Patisserie: Support for Parameter Sweeps in a Fault-Tolerant, Massively Parallel, Peer-to-Peer Simulation Environment

Tim Schoenharl and Scott Christley
Dept. of Computer Science and Engineering
University of Notre Dame
Email: {tschoenh,schristl}@nd.edu

***Abstract*— We propose and implement a parameter sweep framework built upon a fault tolerant, massively parallel, peer-to-peer infrastructure. Our framework is composed of a scheduling algorithm based on the Particle Swarm Optimization algorithm and peer-to-peer storage and query functionality, built on the Pastry peer-to-peer framework. The Particle Swarm Optimization algorithm is a perfect fit for a fault tolerant, distributed environment, as it is able to steer the simulations using local information and minimal communication. PSO is naturally fault tolerant through redundancy and an underlying structure of neighborhoods. Building query and storage functionality on top of a peer-to-peer network addresses scalability of the number of compute nodes, and we describe how fault-tolerance of simulation results is maintained by the framework.**

Key Words: Autonomic Computing, Swarm Intelligence, Peer-to-Peer Systems, Distributed Systems, Parameter Sweep Applications

I. Introduction

Simulation has become a fundamental tool for scientists. Computational simulations have helped to push the boundary of knowledge in pharmaceutical research, climate modeling, global economics and countless other fields. Often when developing simulations, scientists are confronted with a huge parameter space and very little notion of reasonable values. This necessitates a parameter sweep, the running of many simulations in order to determine reasonable values. This type of work is often highly parallelizable and very time consuming. Additionally, given a large parameter space, enumeration of every possible combination of parameters is out of the question. This suggests a need for an adaptive scheduling algorithm that can steer a parameter sweep towards interesting regions. In this paper, we seek to provide such a framework. We introduce a swarm intelligence-based approach that is highly decentralized, fault tolerant and yet yields excellent results.

Our framework provides a decentralized, fault tolerant approach to exploring complex parameter spaces. It extends the work that others have done [1][2][3], but differs in several important ways from these other efforts. The AppLeS Parameter Sweep Template [1] provides a centralized scheduler, which provides better steering performance than our approach, but is subject to the limitations of a centralized system, such as limit scalling. The Organic Grid [3] arranges computational nodes in a hierarchy, which requires enormous effort to maintain. Patisserie is built upon a peer-to-peer network and is resilient to network change. The SOMCP2PN framework [2] is very close to our work, but provides a novel scheduler that has not been peer-reviewed to the level of the Particle Swarm Algorithm. We present a more in-depth discussion of related research in Section VIII.

Our target application is a simulation that has a large parameter space and for which there exists some type of objective function. The fitness value for a set of parameters is a result of the objective function and is used in the global optimization process. An objective function could be the minimization of resting energy of one protein conformation, for example. We do not currently consider cases where the transmission of input/output data is a limiting factor. Nor do we rely on a master/worker relationship, which we feel presents certain challenges in an unreliable computational network.

We demonstrate the functionality of our framework with canonical examples from optimization. We evaluate the performance, fault tolerance and scalability of our system and on several variations. Finally we comment on the strengths and weaknesses of the system and lay out a series of extensions that will make the system more useful to researchers.

II. Patisserie Framework Overview

Our goal is to provide a simulation framework that scales well, offers robust fault tolerant performance, provides decentralized scheduling of simulations and allows researchers to locate and retrieve relevant simulation results. We built our framework on top of the Pastry peer-to-peer framework, giving us a peer-to-peer foundation. We implemented a decentralized Particle Swarm Optimization algorithm for scheduling simulations. And we created a Query API for managing simulation results. Together these components provide a coherent, well-

matched ensemble enabling parameter sweeps on a massive scale.

The Patisserie framework is composed of three core components: the peer-to-peer network, the particle swarm optimization scheduling algorithm, and the simulation results manager. The peer-to-peer network provides functionality for sending and receiving network messages between compute nodes; peer-to-peer networks have the capability to scale to a massive number of nodes because no single node requires global knowledge of the network. The particle swarm optimization scheduling algorithm runs on each compute node and determines the set of parameter values for the next run of the simulation on that compute node. Lastly, after a single simulation run finishes on a compute node, results are produced in both raw data form and as summary metrics; the management of these results require them to be stored so that the user may query the results at a later time. In the following sections, we will describe each component in more detail.

III. Particle Swarm Optimization Scheduling

Particle swarm optimization (PSO) is a swarm intelligence-based approach to optimization where a collection of autonomous particles "fly" through a parameter space, exploring it in loose cooperation[7]. Swarm intelligence refers to the ability of groups of simple agents with limited communication to cooperate and create complex behaviors. PSO is an example of a complex adaptive system, which can be characterized as a system where a large number of autonomous agents cooperate using simple local rules to accomplish a global task. In the case of PSO, the particle agents communicate their best observed value to the other agents (in one implementation, values are shared with a particle's neighbors, in another method results are shared globally). This best observed value is used by other agents as a target, they create a vector from their current position to the best position, add some random noise, and this vector determines their new location in the parameter space. The attractiveness of a swarm approach is that it is decentralized, highly fault tolerant (achieved via redundancy), requires, in its simplest form, limited communication and achieves good results. Each of these criteria are important in a decentralized, peer-to-peer simulation environment.

A. Particle Swarm Optimization: The Algorithm

The fundamental element of the PSO algorithm is a particle. A particle is essentially an agent that has three roles: It holds a parameter list which is its "location" in state space, it has a method for moving from its current location to a new location and it has a neighbor relationship with one or more other particles whereby information is passed back and forth. Each particle represents its location as a vector of values (usually integers or real numbers, in some cases binary or categorical values). In order to explore search space effectively, it is important for the agent to store another vector of values corresponding to the most fit point it has visited thus far. This "best" value is used in the selection of a new location to evaluate. When a new best value is encountered, the particle will alert its neighbors by sending a message with the location and evaluation of that vector. The standard procedure for iterating from one location to another is as follows. For each value in the position vector, compute its velocity according to Equation 1 (As given in [7]).

$$v_{id}(t) = v_{id}(t-1)+\varphi_1(p_{id}-x_{id}(t-1))+\varphi_2(p_{gd}-x_{id}(t-1)) \quad (1)$$

The φ terms simply represent positive values that are taken from a uniform random distribution and satisfying $\varphi_1 + \varphi_2 = 4$. The values p_{id} and p_{gd} are the values from the particle's personal best vector and from its neighbors' best vector, respectively. The value $x_{id}(t-1)$ is the associated value in the particle's current position vector. Once $v_{id}(t)$ has been calculated, it is important to make sure that it falls within the acceptable range of velocities. The velocity must be damped to keep the particle from oscillating out of control. Here the standard practice is to select a value, V_{max} and restrict the velocity to $-V_{max} \leq v(t) \leq V_{max}$. Equation 1 demonstrates how the particle's velocity is affected by its current position, the best location that it has personally visited and the best location visited by one of its neighbors.

In order to finally create a location vector for its next position, the particle simply adds the velocity vector to its current position vector. The velocity vector is stored for use in the next iteration of the algorithm. The particle's old position vector is discarded, unless it evaluates to a new personal best, in which case it is stored as the personal best vector.

Implicit in this definition of a particle is the existence of a method of translating the particle's position into a fitness value. The fitness value helps to direct the particle's flight through the search space. The particle need not be in control of the evaluation, a black box will suffice. The particle is only concerned with the resulting fitness value and not with the manner of the evaluation. This property is another attractive feature of the PSO algorithm.

The particle's neighborhood is important in evaluating the search space. The neighbor relationship allows several particles to collaboratively evaluate search space, while passing only useful information. There is certainly the potential for a particle to evaluate a location that has already been visited by one of its neighbors. The cost of this repeated evaluation is directly proportional to the cost of the evaluation function (which is important in our context). For a simple polynomial equation, repeating an evaluation is trivial. In the case where an evaluation entails the running of a simulation, the cost is high. In circumstances where the cost to benefit ratio is high, it

might make sense to store a small amount of data enumerating the locations in search space that have been visited. However, this decision must take into account the likelihood of revisiting a location, which might be low in a search space such as $\Re^{20}$, as used in our examples.

The restriction placed on velocity will have consequences on the behavior of the PSO algorithm. For large values of V_{max}, the algorithm will converge quickly around a solution. However, this comes at a cost in terms of the accuracy of the solution. Large values of V_{max} have the unintended consequence of preventing a particle from getting within an associated range of the optimal value. Smaller values of V_{max} will be able to get closer to the optimal value, as their step size is smaller, and thus they are less likely to overshoot the optimum. A consequence of using a smaller V_{max} is that it takes the algorithm far longer to converge. The V_{max} value can be seen as the size of the biggest jump in state space. Big jumps get particles close to the optimum faster, but prevent it from getting within a certain radius of the optimum. And conversely for small jumps, particles can get closer to the optimum, but they will take much longer to get there. There are more sophisticated methods of dealing with velocity, such as using inertia weights[7].

Pseudocode for the main loop of the Particle Swarm Optimization Algorithm is provided in Figure 1 (Taken from [7]). In the standard implementation, each Particle queries its neighbors at each iteration to determine the new local best value. We felt that this was an unnecessary waste of messages, and so inverted the procedure. Now each node maintains a local best value and upon discovering a new personal best, nodes will send out messages to their neighbors.

Readers who are interested in learning more about the intricacies of Particle Swarm Optimization are encouraged to read "Swarm Intelligence" [7] by Kennedy and Eberhart. The text identifies several modifications to the standard PSO algorithm that are beyond the scope of this paper, among them using inertia weights to ameliorate the velocity trade-off.

B. The PSO Algorithm in a Peer-to-Peer Context

Our approach to scheduling uses the standard particle swarm optimization, with a few small changes required by the nature of the distributed system. Here, each computational agent (a compute node in the p2p network) explores some area of parameter space and informs its PSO neighbors when a particular vector of parameters evaluates to a new best value. We define PSO neighbor relations using the peer-to-peer framework, which has certain implications. In the traditional, centralized PSO, neighbor relations are static, symmetric and defined a priori. In the Pastry framework, neighbor relations are dynamic and are not symmetric. We accept the dynamic neighbor relations as provided by Pastry, but keep the overall number of PSO neighbors constant. Given the importance of neighbor relationships to the performance of the PSO, we made PSO neighbor relationships symmetric.

```
while(!simulationEnd){
  for i=1 to number of particles {
    if fitness(i.pos) < fitness(i.perBest) {
      i.perBest = i.pos;
    }
  }
  g = i;
  for j= indexes of neighbors{
    if fitness(j) < fitness(i.localBest){
      i.localBest = j.pos;
    }
  }
  for d=1 to number of dimensions{
    i.nextVel[d] = i.curVel[d] \
        + q1*(i.perBest[d] - i.pos[d]) \
        + q2*(i.localBest[d]-i.pos[d]);
    if i.nextVel[d] > velocityMax {
      i.nextVel[d] = velocityMax;
    } else if i.nextVel[d] > velocityMin {
      i.nextVel[d] = velocityMin;
    }
    i.nextPos[d] = i.pos[d] + i.nextVel[d];
  }
}
```

Fig. 1. Pseudocode for the Particle Swarm Optimization algorithm

We take advantage of the flexibility of the PSO algorithm by separating the fitness evaluation and the position calculation. In order to make our system flexible, the particle's responsibility is to maintain its current location and personal and global best locations and to iterate from the current location to the next location. Details of this are given in Section VI.

It should be noted that it was necessary to make only minimal modifications to adapt the PSO algorithm to a peer-to-peer context. PSO is inherently fault tolerant, has low message traffic and is useful as a general purpose optimizer. The fault tolerance of PSO comes through redundancy and ambivalence to neighbor failures. It exhibits graceful degradation in the face of increasing rates of failure. The fault tolerance and graceful degradation both relate to the way PSO neighbors contribute information. When a node disappears, its PSO neighbors no longer receive its input, however, they remain connected to their other PSO neighbors, so the algorithm can continue to function. Since only useful results are passed between particles, and because communication is local, at least in the standard version of the algorithm, the number of messages passed through the network is relatively small.

Another strength is that the PSO algorithm works with a variety of fitness functions, which makes it possible to use PSO as a steering mechanism for a wide variety of simulations. A final advantage of PSO is that it maps well to an environment with heterogeneous computing resources. Fast nodes are not penalized when a slow node joins the network. Given the communication mechanism, there is no scenario where one node can cause the entire system (or even a neighborhood) to slow down or block.

C. Variations of the Particle Swarm Optimization Algorithm

We present and evaluate three variations on scheduling using Particle Swarm Optimization: Centralized scheduling with a global-knowledge PSO, centralized scheduling using a local-knowledge PSO and distributed scheduling using a local-knowledge PSO.

1) Centralized, Global-Best PSO: In the centralized implementation, the nodes in the network rely on a central server. The central server maintains neighbor relationships and is in charge of updating particles with new local or global best vectors. Each node maintains its own ability to calculate a new position in the state space and handles the evaluation of its location. The central server acts as an aggregator of the personal best vectors, making it easy for a researcher to see the progress of the optimization.

In a global-best implementation of PSO, neighborhoods are not important. Each particle shares its personal best result with the entire swarm (in our context we will refer to it as the network). This implementation is ideal for a central server, as the server need not maintain neighborhood relationships and nodes only need to be aware of the central server. The server needs to maintain only a simple list of the nodes in the network and a global best vector. The global algorithm, according to the literature, provides the fastest convergence behavior. However, in a peer-to-peer context, this requires at least O(N) messages for every global best update.

2) Centralized, Local-Best PSO: The local-best PSO is as described above. Here the central server is responsible for forwarding local best messages between neighbors, as well as maintaining neighborhoods as nodes join and leave the network. In a centralized system, we have more control over the topology of the neighborhood. Kennedy and Eberhart in [7] note that the neighborhood topology can effect the performance of the PSO algorithm. For more discussion of this, see section IX. The local-best PSO provides slower convergence compared to the global method, but there is a reduction in the required number of messages, O(K), where K is the number of neighbors. However, the central server still provides a single point of failure and a limiting factor on the scalability of the system.

3) Peer-to-Peer, Local-Best PSO: The peer-to-peer implementation is the focus of this paper. In this version, each node maintains its own neighbor list and handles communication with its neighbors. The algorithm is as discussed in Section III-B. This implementation was by far the easiest to implement, however, it presents certain challenges to the researcher. With no central server, it is not possible to view the state of the current system. This limitation is what drove the creation of the Query API. The behavior of the peer-to-peer, local-best algorithm is equivalent to that of the centralized, local-best algorithm, but without a central point of failure.

IV. Peer-to-Peer Network

The peer-to-peer network software component provides functionality to send and receive network messages in an asynchronous, event-driven manner. Recent work in peer-to-peer networks include definition of a common API[5] across implementations which allows applications to explore changing the underlying peer-to-peer implementation; however, we currently just use the Pastry[6] implementation. Pastry provides each node with a unique identifier and partitions a key space across all nodes such that each node has a roughly equal distribution of that key space. Nodes can join and leave the network dynamically, and Pastry maintains the underlying routing tables so that messages are efficiently transmitted to the destination node; likewise, it performs redistribution of the key space for the nodes to maintain the uniform distribution. These capabilities provide for high scalability of the network because the routing tables only need to maintain links to a relatively few number of neighbors nodes for network connectivity, and notification of network changes provides for fault-tolerance that we utilize at the application layer.

Figure 2 shows a typical network topology for Patisserie. Pasty has its set of neighbors for maintenance of the peer-to-peer network topology and routing of messages; the PSO neighbors are a subset of the Pastry neighbors which are used by the PSO algorithm as part of its search mechanics. Therefore, Patisserie automatically defines the PSO neighborhood as an overlay network on top of the physical peer-to-peer network, and adjustments to the peer-to-peer network due to node joins and removals are automatically reflected in the PSO neighborhood which keeps the neighborhoods balanced among all particles for the PSO scheduling.

V. Management of Simulation Results

A. Storage and Backup Copies

When a simulation run finishes, the raw data files should be saved such that those results can be queried and retrieved at a later time. We make the assumption that the raw data is large in size, so transferring the data from one node to another should be avoided whenever possible. Likewise, in a large network, we assume that no shared file system is available; only local storage on each compute node can be used. For this reason, our design is to leave the raw data files on the local compute

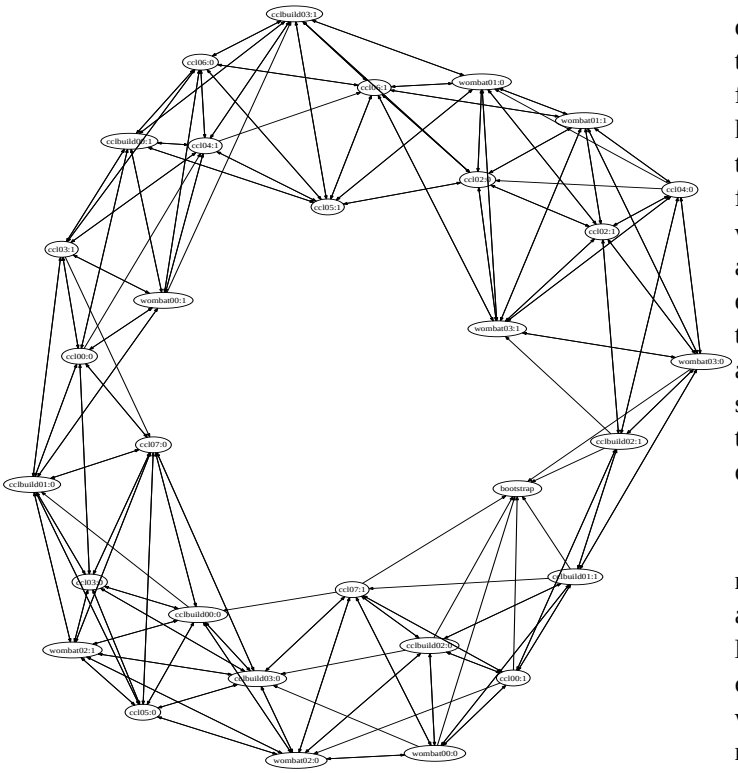

Fig. 2. Peer-to-peer network of sixteen machines each with two virtual nodes. Each node has seven PSO neighbors represented by directed edges with the topology defined by the Pastry network.

node and just store referential data that points back to the compute node. Our framework provides the *Store*, *Lookup*, and *Retrieve* message types which are implemented on top of the distributed hash table functionality supplied by the peer-to-peer network. *Store* takes a key/data pair; the key is the computed value from the PSO evaluation function, and the data contains the parameter set for the simulation run, the compute node identifier that stores the raw data files, local pathnames for the raw data files, and possibly some computed summary metrics for the simulation run. The key is mapped to a node in the peer-to-peer network, generally different from the original compute node, and that node stores the referential data. *Lookup* performs the inverse of *Store*; given a key value it returns the referential data associated with that key. *Retrieve* is used if the raw data is desired, so given the referential data from a *Lookup* message, *Retrieve* will transfer the raw data files from the original compute node back to the requester.

Storage of a simulation result in the peer-to-peer network requires transforming the result into the key space for the network so that the appropriate node can be determined; however, the peer-to-peer network key space generally has a vastly different range. For example, the Pastry implementation provides a 2^{160} bit key space, so if the simulation produces results in a small range then most likely all of those results will get stored onto a single node in the network; this can defeat both scalability and fault tolerance. Therefore, given that the simulation produces a single value from the evaluation function, the approach we take is to scale that value to the key space range; this requires that the simulation provides the range of values that will be produced by the evaluation function. This, however, is not an optimal solution because while we may have alleviated results being all stored at a single node, the distribution of storage across nodes is directly correlated to the distribution of results generated by the simulations. If the distribution of results are not uniform across the range; which they most likely are not, then the storage of results will not be uniform across all the nodes in the peer-to-peer network. We are continuing investigation into other techniques which will provide better storage distribution.

Our framework must adjust to a dynamically changing network environment. Nodes may leave or join the network at any time. When nodes leave, they can either gracefully leave by cleaning up and letting their neighbors know, or they can die with no warning. When a node leaves gracefully, we want to migrate simulation data to other nodes, and when a node involuntarily dies then we want data redundancy in the network so that simulation results are not lost. Our mechanism for storage of simulation results degrades as nodes leave the network because the raw data files and the referential data can be lost. There are two viewpoints that can be considered; one recognizes that simulation runs may be cheap so it is more efficient to regenerate the simulation results when they are lost. The other approach is to store backup copies of the data on other nodes; this is often called replication but we use the terminology of backup copies to avoid conflict with the notion of replication runs for a simulation. Backup copies need to be stored for both the referential data and for the raw data files. Additional copies of the referential data is handled by constructing slightly different key values, so the different keys will be mapped and stored on different nodes in the peer-to-peer network. If lookup fails on the main key, the backup keys can be constructed for lookup. Backup copies of the raw data files requires that the data be copied from one node to another; then the referential data can hold names for all of the nodes with copies, so the requester can try any of the nodes to retrieve the data. It should be noted that with the first viewpoint where simulations are cheap, referential data backup copies should be kept in case a node goes away, while if simulation data cannot be cheaply reproduced then both referential data and raw data files should have backup copies. Our framework does not currently implement replication, but we intend to provide it in the future. The main implementation consideration is how to generate different key values that map to different nodes; one approach we are investigating is using a permutation modulo based upon the number of replications, so the key is shifted around the key space in a circular fashion.

B. User Queries

If the researcher knows the exact PSO evaluation function value then the lookup and retrieval functionality described in the previous section is sufficient to get simulation results; however, this is generally not the case. What we describe is an algorithm for how the researcher can obtain the simulation results for a range of values; and because referential data is stored, the researcher can preview the query results before deciding on retrieving the actual simulation data.

Structured peer-to-peer designs, like Pastry[6] and others[8][9][10], provide functionality for exact match lookup of key values, but we can build support for range lookups on top of that functionality. Essentially when processing a range, take the lower bound of the range and perform a exact match lookup for the lower bound. Each node in the peer-to-peer network knows the range of key values that it maintains, so it compares its key range to the query range. If the node's key range covers the query range then the node holds all the simulation results; otherwise, the node constructs a new range using its key upper bound as the query lower bound and passes the new range to its next peer neighbor. Each node sends its results, even if null results, back to the requesting node which combines the results from each subrange into a total set of results.

One relaxation we apply is that timeliness is more important than completeness for query results. New simulation results may be generated in the middle of processing a query that are relevant to that query, so providing an exact set of results would entail synchronizing the query with the currently running simulations. Such a synchronization requires non-scalable and global communication, so by not requiring exact results; range queries can be processed quickly and involves only nodes which are known to contain store relevant results.

VI. Implementation

Here we present all relevant details of the Patisserie system that were not explicitly covered in earlier sections. Figure 3 shows the architecture for a Patisserie node in the framework. The ResultsManager sends and receives peer-to-peer network messages using Pastry. Incoming messages related to simulation results are handled by ResultsManager while messages dealing with currently executing simulations are forwarded to the SimulationManager. The SimulationManager is responsible for management of the executing simulation; the PSO algorithm is consulted for the set of parameter values to be used, and simulation results are passed back to the ResultsManager for storage in the peer-to-peer network. Both the ResultsManager and the SimulationManager execute in separate threads allowing them to work independently and communication between the two is asynchronous.

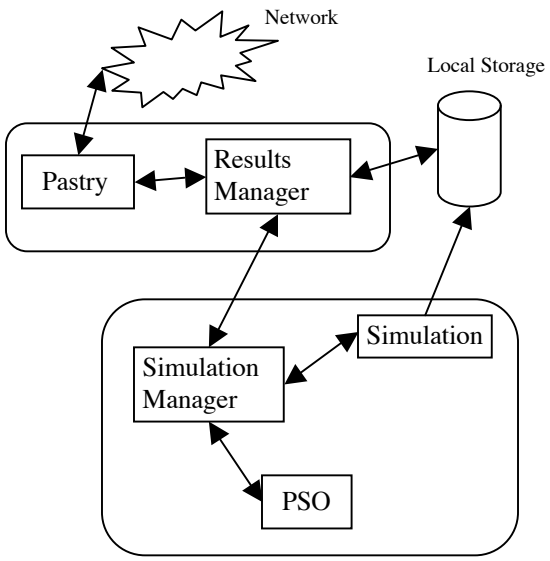

Fig. 3. Architecture of a Patisserie Node.

A. SimulationManager

The SimulationManager is responsible for running simulations, storing the results of the simulations and managing the PSO algorithm. The SimulationManager runs in a separate thread from the ResultsManager and runs a simple loop until it is called to exit. The execution loop consists of generating a vector using the PSO, using the vector as input to the simulation, running the simulation, storing the output of the simulation (which entails a method call to the ResultsManager) and, if the result of the simulation is a new personal best, sending a message to neighboring nodes. We use a shared data structure to hold local best results. Thus if the SimulationManager is running a simulation when a "new local best value" message arrives, the ResultsManager can directly modify the data structure. The update of the data structure is atomic, which prevents problems with the PSO receiving corrupted local best vectors.

At the beginning of each iteration of SimulationManager's main loop, its personal best and local best values are synchronized with the PSO object. Since the evaluation of position vectors is done by the Simulation object, the SimulationManager is responsible for closing the loop and returning the feedback to the PSO object. During the initialization phase, the SimulationManager gives the PSO object a random starting vector. This is done as a convenience. It is certainly conceivable that a researcher will provide a starting position for the Simulation, in which case we will generate a starting position using the given input as a base.

It is important at this point to mention the method that we use to make PSO neighbor relations symmetric. In the peer-to-peer implementation, we use the neighbors provided by the Pastry framework. Pastry neighbors are called leaves and the

neighborhood (set of all leaves) is referred to as the leaf set. A node has access to its leaf set, and thus can determine its outgoing PSO neighbors. However, it has no idea of which nodes have it in their leaf sets. In the interest of frugality, we refrain from sending a notification to nodes in the leaf set. Instead, we use an implicit join mechanism that works in the following manner. Given two nodes, A and B, where A has B in its leaf set and B does not have A in its leaf set. When A encounters a new personal best, it notifies its PSO neighbors by sending a message to every member of its leaf set. Node B will receive the message and note that A is not in B's leaf set. B will then add A to its leaf set, but no acknowledgment is sent to A.

The PSO neighbor relation method is not without problems. Pastry does not guarantee delivery of messages, therefore node A cannot be sure whether B has successfully added A to its leaf set. Conceivably a reliable transmission protocol could be built on top of Pastry, but this is beyond the scope of our project. We are content to suffer the occasional error, and as our results demonstrate, the functioning of the algorithm does not appear to be affected.

B. Particle Swarm Optimization Class

The Particle Swarm Optimization class is essentially as described above. Several small details differ from the algorithm and merit mention. The data structures in the Particle Swarm Optimization object are not directly accessible to outside classes, and in our design, only the SimulationManager directly interacts with the PSO object. Most of the methods in the PSO class are idempotent. The exception to this rule is the computeNextPosition() method. This method is not idempotent owing to the use of a pseudo-random number generator in the velocity computation. This method could be made idempotent by exposing the random seed in the pseudo-random number generator, however we currently have no compelling reason to do this. This limitation could pose a problem when attempting to replicate results, as non-idempotent methods will generate different results each time they are called.

Given the number of steps involved in passing local best values around the network and between objects (and threads), the peer-to-peer implementation of the Particle Swarm Optimization algorithm will not react as quickly to change as the centralized algorithm on a uniprocessor machine. This is a concern, but it is notable only in the speed of the algorithm's convergence and not on the quality of the final result. That is assuming that users will run simulations based on given minimization criteria, e.g. "Run until f(x) is below 500.0". If the users are forced to bound their simulation ensembles on computation time, then this may be an issue. We currently do not consider the impact of this on the overall system.

C. Simulation

The simulation is our black-box evaluation function. The SimulationManager passes a vector to the Simulation and the Simulation returns a scalar value. The generic structure of this solution allows a broad range of simulations to be evaluated using this framework. Should a research have a custom-tailored implementation of the PSO, suited to the needs of the associated simulation, our modular design will allow her to replace the provided implementation.

In the Patisserie reference implementation, we provide 3 different implementations of the Simulation class. Two are canonical examples from optimization and the third is a research simulation. Researchers wishing to incorporate an existing simulation into Patisserie should consult the Patisserie API JavaDocs.

The two reference functions from optimization are DeJong's Sphere, Equation 2, and the Rastrigin function, Equation 3. Figure 4 shows the Rastrigin function in 2 dimensions. This function is used to demonstrate the effectiveness of optimization algorithms. It is an attractive choice given its infinite local minima and single global minimum. DeJong's sphere is a relatively simple function that is routinely used to measure the performance of optimization algorithms. In our implementation, we evaluate both functions in 20 dimensions.

$$f(x) = \sum_{i=1}^{n} (x_i^2) \tag{2}$$

$$f(x) = \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i) + 10) \tag{3}$$

These functions were chosen for their simple implementation and their level of challenge to the optimization algorithm. Also, owing to their prevalent use in optimization literature, we have a reference point with which to compare our algorithm.

In addition to the detailed studies done with the optimization functions, we present results using a research simulation, Schoenharl's Agent-Based Neural Network Simulator[4]. This biologically inspired neural network simulator is used to explore the various structures that can emerge using only local communication. The simulation currently exists in an 11-dimension parameter space, with a mixture of discrete and continuous values. Potential applications of this research include building large neural networks that scale better than current approaches, as well as suggesting possible methods that biological neural networks self-organize. In our framework, the objective function is the minimization of several topological parameters on the generated neural network.

The Agent-Based Neural Network Simulator is useful in that not only benefits from the PSO steering algorithm, but also in the Query API's ability to store and retrieve results. The power of Patisserie yields a noticeable benefit in terms

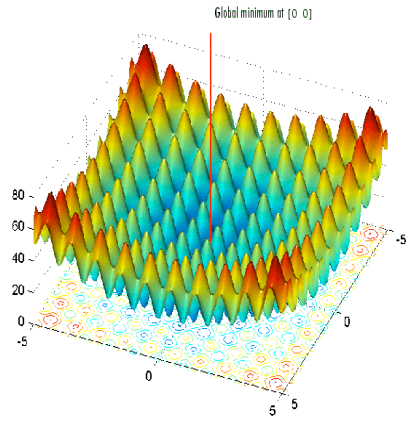

Fig. 4. The Rastrigin Function in 2 Dimensions

of simplifying the researcher's regime of running simulations and collecting data.

D. ResultsManager

The ResultsManager implements the MessageReceiver interface and extends the PastryApp class. This gives the ResultsManager the ability to run as a Pastry Application and send and receive messages over the Pastry peer-to-peer network. Being a Pastry Application exposes several useful aspects of the Pastry API to the Patisserie system. Among these are the ability to detect when Pastry has changed the leaf set (neighbors), to lookup nodes in the network and to view the network routing table.

As mentioned above, ResultsManager maintains a reference to the SimulationManager. When ResultsManager receives a message with a neighbors' new personal best, it will compare it to the local best value stored in SimulationManager, and if the new value is better, it will replace the value. It may seem like there is potential for error here. Potentially ResultsManager could read the current local best value in SimulationManager, then SimulationManager could finish running a simulation and read the local best value and pass it on to the PSO, and after that ResultsManager could update the local best value. This is certainly a possibility, however there would be little noticeable effect on the overall system. It would merely delay the updating of the PSO. Also of importance is to consider whether SimulationManager could read the local best vector while it is being written by the ResultsManager. This situation might have the potential to corrupt data, so we have synchronized the personal best vector so that updates to it are atomic.

E. Virtual Nodes

For scalability testing, acquiring the utilization of thousands of machines is not generally practical, so the solution is to use virtual nodes. Virtual nodes are full peer-to-peer nodes that reside within the same process; each has its own unique node identifier but network communication and CPU usage is multiplexed. Virtual nodes allow much larger networks to be simulated; however, one has to be careful not to introduce artifacts into the simulation by loading too many virtual nodes onto a single machine. Pastry provides support for virtual nodes by simply making additional nodes within the same process, and each Pastry node gets its own ResultsManager object for sending and receiving peer-to-peer network messages. The Patisserie framework creates a SimulationManager object running in its own thread for each virtual node. In terms of particle swarm optimization, each peer-to-peer node corresponds to a particle, so virtual nodes allow multiple particles to be executed on a single machine.

We can claim validity with results from virtual nodes as we leverage the power of threads and make use of the delay that we were forced to put in the evaluation process. Each SimulationManager will evaluate its current location and then sleep for a period of time. The length of the sleep dominates by far the computation time, thus most threads are sleeping most of the time. This allows the computation to be interleaved without putting a burden on performance.

VII. Results

We will show that our approach yields good performance, scalability and fault tolerance. Where certain simulations remain to be done, we have inserted discussion relating to experimental setup and evaluation criteria. Where evaluations have been run, we provide the results as well as discussion.

A. Performance Evaluation on Optimization Functions

Our first evaluation is a demonstration of the performance of the system on the DeJong's Sphere optimization function. We present the results to demonstrate the behavior of the working system. Figure 5 shows the trajectories of a sample of particles for a 64 virtual node Patisserie as they optimize the DeJong's Sphere function; this implementation of the function has a 20 dimension parameter space. Each particle is initialized to a random position in parameter space and, as the figure indicates, the particles converge in a relatively small number of iterations to a minimum. The lines in the figure represent the evaluation of a particle's location over time. At the beginning of the simulation, the particles are at relatively "high" points in state space. As the simulation progresses, the individual particles move to positions with a lower evaluation in state space. This behavior is encouraging, as researchers will not want to run many iterations of a long running simulation. It is possible to decipher from the picture that several nodes are

in communication during the optimization routine. Several of the lines display interesting similarities in terms of slopes and curves.

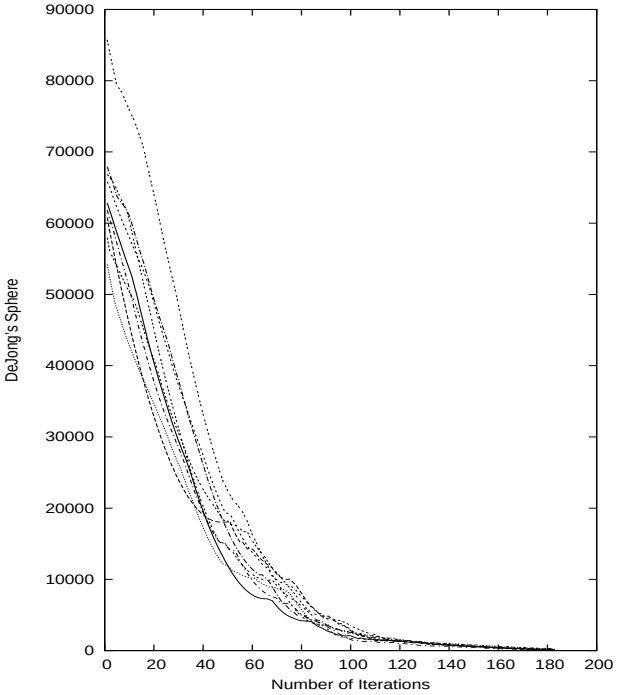

Fig. 5. Optimization of DeJong's Sphere by 64 virtual nodes

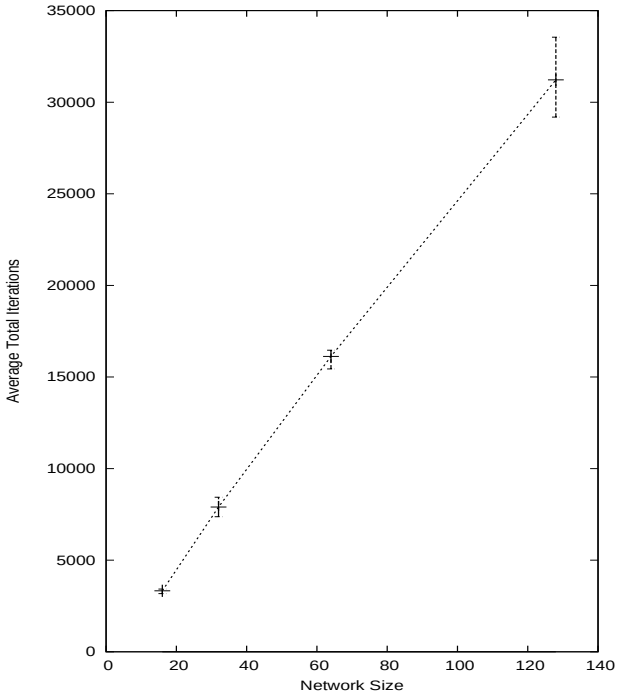

Fig. 6. The average number of total iterations with error bars of five replications for optimization of DeJong's Sphere on a Patisserie network of 16, 32, 64, and 128 virtual nodes across 16 machines.

Figures 6 and 7 shows the total number of iterations, or PSO evaluations of a simulation, performed by all of the particles versus the number of virtual nodes in the network for two different simulations. The simulation for Figure 6 optimizes a twenty parameter DeJong's Sphere function, and the simulation for Figure 7 optimizes a twenty parameter Rastrigin function. Both simulations are throttled with a five second delay to prevent flooding of the network, and they are run until one of the virtual nodes reaches a sufficiently small threshold value. The plots indicates that a linear increase in the number of simulations performed is obtained as the number of virtual nodes is increased; this provides an initial indication that the system can be scaled to a larger number of nodes. The base assumption is that the execution time for a simulation is significantly greater than the latency for peer-to-peer network messages; therefore, storage of simulation results in the peer-to-peer network does not significantly impact performance and scalability. If this assumption is violated then the network will be flooded with storage messages; and while many simulations will continue to be executed, the peer-to-peer network messages for storage of simulation results will get dropped as send and receive queues overflow. In the future, we will investigate guaranteed delivery of storage messages which may adversely affect the number of simulations that can be performed.

Figure 8 shows a different viewpoint versus Figure 6 with the average number of iterations per node before the threshold value is reached. The plot seems to indicate that the number of iterations per node actually increases then tapers off as more nodes are added to the system; however, the error bars show significant overlap which makes that conclusion questionable. A plausible explanation is that when one node has reached the threshold, it takes longer to stop all of the nodes in a larger network then in a smaller network; and in the meantime, those nodes are continuing to run simulations. We need to investigate further to determine if this behavior is an artifact of our simulation runs or intrinsic to Patisserie. One would expect that adding more nodes would make the convergence to the threshold faster, but this is not a guarantee with the PSO algorithm. However, Figures 6 and 7 indicate encouraging results that adding more nodes to the network does provide more exploration of the parameter space, due to more simulation runs, which is one of the primary goals of Patisserie.

B. Evaluation Platform

All the testing and development was conducted on the NOM Research Group Computing Cluster (NGCC). NGCC is a 10

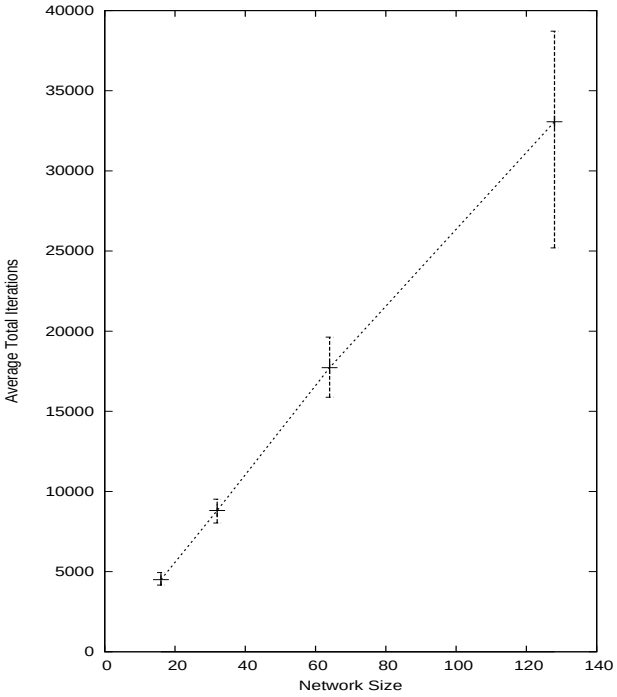

Fig. 7. The average number of total iterations with error bars of five replications for optimization of Rastrigin function on a Patisserie network of 16, 32, 64, and 128 virtual nodes across 16 machines.

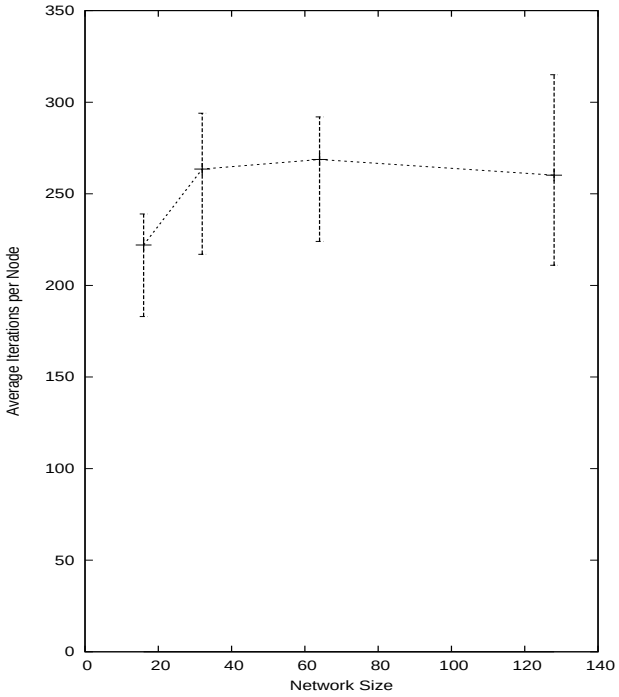

Fig. 8. The average number of iterations per node with error bars of five replications for optimization of DeJong's Sphere on a Patisserie network of 16, 32, 64, and 128 virtual nodes across 16 machines.

node cluster comprised of 7 simulation servers (dual 650mHz Pentium III, 2GB RAM), 2 database servers (1 dual 3.2gHz Xeon, 2GB RAM, 1 dual 650mHz Pentium III, 2GB RAM) and a file server (dual 650mHz Pentium III, 2GB RAM) with attached 1.2TB storage. All machines run RedHat Advanced Server 3.0. The system runs proprietary scheduling and load balancing software, as well as the Condor system. We used Condor's scheduler to distribute and manage our simulations.

Data collection was conducted on the Cooperative Computing Lab's combined Condor cluster. It is comprised of the 8 node Wombat cluster, the 12 node CCL cluster, the 8 node, dual-processor GIPSE cluster and various workstations around the department. We restricted our experiments to the Wombat and CCL machines, as there appears to be an issue with our threaded code running on the dual-processor machines.

VIII. Related Work

The AppLeS Parameter Sweep Template [1] is middleware that attempts to simplify the use of computational grids for Parameter Sweep Applications. Its focus is on scheduling simulations in a grid environment. The application consists of four different components: a Controller that acts as an interface between the user and the Scheduler, a Scheduler with several heuristics for scheduling jobs, an Actuator, which implements the schedule and attempts to move input/output data and processes around to minimize communication and finally a Meta-Data Bookkeeper, which stores information on observed and predicted resources, ie network performance, application state, predicted load, etc. The AppLeS Scheduler is better suited to the task of parameter sweeps, however, it has a single point of failure and does not display the kind of scaling behavior of the distributed PSO. Our project addresses a niche that is not well served by APST, applications with high parallelism and low input data requirements, but which may require steering during the running of the application. AppLeS is designed to integrate well into a grid environment, whereas Patisserie can be used as a stand-alone component. The implications of this are that AppLeS can take advantage of existing grid resources, but new users must become masters of the grid environment in order to use it. Patisserie does not require that users become familiar with grid tools and software, but places the burden on the user of finding and accessing computational resources.

The Organic Grid approach [3] uses an overlay network to create a tree structure, where the root node is essentially the master of the simulation and determines which jobs will be scheduled on the child nodes. The child nodes, as in a self-scheduled workqueue [11], query their parent for jobs to execute. This approach, while interesting, suffers from many of the same problems as a centralized approach, namely

any adaptation must take place at a central location and be propagated down the tree. Our approach does not suffer from this limitation, nor does it introduce the attendant problems with a centralized approach: costly workarounds to recover when a parent node dies and limited scalability. Additionally, there is not an intuitive mapping between the tree-structured hierarchy and the peer-to-peer network.

The latest contribution to this area is "Self-Organizing Monte Carlo Optimization in Peer-to-Peer Networks" [2]. In this application the authors present a parameter sweep application scheduler that is very similar to our proposed architecture. The authors designate each machine in a grid as a computational agent and connect these agents using an overlay network. They use a given algorithm to ensure that the overlay network displays several desired characteristics: it is a scale-free network which has a small overall distance between nodes in the graph, and it is resilient in the face of random node failure. In this network, each agent begins working on some piece of the optimization problem. When it finishes its piece it compares its result to its neighbors. Agents discard bad solutions and replace them with better solutions from their neighbors and use this as a starting point in a local search. In this way good solutions can propagate throughout the network via local communication.

Although the SOMCOP2PN approach appears similar to Patisserie, there are significant differences. The authors note that their Monte Carlo optimization can be viewed as similar to particle swarm optimization; however, their computation network converges to a single solution while PSO continues to randomly explore solution space which is more appropriate behavior for parameter sweeps. Additionally, scale-free networks have the property that some nodes are hubs with a large degree which means they have a disproportionate amount of communication, and this property is a hinderance to scalability. The authors suggest that this condition can be alleviated by imposing a maximum degree; though this causes a deviation from a scale-free distribution. By using the structured network overlay provided by peer-to-peer networks, we can guarantee that no single node has a inordinate number of neighbors and still maintain a short distance between any two nodes in the network. Our Query API makes the retrieval and management of results feasible.

IX. Future Work

In the future we intend to develop Patisserie by implementing a more sophisticated version of the Particle Swarm Optimization algorithm. Possible modifications include adding an adjustment to the simple velocity maximum, such as the inertia weights of Shi and Eberhart [12].

We intend to demonstrate the fault-tolerant behavior of the system by running simulations and subjecting the network to random node failure. Literature on PSO lead us to believe that the system should perform well and display graceful degradation. We would also like to explore the effects of network partitioning on the system. The PSO should continue to perform, assuming that the partitioning doesn't leave nodes without PSO neighbors. Rejoining a Patisserie network that is separated by a network partition would be a function of the underlying Pastry framework, but is an interesting consideration.

We would like to study the effects of indirection and storage replication on the performance and scalability of the Query API. Important results should be replicated on the network, both for availability and performance (in terms of latency).

After some interesting results are found, the researcher may wish to run multiple replications using different random seeds for the simulation to statistically validate the results of the single run. We intend to implement a *Replicate* message type that specifies a parameter set, a number of replication runs to perform, and optionally a set of random seeds to use; and these replication simulations will be scheduled in the Patisserie network. The raw data files from each simulation run can have the random seed as part of its identification, so each replication run will create separate raw data files that can be analyzed. Beyond support for replication runs, we are investigating the ability to allow additional metrics and computation to be performed on simulation results without requiring a new version of the simulation or framework executable to be distributed.

Similarly we would like to investigate the effect of modifying the neighborhoods so that they more closely correspond to communication latency between nodes. Nodes with lower communication latency should be more likely to be neighbors than nodes with high latency. The effect of this modification would be to reduce the cost of communication in the system. However, one consequence of this would be to expose the algorithm to site failures. A power outage or shut-down due to a cooling system malfunction would affect nodes in the same location. Nodes that were grouped by location would suffer with the loss of (potentially) entire neighborhoods, whereas if the neighbor relations were distributed evenly, many neighborhoods would suffer the loss of a few neighbors. Simulations are warranted to explore this issue and see if it would affect the performance of the system.

Study the effects of alternate neighborhood topology on system performance. We would like to examine the performance of the PSO on networks exhibiting small-worlds topologies, as well as scale-free and exponential networks.

Finally, it would be enticing for researchers to have a visualization tool for monitoring the progress of the system. One of us (Schoenharl) has created a visualizer for the uniprocessor PSO implementation. The visualization consists of a 2-dimensional grid, where each grid location represents a particle. Each grid square is color coded with a color whose

intensity matches the objective function evaluation of that particle's personal best. This visualizer offers an interesting glimpse into the functioning of the system, and the color coding is an excellent method of concisely conveying a large amount of information.

X. Conclusion

A. Benefits of Patisserie

We have demonstrated the Patisserie framework for conducting parameter sweeps in a heterogeneous peer-to-peer environment. Our demonstration system utilizes several canonical functions from the optimization field. Based on our results, we claim that we have met our goals of scalability and fault tolerance and that we also have demonstrated the system under a representative load.

Our results show that the Particle Swarm Optimization algorithm is an excellent match for our simulation environment. It is robust, displays graceful degradation and scales well. Its performance as a general-purpose optimizer make it an attractive choice in a context where a variety of simulations and parameter spaces must be evaluated. Our Query API provides simple, intuitive commands that are powerful and extensible and aid considerably in the evaluation, storage and management of simulation data. They display favorable scaling properties and are resistant to random node failure.

We feel that our system goes beyond competing projects. Our design is more fault tolerant and maps better to our project domain than the Organic Grid. Our scheduling algorithm is more robust than AppLeS, and our simple design and interfaces make our system more attractive to researchers who desire a working system and not an education in grid computing. Finally our design rivals that of the "Self-Organizing Monte Carlo Optimization in Peer-to-Peer Networks", mainly through our use of the proven PSO algorithm and the advanced query API. Overall our system is simple, easily extensible, highly robust and ideally suited to the task of running simulations in a peer-to-peer environment.

B. Limitations

There are several issues that we have not addressed, as well as several overall limitations of our framework. The validity of our work hinges at least in part on the assumption that virtual nodes are a reasonable facsimile of real individual nodes. This may not be the case, as network latency and cost will be magnified considerably in a geographically diverse system. The cost of network latency and speed in the reference simulation may be minimized if several neighbors a running on the same CPU and communicating over the loopback interface. In a real system these values may be orders of magnitude higher for each communication, compounding the problem. Further testing and exploration are required before any definitive claims concerning network performance can be made. That said, results for the iteration to message ratio should remain valid and give us a guideline should we get better information on network speed and latency.

The framework as a whole is useful for simulations that have one output that needs to be minimized. It is possible that Patisserie could be modified to satisfy multiobjective functions, but this is well beyond the scope of our current work. Additionally, we need better tools for running and managing jobs in a batch system, such as the SunONE Grid Engine that schedules jobs for HPCC and BOB. Finally we need to prove with a compelling example that results similar to our Query API could not be achieved by merely running simulations and storing the results in a large, fast database.

Given the independent nature of the Particle Swarm Optimization scheduling algorithm, it is not easy to specify a priori an ensemble of parameter sets to evaluate. This is one area where Patisserie lags behind competitors such as AppLeS and the Organic Grid. This functionality could be built on top of the Query API, but it would be a bit of a hack and goes against the basic principles of the system.

XI. Acknowledgements

The authors would like to thank Professor Greg Madey for the use of the NOM Research Group Computing Cluster.

This work was supported in part by a Fellowship provided by the Arthur J. Schmidt Foundation.

References

[1] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, "The AppLeS parameter sweep template: User-level middleware for the grid," in *Proceedings of Super Computing 2000*, pp. 75–76, 2000.

[2] J. Saramaki and K. Kaski, "Self-organizing monte carlo optimization in peer-to-peer networks," 2004.

[3] A. J. Chakravarti, G. Baumgartner, and M. Lauria, "The organic grid: Self-organizing computation," in *Proceedings of ICAC 2004*, 2004.

[4] T. Schoenharl and G. Madey, "Using agent-based modeling in the simulation of self-organizing neural networks," in *Proceedings of the Workshop on Agent/Swarm Programming (WASP) '03*, pp. 27–32, October 2003.

[5] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a common api for structured peer-to-peer overlays," in *Second International Workshop on Peer-to-Peer Systems (IPTPS 2003)*, 2003.

[6] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pp. 329–350, Springer-Verlag, 2001.

[7] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *International Conference on Neural Networks, IV*, pp. 1942–1948, 1995.

[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 161–172, ACM Press, 2001.

[9] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 41–53, Jan. 2004.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 149–160, ACM Press, 2001.

[11] T. Hagerup, “Allocating independent tasks to parallel processors: an experimental study,” in *Journal of Parallel and Distributed Computing*, vol. 47, pp. 185–197, December 1997.
[12] Y. Shi and R. Eberhart, “A modified particle swarm optimizer,” in *Proceedings of the IEEE International Conference on Evolutionary Computation*, 1998.

XII. Biography

Fig. 9. Scott ”You Said This Would Only Take Two Hours!” Christley

Scott Christley is Mr. EverQuest. He spends the majority of his day wasting his colleagues time recounting his adventures in the virtual world. The included shot is Scott hard at work running simulations after a long night of EverQuest.

Fig. 10. Tim ”Behold the Effect of 9 Months Without Sunlight!” Schoenharl

Tim Schoenharl is a 14th Century Shaolin monk trapped in a 21st century hacker’s body. He works hard at maintaining his heroin chic physique by guzzling coffee and incessantly clicking his ballpoint pen. He hopes to one day finish writing his Master’s Thesis.