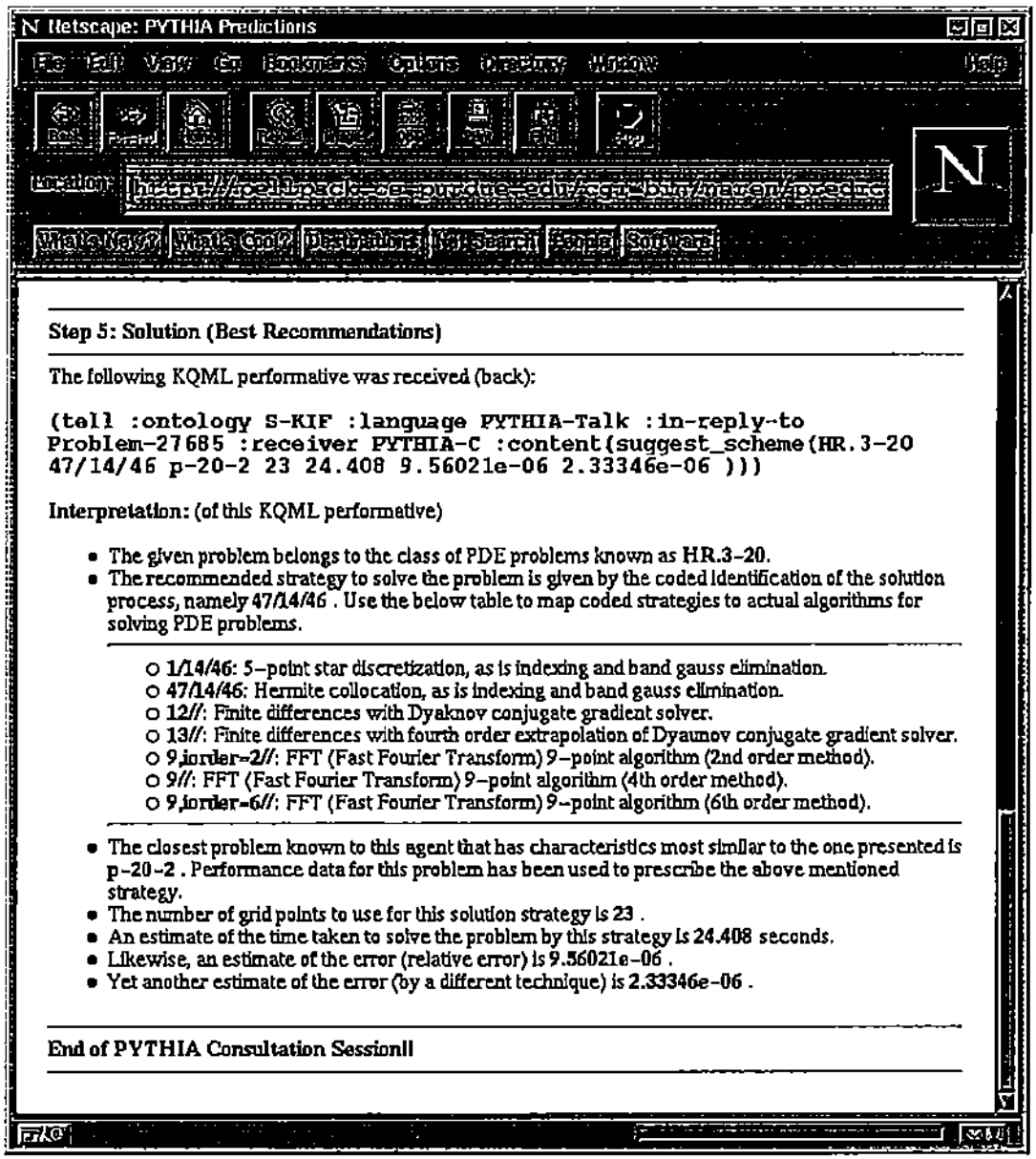

Fig. 2. Web Session with C-PYTHIA, output

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.

Scenario No.	No. of PYTHIA Agents	Details
1	6	One Agent for each class
2	5	same as (1) except that classes (d) and (e) are the combined expertise of one agent
3	4	same as (2) except that classes (a) and (c) are the combined expertise of one agent
4	3	same as (3) except that classes (b) and (f) are the combined expertise of one agent

Our experiments consisted of two main kinds: (i) The knowledge of the various PYTHIA agents in each setup was assumed to be 'static' (fixed *apriori*). In other words, the expertise of each agent contained information about the entire class(es) as described above and (ii) Each agent starts with a very small fraction of its representative PDE knowledge base and this gets subsequently refined over time. For example, Agent 2 (whose expertise is of the analytic PDE nature), starts with information about 9 problems and progressively improves with time to the full set of 35 problems. We refer to the first kind as the static case (where the knowledge of the various agents is assumed to be static and it is required to learn a one–shot mapping to these agents) and the second kind as the dynamic case (where the knowledge of the agents improves with time and we need to switch between learning and stable modes as appropriate).

This switch between learning and stable modes, can itself be done in a variety of ways. In this paper, we concentrate on the time–based, reactive and time–based reactive techniques. We detail these experiments in detail next. As mentioned before, there is also assumed to be a 'central agent' (PYTHIA–C) in this setup.

7.2 Static Scenario

In the static scenario, the expertise of each agent is fixed and equals the cardinality of the classes as described earlier. 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 to shoot out queries about each problem to each of the agents in the setup and soliciting recommendations from them. An example of a query by PYTHIA–C is as follows:

```
(tell   :ontology S-KIF
        :language PYTHIA-Talk
        :reply-with Problem-22750
        :sender PYTHIA-C
        :receiver PYTHIA-Agent1
        :content  (select_solution_scheme
```

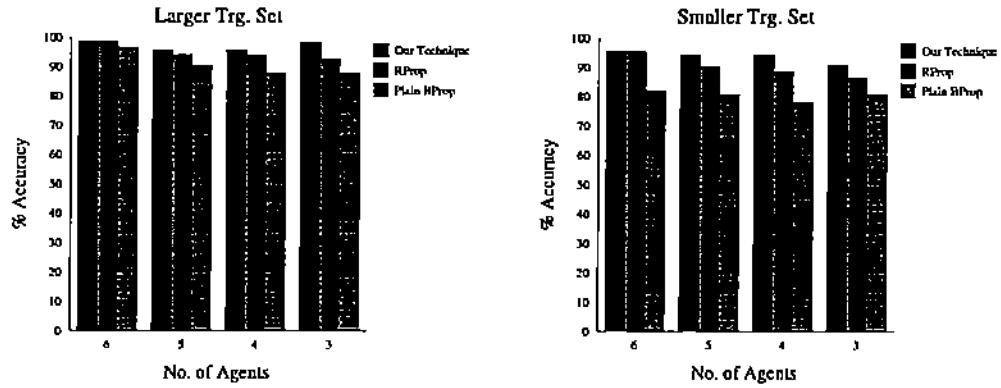

Fig. 4: Results of Learning. The graph on the left depicts the results with the larger training set and the one on the right shows the values for the smaller training set. In each case, accuracy figures for the 4 scenarios (with 2, 3, 4, 5 and 6 agents) are presented with all the three algorithms considered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

```
        (2 0 0 0 0 0 0 0 0 1 0 1 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 the `sender` is the PYTHIA–C agent, the `receiver` is one of the agents in the setup, and the `content` requires the agent PYTHIA–Agent1 to `select a solution scheme` for the PDE problem whose characteristic vector is given by the above sequence of numbers. This sequence is understood to be provided in the PYTHIA–Talk `language` in the S-KIF `ontology`. The tag `Problem-22750` indicates a reference that PYTHIA–Agent1 can reply with when communicating an answer. Similarly, an output from PYTHIA–Agent 1 would be:

```
(tell   :ontology S-KIF
        :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 scheme` described in the `content` field. This again refers to a PYTHIA–Talk description of the solution strategy.

Then, each of these recommendations is given a 'reasonableness' value and PYTHIA–C chooses the best agent as the one which has the highest reasonableness value as described in Section 4. Thus, this piece of 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 4 agent scenarios.

We conducted two sets of experiments: we first trained PYTHIA–C on the larger training set of problem, agent pairs and 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) [26] and Resilient Propagation (RProp) [2]. Fig. 4 summarizes these results.

It can be easily seen that our neuro–fuzzy method consistently outperforms BProp and RProp in learning the mapping from problems to agents. Also performance using the larger training set was expectedly better than that on the smaller training set. Moreover, our algorithm (housed in PYTHIA–C) operates in an on–line

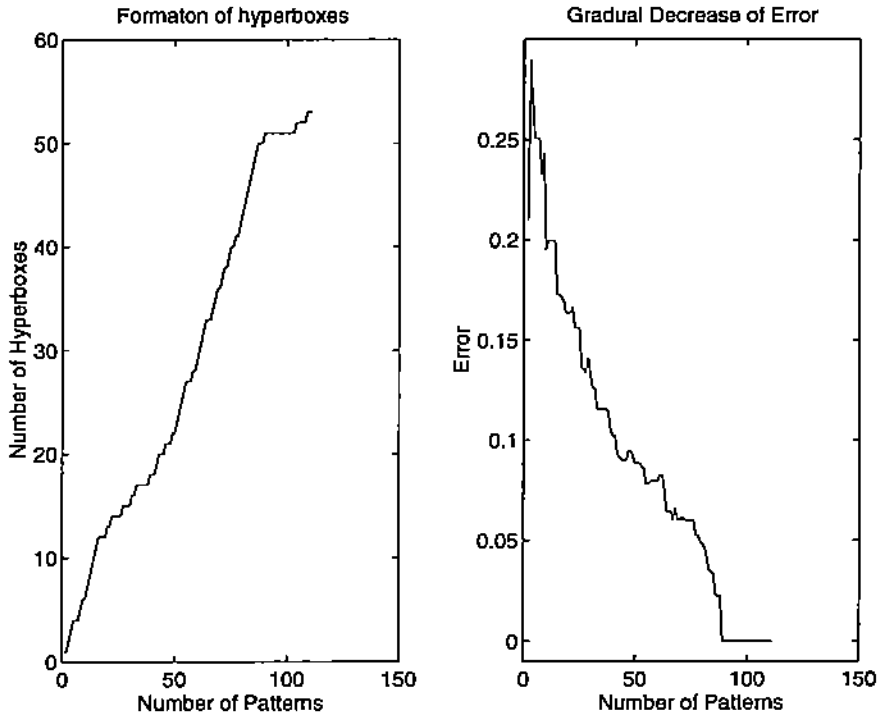

Fig. 5: On–Line Adaptation: The '6–Agent' scenario was trained with the larger training set. The graph on the left shows the incremental formation of hyperboxes. The one on the right shows the corresponding decrease in the error.

mode. new data do not require retraining on the old. For the larger training set, we incrementally trained our algorithm on the 111 exemplars and the accuracy figures on the test set were found to steadily rise to the values depicted in Fig. 4. For example, in the 6–Agent scenario (with the larger training set), we plotted the gradual decrease of error with increase in the number of problems seen by PYTHIA–C. Fig. 5 shows this steady improvement in accuracy alongside the number of hyperboxes utilized by the neuro–fuzzy technique housed in PYTHIA–C. (It is to be noted that the hyperboxes are incrementally added to the neuro–fuzzy scheme to better represent the problem space characteristics). The number of hyperboxes created for this scenario was 53. The progressive increase from zero to this number can also be seen in Fig. 5.

In a collaborative networked scenario, where the resources change dynamically, this feature of our neuro–fuzzy technique enables us to automatically infer the capabilities of multiple PYTHIA agents. The graphs in Fig. 5 should thus be viewed in an incremental learning scenario where the abilities of each of the agents are being continually revealed in the dynamic setting.

7.3 Dynamic Scenario

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. As explained earlier, this switch could be done in one of several ways — Time–Based, Reactive and Time–Based Reactive. We now describe results from each of these experiments.

7.3.1 *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, shoots out 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. Figs. 6, 7, 8 and 9 depict the results with the four agent scenarios and the time–based scheme with the larger training set. i.e., at periodic time intervals, PYTHIA–C switches to learning mode and cycles through the larger training set with each of the agents in the setup. The performance is then measured with the smaller training set. As can be

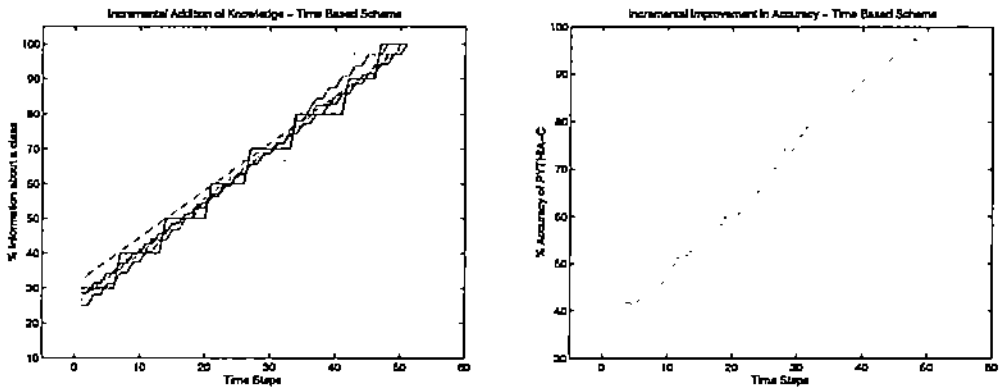

Fig. 6: Results with the Time–Based Scheme for 6 Agents with the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA–C

seen, the accuracy figure steadily improves in all of the cases to the accuracy observed in the previous set of experiments. Fig. 10 shows the accuracy figures for all the agent scenarios with the time–based reactive scheme. The steeper lines are the results for the case when the number of agents was maximal (in this case 6), and the one on the top of the graph is the case when there were 3 agents. (This phenomenon is common to all the three schemes.) As is expected, the scenarios with a lesser number of agents start off at a higher accuracy level and converge to the figures already discussed. This is because, these agents actually combine the expertise of the individual agents in the first scenario. PYTHIA–C's accuracy improves from 40.85% to 98.20% in scenario 1, from 59.15% to 98.20% in scenario 2, from 59.28% to 98.20% in scenario 3 and from 68.86% to 98.20% in scenario 4 (In fact, this is true for all the three dynamic schemes discussed. The only difference is the rate of progress towards these values). 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.

We conducted yet another series of experiments with this scenario, which are more realistic of real multi–agent systems. Rather than having each agent with approximately 1/3rd of their initial knowledge as a starting situation, we began with a scenario when there are no 'known' agents in the setup. i.e., PYTHIA–C does not know about the existence of any agents or their capabilities. Then, each agent was slowly introduced into the scenario with a small initial knowledge base and then their abilities were slowly increased. For example, Agent 1 comes into the setup with a 'small' knowledge base and announces its existence to PYTHIA–C. PYTHIA–C creates a 'class' for Agent 1, reverts to learning mode (though wasteful) and learns mappings from PDE problems to agents (in this case, there is only one agent in the setup). After some time, Agent 3 comes into the scenario and this process is repeated. While the addition of new classes is taking place, the abilities of existing agents (like Agent 1) also increase simultaneously. Thus, these two events happen in parallel; i.e., addition of new agents and refinement in the knowledge (abilities) 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. Needless to mention, the accuracy figures converged to the values already mentioned.

7.3.2 *Reactive.* In the reactive scenario, PYTHIA–C doesn't do polling at regular intervals, but instead waits for other agents to broadcast information if their confidence for some class of problems has changed significantly. Again, each agent started with the same initial level of ability and their 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 figures 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. Figs. 11, 12, 13 and 14 illustrate the results with each of the agent scenarios described in this paper. As can be seen, there is a more progressive increase in the accuracy figures and these curves begin more to resemble a 'staircase' pattern. Figs. 15 summarizes these curves for all the agent scenarios.

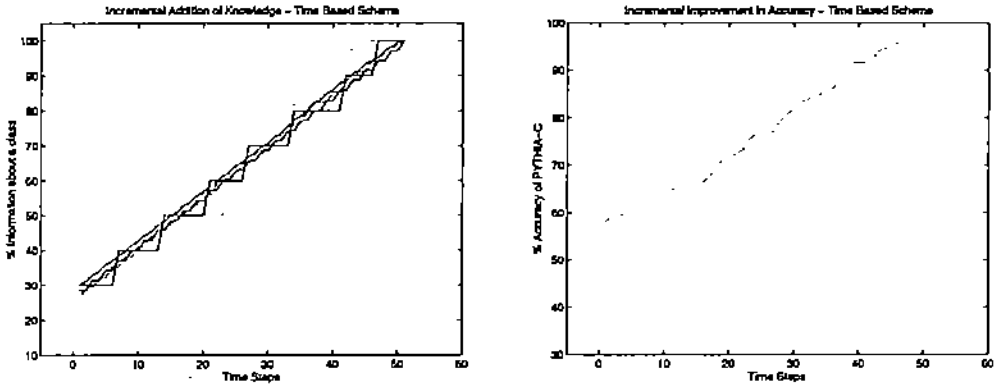

Fig. 7: Results with the Time-Based Scheme for 5 Agents with the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C

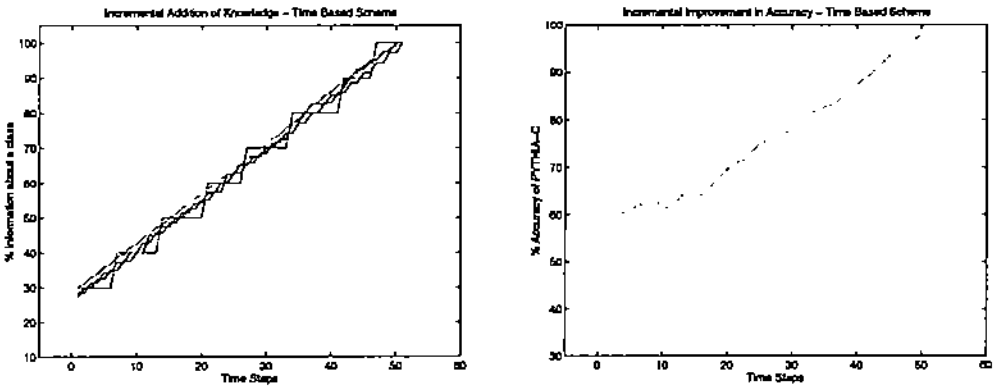

Fig. 8: Results with the Time-Based Scheme for 4 Agents with the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C

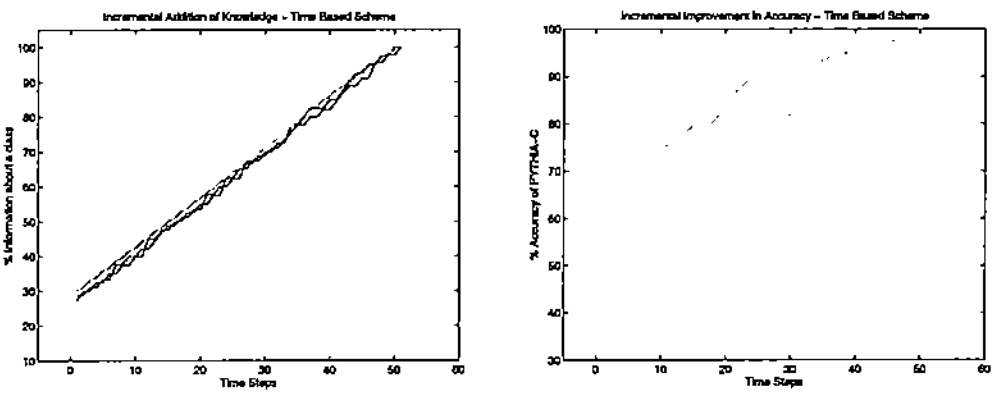

Fig. 9: Results with the Time-Based Scheme for 3 Agents with the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C

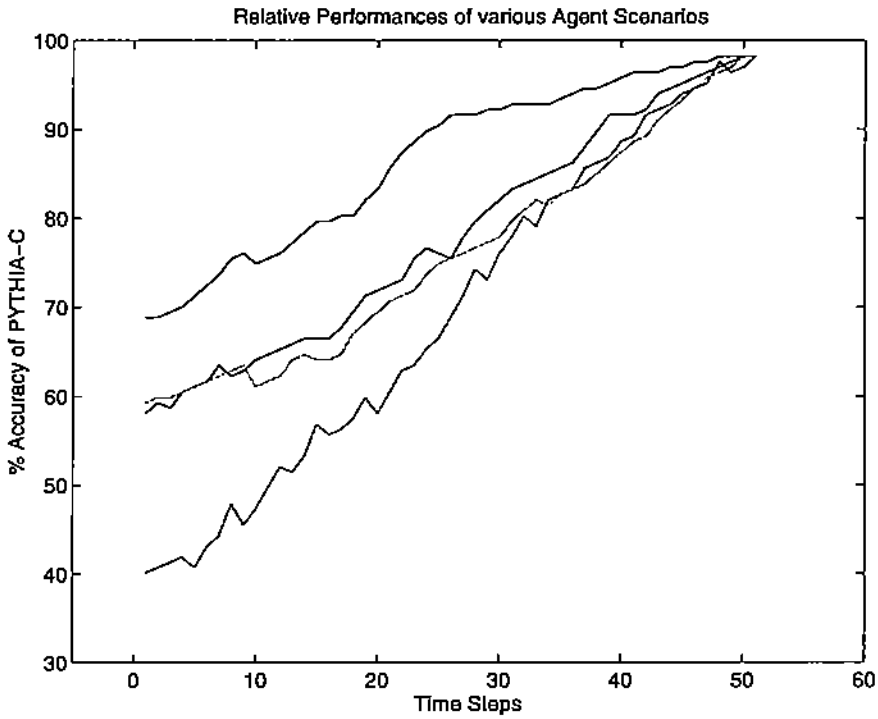

Fig. 10. Relative Accuracies observed by the various Time-Based schemes with the larger training set.

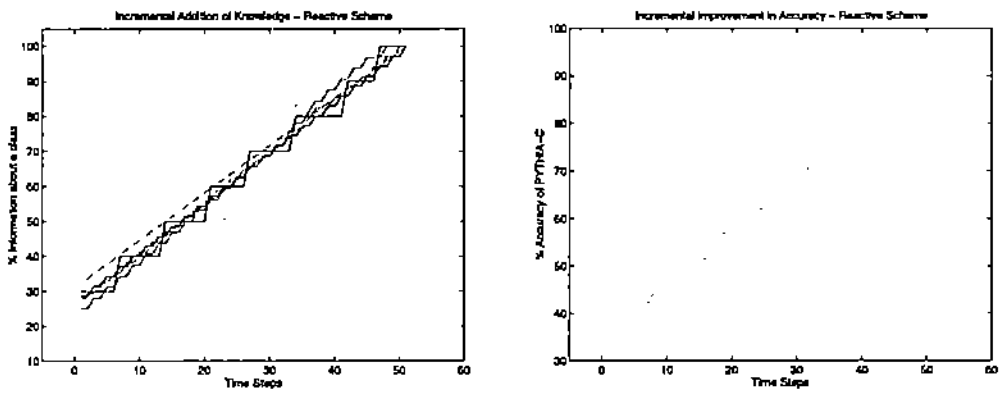

Fig. 11: Results with the Reactive Scheme for 6 Agents for the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C

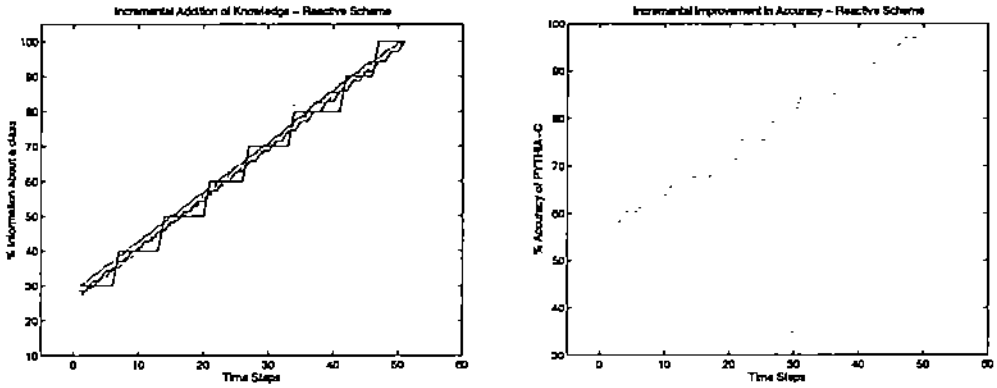

Fig. 12: Results with the Reactive Scheme for 5 Agents for the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C

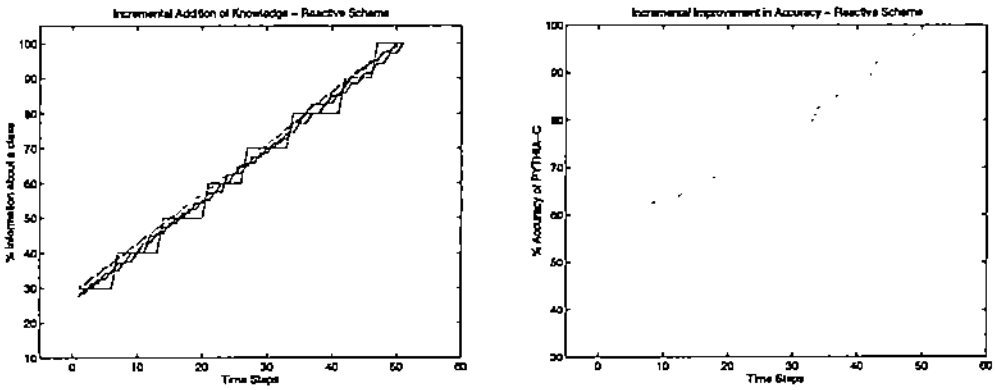

Fig. 13: Results with the Reactive Scheme for 4 Agents for the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C

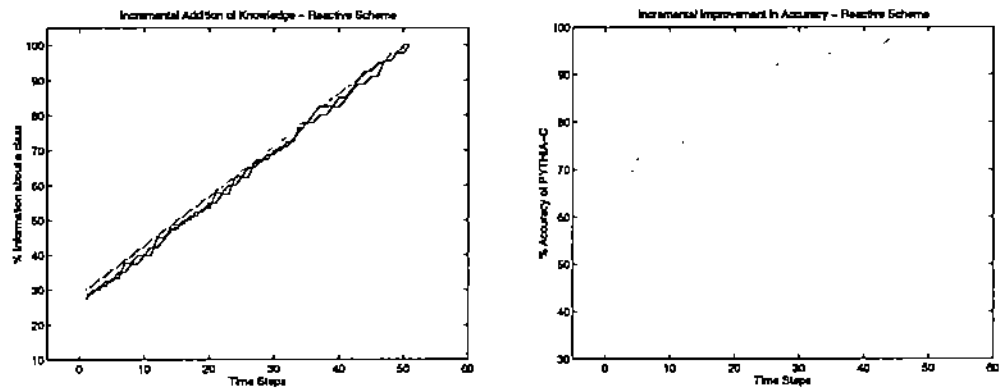

Fig. 14: Results with the Reactive Scheme for 3 Agents for the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C

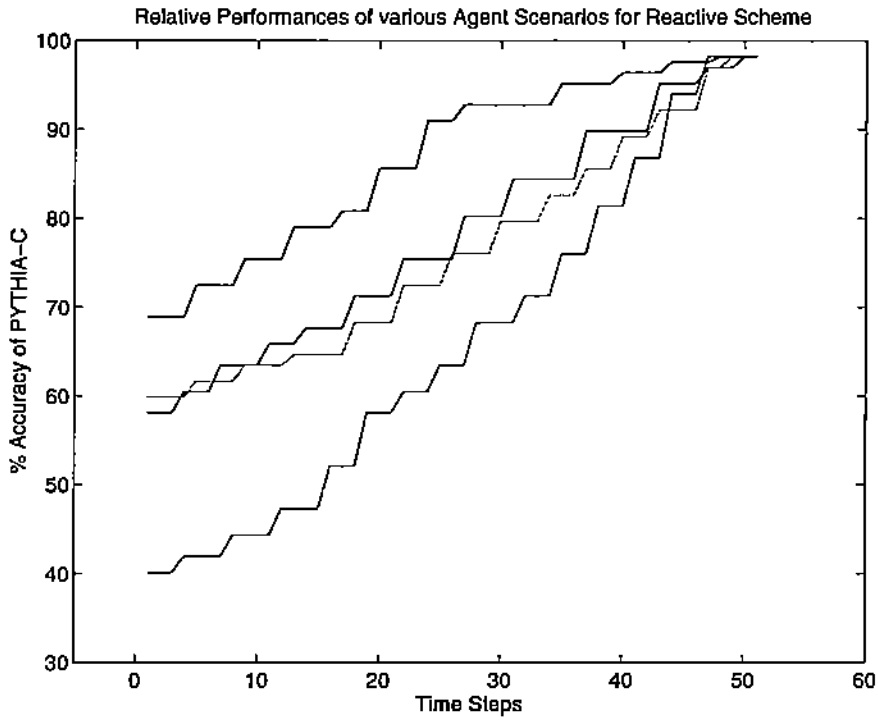

Fig. 15. Relative Accuracies observed by the various Reactive schemes with the larger training set.

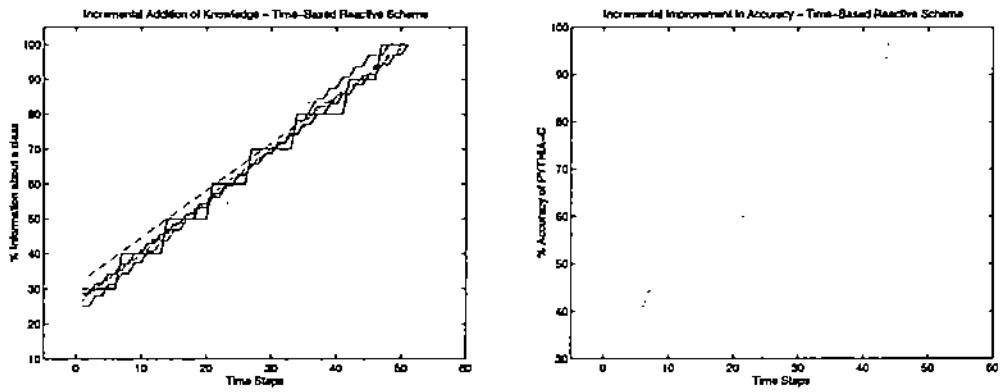

Fig. 16: Results with the Time-Based Reactive Scheme for 6 Agents for the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C

7.3.3 *Time-Based Reactive.* In the time-based reactive paradigm, we use a combination of the two approaches previously outlined. PYTHIA-C sends out a "has anyones abilities changed significantly" message at periodic time intervals and reverts to learning mode if it receives a reply in the affirmative. Each agent, as before, started with the same initial level of ability and their expertise was slowly increased. It was observed that the accuracy figures 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. Figs. 16, 17, 18 and 19 illustrate the results with each of the agent scenarios. Figs. 20 summarizes the accuracy curves for all the four agent scenarios.

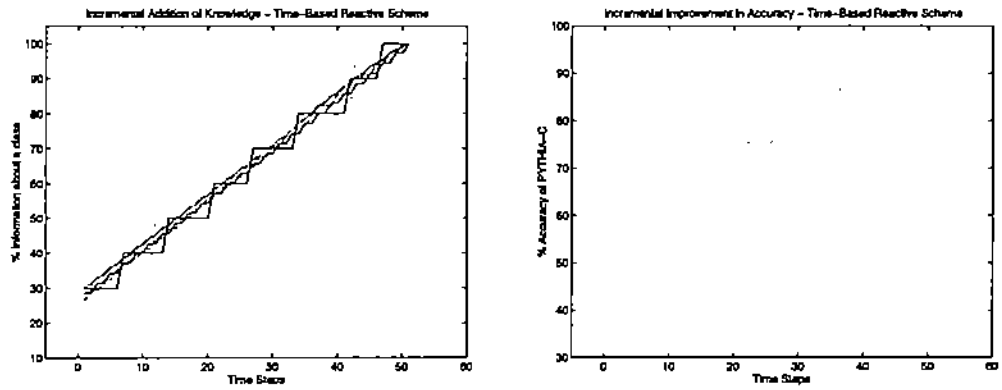

Fig. 17: Results with the Time-Based Reactive Scheme for 5 Agents for the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C

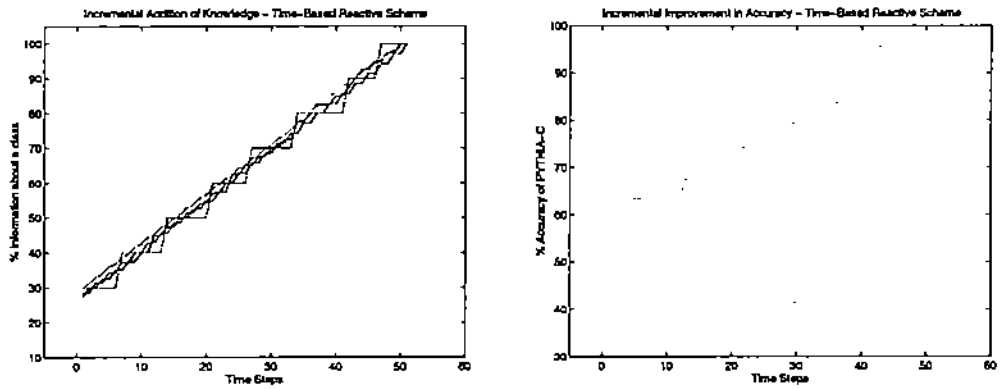

Fig. 18: Results with the Time-Based Reactive Scheme for 4 Agents for the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C

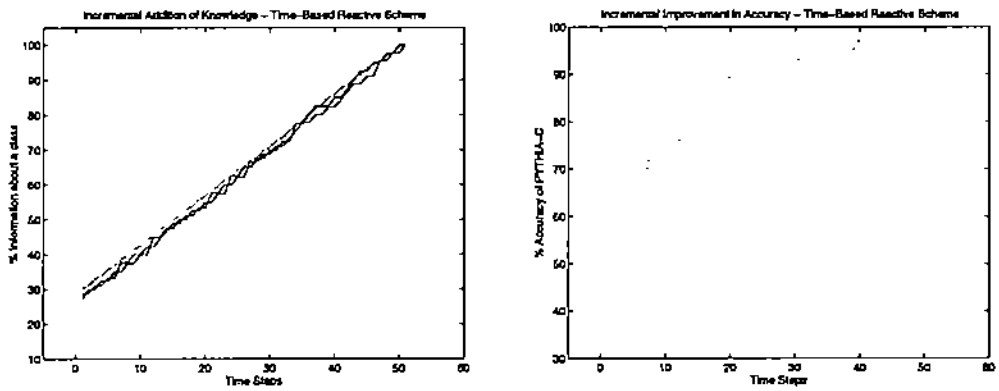

Fig. 19: Results with the Time-Based Reactive Scheme for 3 Agents for the larger training set. The graph on the left shows the periodic increase in the abilities of the agents and the one on the right shows the corresponding improvement in accuracy of PYTHIA-C