A Scalable Framework for Parallel Discrete Event Simulations on Desktop Grids

Alfred Park, Richard Fujimoto

Computational Science and Engineering Division, College of Computing, Georgia Institute of Technology
266 Ferst Drive, Atlanta, Georgia 30332-0765, USA
{park,fujimoto}@cc.gatech.edu

***Abstract*— Utilizing desktop grid infrastructures is challenging for parallel discrete event simulation (PDES) codes due to characteristics such as inter-process messaging, restricted execution, and overall lower concurrency than typical volunteer computing projects. The Aurora2 system uses an approach that simultaneously provides both replicated execution support and scalable performance of PDES applications through public resource computing. This is accomplished through a multi-threaded distributed back-end system, low overhead communications middleware, and an efficient client implementation. This paper describes the Aurora2 architecture and issues pertinent to PDES executions in a desktop grid environment that must be addressed when distributing back-end services across multiple machines. We quantify improvement over the first generation Aurora system through a comparative performance study detailing PDES programs with various scalability characteristics for execution over desktop grids.**

I. Introduction

Distributed computing with respect to donating idle processing cycles and volunteer computing have become well-received in recent years due to the success of projects which began with the Great Internet Mersenne Prime Search and evolved with distributed.net and SETI@home [1]. The popularity of these efforts has expanded to include Graphics Processing Units (GPUs) and even video game consoles such as Folding@Home for the Sony PlayStation®3. These projects utilize spare processor cycles or share processor time at the user's discretion to execute portions of a larger task.

Many PDES applications are computationally intensive and the allure of harvesting computing cycles from trusted workstations in a desktop grid is intriguing. Although PDES computations are constrained further than typical projects in this distributed computing space, these applications can leverage and benefit from the ever increasing ubiquity of computational devices [2]. Due to the special needs of these programs, high performance middleware to leverage desktop grid environments are a necessity in providing scalable distributed simulations. The first generation Aurora system [3] proved that this environment was suitable for various PDES applications that were computationally intensive per work unit lease. However, limitations inherent to the design and web services messaging middleware were unfavorable for large scale simulations and PDES programs which shipped large amounts of state vector data or messages per work unit lease or exhibited a high work unit return rate.

The Aurora2 system is an effort to overcome the limitations intrinsic to its predecessor. This new architecture features a distributed back-end infrastructure with multi-threaded services to allow for large volumes of requests to be serviced concurrently. An enhanced multi-threaded high performance client is also introduced to reduce overhead associated with PDES executions over desktop grids. Furthermore, the entire Aurora2 system utilizes a faster sockets-based communications middleware. These enhancements provide significant performance improvement for large-scale PDES applications compared to the first generation Aurora system.

The remainder of this paper is organized as follows. We next describe the special needs of PDES codes with respect to master/worker executions. Issues encountered with the first Aurora system are described and the solutions proposed in the Aurora2 system are discussed. This is followed by a detailed description of the Aurora2 system components. The performance of the Aurora2 system is compared with its predecessor in the performance study section and is followed by related work, a discussion of future work, and conclusions.

II. PDES and the Master/Worker Paradigm

The master/worker paradigm is a mechanism in a distributed computing environment for dividing up potentially large computations into smaller portions of work that can be tasked out to a pool of workers operating under the direction of a master controller. Although this concept has been well-studied in the general distributed computation literature, it has not been explored fully when coupled with the requirements of parallel discrete event simulations.

In conventional PDES systems, a simulation consists of many logical processes (LP) that communicate by exchanging time-stamped messages as the simulation progresses. The first important difference in a master/worker PDES system is that the LP state, comprised of program variables, must be tracked and stored. Some general distributed computing projects only require a simplifying result at the end of computation; however, a PDES system must maintain the state of the LP which may be leased to a different client at a later time. Moreover, work units must be properly constrained for execution as processing events and messages arbitrarily into the future may violate the local causality constraint (LCC). Consequently, conservative or optimistic synchronization must be employed alongside LP and work unit management in a master/worker PDES system. PDES executions which preserve the LCC are referred to as conservatively synchronized simulations in which only the last stored LP state is needed. An LP state history is required

1-4244-1560-8/07/$25.00 © 2007 IEEE

for simulations employing optimistic synchronization which allows the LCC to be violated but must provide means to rollback the simulation and restore a prior state. Furthermore, the PDES system must ensure that time-stamped messages are delivered in order to the proper destination LP to preserve the LCC. Therefore, in addition to correctly ordering messages sent between LPs, these messages must be maintained for future delivery to any LP or collection of LPs that are leased as a work unit to an available client. In optimistic synchronizations, additional requirements are needed such as storage of processed uncommitted messages, i.e., processed messages with a time stamp greater than Global Virtual Time (GVT) and handling anti-messages if the rollback mechanism requires them. These key differences and requirements between general distributed computing programs and master/worker PDES is the impetus for the Aurora system.

III. A Scalable Desktop Grid PDES Framework

The Aurora system is a prototype framework to deliver high throughput PDES performance using a computational infrastructure that is vastly different from those typically used for these codes. The first Aurora system provided a means to test feasibility of PDES on such infrastructures. However, the system could not deliver high performance for many large-scale simulations. This was a major issue as often the decision to parallelize a discrete event simulation is to gain speedup over a serial implementation. Therefore, the Aurora2 system includes significant design changes over its predecessor to further improve the performance of large scale simulations as a principle goal.

A. First Generation Aurora: A Web Services Approach

The first generation Aurora system utilized the gSOAP toolkit [4] to provide web services in the underlying architecture to transport work unit state and messages between clients and the server. The decision to use web services was to provide standardized and interoperable components of the system. The gSOAP toolkit was chosen due to its high performance and low latency characteristics [5] while providing web services for C/C++ programs.

While performance under the first system was acceptable for many test cases, large scale simulations would incur too much overhead thus diminishing throughput gains achieved by capturing cycles on idle desktops. Although gSOAP is a high performance web services toolkit, these services are inherently slower than traditional network communication middleware such as sockets. Human-readable XML encoded messages increase both message size and processing time.

Although multiple servers were used, these additional servers were intended for fault recovery through replication and not to enhance performance. This initial web services-based Aurora system used a single threaded, single server design. This can lead to bottlenecks in a large scale distributed simulation. For instance, if a simulation exhibits a high work unit return rate or large LP state buffers, the server would not be able service existing requests quickly enough to avoid delaying the processing of new incoming requests.

With these limitations to the first generation Aurora system, it was apparent that significant changes to the core architecture were needed to accommodate large scale simulations with potentially hundreds or thousands of clients. The single server design of the first generation Aurora could be modified to accommodate extra load by multi-threading the service, however, this would only provide limited scalability improvements. The system lacked the architecture to support scalable high performance PDES executions.

B. Aurora2: Scalable Desktop Grid Services for Large Scale PDES Programs

The Aurora2 architecture is a departure from the first generation design in the following important areas: concurrent simulation and simulation replication support (i.e., performing a PDES execution multiple times for data and output, not for fault tolerance), communication middleware, and multiple functionally different multi-threaded state servers.

One of the important uses of a high throughput simulation system is to not only run a distributed simulation across many processors efficiently but also to create replicated runs of large scale simulations. The Aurora2 architecture is flexible and allows distinct simulations requiring multiple replications to be run under a single back-end instance composed of any number of storage services.

The web services communication framework in the first generation Aurora system was replaced with a traditional sockets-based framework allowing for higher performance data transfers that account for a non-trivial portion of the overall overhead. Serialization of data to a compact binary format lead to improved throughput rates and reduced CPU load while processing messages.

The area which received the greatest performance improvement was the change from a single server design to multiple functionally different multi-threaded servers that could be dynamically allocated. The two main contributors to overhead from work units come from state vector storage and message buffers which are proportional to the amount of state and emitted messages. The functionality of the storage server for these entities was separated into a work unit storage server and a message state server. This allows dynamic deployment of the Aurora back-end state storage server tailored for a particular simulation or for large scale simulations in general.

IV. The Aurora2 System

The Aurora2 system can be logically divided into three parts: back-end services, simulation package management, and the client that contains the simulation application, as shown in Fig. 1. The solid arrows denote data exchange. For the Aurora2 system in a desktop grid environment, we assume that the services and clients exist on the same side of a firewall or that steps are taken to allow communications to such services which span across a firewall such as SSH tunneling. The work unit pull-based system does not require clients to act as servers, thus client requests are unidirectional in nature (e.g., client to server only).

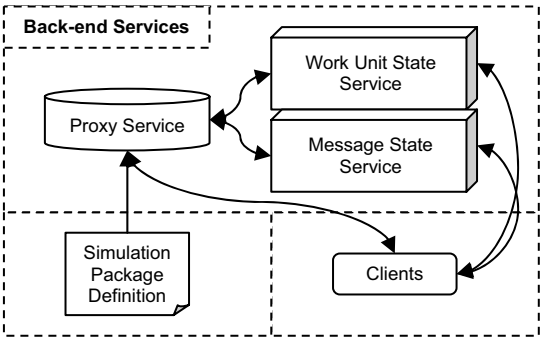

Fig. 1 Interaction Overview of Aurora2 Components

A. Aurora2 Proxy Service

The proxy service is the central core controller and is regarded as the logical master in the master/worker paradigm. The proxy contains metadata for all simulation packages and oversees the other two back-end components: the work unit state service and message state service. Internally, major metadata portions are contained within managers as shown in Fig. 2. Incoming connections are accepted by the connection manager thread leveraging the communications manager as needed for servicing the network. Valid connections are then passed to a thread in the generic worker thread pool which is loaded with data specific to the request and then is serviced by invoking applicable methods in one of the metadata managers: simulation package manager, work unit server manager, or the message state server manager.

The simulation package manager contains two main metadata tables and the time management module. The client metadata contains information about the client such as IP address, global unique client key, and any outstanding work unit leases. Information pertaining specifically to the simulation LPs such as simulation times, consistency of state vectors, and reverse lookup tables to work unit state and message stores are contained in the LP metadata. The time management manager is instanced within the simulation package manager which controls global simulated time. This manager contains methods to compute safe processing bounds much like a Lower Bound Time Stamp (LBTS) or GVT as well as an upper processing bound for a designated LP.

Time management, can include a mixture of conservative (blocking) or optimistic (rollback-based) schemes. The time management mechanism can be chosen at runtime by the simulation package definition. Multiple different time management schemes can be used for separate simulations all running on the same back-end instance. The current version of Aurora2 includes a centralized conservative time management mechanism. If an Aurora system were to allow multiple proxies through a front-end load balancer, distributed time management would need to be addressed.

The final two managers inside the Aurora2 Proxy handle metadata for the ephemeral data stores involving work units. The work unit server manager oversees work unit server instances storing information such as server IP address, port number, memory allocation, and listing of simulation packages the server is hosting. The message state server manager stores similar information but is instead dedicated to handling metadata only from message servers and not work unit state vector servers. The decision to keep the managers separate is intended for upcoming additions pertaining to performance enhancements detailed in the future work section.

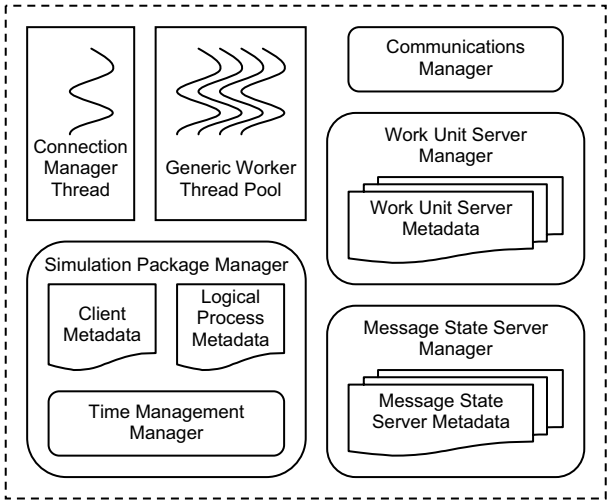

Fig. 2 Components of the Aurora2 Proxy Service

B. Aurora2 Work Unit State and Message State Services

Due to PDES applications inherently belonging to a different class of problems typically associated with distributed computing programs on desktop grid architectures, it is necessary to add two additional storage services. State vectors are modified as the LP advances in simulation time. Unlike typical Embarrassingly Parallel (EP) class of problems, the simulation time must be limited as to not pass an upper safe processing bound to preserve the LCC. When the simulation time reaches this end time limit, the final state vectors and any messages generated during the simulated time must be stored for a future work unit lease to another client.

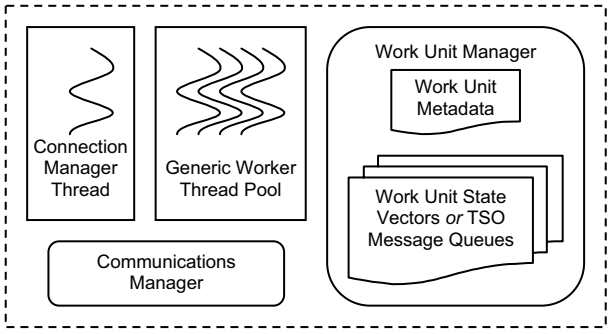

Fig. 3 Components of the Aurora2 State Services

The Aurora2 work unit state and message state services provide a distributed storage system for collecting pertinent states. Due to the possibility of a high volume of clients that may upload large state vectors and messages, these services must be designed for high performance. Following a similar design as the Aurora2 proxy service as shown in Fig. 3, these state services are multi-threaded and any number of them can be instanced for improved scalability.

Depending upon the functionality of the service, the work unit manager stores either state vectors or a Time Stamp Ordered (TSO) message queue. State vectors are stored as application-defined contiguous blocks of memory that are packed and unpacked by simulation-specific routines overwritten by the Aurora2 clients. Similarly, packaged messages are stored in their respective destination LP TSO message queue.

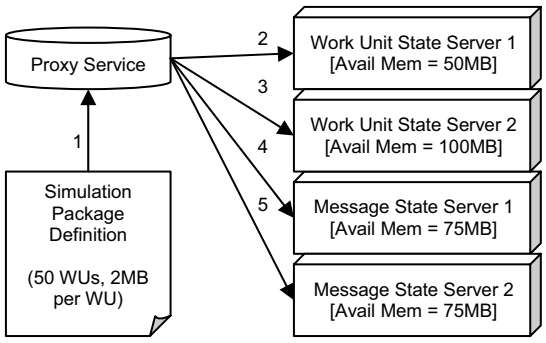

Fig. 4 Example Work Unit Distribution

C. Simulation Packages

With the addition of concurrent simulation support in Aurora2, a specification was needed to create a simulation instance on the back-end services. The simulation package definition accomplishes this by providing initial metadata about the distributed simulation such as the number of LPs as work units, a lookahead connectivity graph, simulation begin and end times, and deadlines in wallclock time per work unit lease. In addition to these initial runtime parameters the simulation package definition may upload any initial LP states.

After the simulation package definition is uploaded to the Aurora2 proxy service, the necessary metadata tables and allocation of resources is done prior to client execution such as the distribution of work unit storage load. For example, Fig. 4 illustrates uploading the simulation package definition to the Aurora2 proxy service (step 1) where the simulation contains fifty total work units with an anticipated memory requirement of two MB per work unit in both state vector and message storage. The proxy service will then query its internal tables for available work unit and message state servers. 25 work units are allocated to each in steps 2 and 3 to work unit state servers 1 and 2 respectively. For message state storage, 37 work units are allocated in step 4 and the remaining 13 are allocated in step 5. Initial work unit distribution, dynamic load balancing, and memory distribution and re-distribution techniques are discussed further in the future work section.

D. Aurora2 Client

The Aurora2 back-end provides the necessary infrastructure to run PDES applications in a distributed fashion over desktop grids. However, the actual simulation computation is done by the Aurora2 clients. After communicating with the Proxy service, these clients contact the proper back-end state services to download simulation state and associated messages per work unit lease. Computation is done locally and independent of other clients. Once the designated client simulation period is processed, the final state and messages are uploaded to the work unit state and message state servers. This process can repeat for a specified number of times, until simulation end time is reached, or if interrupted by a user.

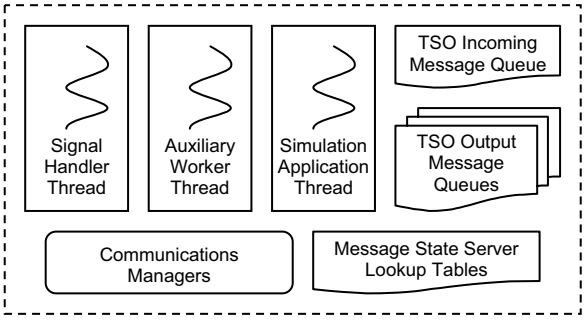

Fig. 5 Components of the Aurora2 Client

The Aurora2 client is multi-threaded to allow unimpeded execution of the simulation application code while Aurora2-specific tasks can be run concurrently. Fig. 5 shows a schematized view of the Aurora2 client. The auxiliary worker thread can perform background processing tasks such as message state server location lookups while the simulation is running.

The auxiliary thread is particularly important with regard to inter-LP messaging. When a message is generated during traditional PDES executions, the message is usually immediately sent to the recipient LP. In a master/worker desktop grid environment, connectivity between clients is not assumed; therefore generated messages are buffered at the client in the TSO output message queues. Due to the distributed nature of the Aurora2 message state service, lookups must be performed to ascertain the location of destination LPs (work units). This information is gathered into the message state server lookup tables which are necessary for the work unit finalization task that runs after the simulation computation completes.

Other components are standard support code for clients such as the signal handler thread and the communications managers which are used to exchange information with the Aurora2 proxy, work unit state server, or the message state server. The TSO incoming message queue is used to store messages destined for LPs contained within the downloaded work unit for the designated simulation execution window.

E. Work Unit Lifecycle

The Aurora2 client performs a series of steps to download pertinent metadata, un-packaging of various data, simulation execution, re-packaging of state variables and messages, and finally uploading results back to the proper back-end services. These steps can be characterized as the Aurora2 work unit lifecycle. The lifecycle is comprised of three major steps, excluding the one-shot initialization phase. This initialization phase includes thread initialization, signal handler setup, communications manager handshaking, and command-line parsing. The three major cyclic steps for an Aurora2 client are: unit request and download, simulation application computation, and work unit finalization and upload. The

client may fail at any point during these steps. A work unit that is not returned within its wallclock deadline lease period due to client failure is rolled back and leased to a different client. These steps are similar to conventional volunteer computing lifecycles. However, additional PDES requirements modify the details significantly. A general overview of the process is presented in Fig. 6.

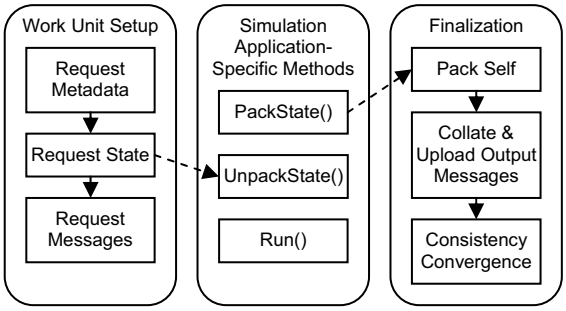

Fig. 6 Overview of the Work Unit Lifecycle

The work unit request and download phase is comprised of six steps. The first step is to contact the proxy and receive a client key authorization. Client keys are unique identifiers issued to the clients and is used by the client to identify itself in future communications. Once a unique client key is issued, the second step is for the client to perform a work unit request where the proxy may or may not lease a work unit to the requesting client. If a work unit is available to be leased, associated metadata about the work unit is downloaded from the proxy service. This third step also includes creating any necessary internal data structures to host the pending work unit. In the fourth step, the client contacts the designated Aurora2 work unit state server for the packed state vectors from information received in step three. Similarly, in step five, the client contacts the appropriate Aurora2 message state server and downloads messages destined for LPs contained in this work unit. The sixth step sends the downloaded state vectors to the application-defined `UnpackState()` method to initialize the LPs with the proper state variables. Likewise, the messages downloaded in the previous step are unpacked and populated in the incoming TSO message queue.

After the work unit setup has completed, the application-defined simulation execution method `Run()` is invoked and the simulation runs for the entirety of the simulation execution window as specified by the request phase of the work unit setup. The final work unit phase is initiated when the simulation computation reaches the end time.

Work unit finalization and upload consists of six steps. The first step calls the application-defined `PackState()` for packaging the final state into a contiguous area of memory for upload to the work unit state server. In the second step, the client packages any remaining unprocessed messages in the incoming TSO message queue which may have been generated through self message sends. These messages are generated during the simulation execution from any LP within the work unit to destination LPs residing in that same work unit. In the third step, the client initiates a consistency convergence to the Aurora2 proxy service. A consistency convergence locks down the work unit so it may not be leased again or updated in multiple lease scenarios. This allows an atomic work unit commit, preventing inconsistent updates of work unit and message states. The fourth step involves the client initiating an outgoing message collation process where future messages destined for work units which reside on the same physical message state server are packed together. This process leverages the data gathered by background message state server lookups performed by the auxiliary thread during simulation computation. This reduces the frequency of smaller message updates and allows the client to update groups of messages in large chunks. After this process completes, the fifth step includes uploading the final state and packed messages in step two to their respective servers. The sixth and final step is a verification of consistency convergence from the proxy that the process was completed successfully without errors on the Aurora2 back-end. During steps four and five, the Aurora state and message state servers send messages to the proxy specifying updates to their respective states. The proxy acknowledges these messages and keeps track of the consistency convergence process with the final result of the update returned to the client in the final step of the finalization process.

Client failures due to a crash (e.g., simulation code that causes the client to crash) are identified by the proxy not receiving a work unit finalization request before its wallclock deadline. These work units are reset by the proxy service and re-leased to other clients. Communications problems, such as a broken socket will cause the proxy to invalidate the lease to that particular client. User-based interrupts are handled through signal handlers and client unregistration requests, if required, are sent to the back-end system.

V. Performance Evaluation

In order to validate the scalability of this new architecture, the performance of the Aurora2 system was evaluated and compared with the first generation Aurora system. Two main applications were utilized in this study: a torus queuing network and a physics simulation modeling a one-dimensional hybrid shock using the piston method.

The original Aurora system and the Aurora2 system were both compiled with gcc 3.4.6. The original Aurora system was compiled with gSOAP 2.7.6c. Nodes designated as *Xeon* consist of dual processor Intel Xeon CPUs ranging from 2.8GHz to 3.06GHz with SMT (Hyperthreading) enabled and 1-2GB of memory. Nodes designated as *Pentium-III* consist of 8-way Pentium-III 550MHz CPUs with 4GB of memory. All machines use a GNU/Linux 2.6.9 kernel. Nodes are connected with Fast Ethernet, however, the *Xeon* and *Pentium-III* nodes reside on different LANs. Although the Aurora system is able to utilize non-dedicated machines, for this performance study it was necessary to isolate these machines from external factors such as variable user load to obtain pure timings to compare to the first generation system.

For the figures pertaining to this performance evaluation, *deferred* refers to the amount of wallclock time (seconds) a client spends waiting for a valid work unit lease from the

master. *Import* indicates the amount of time the client spends downloading work unit metadata, work unit state vectors, messages and the associated time spent in the application-dependent deserialization routine. *Finalize* is the time to perform the logical inverse of *import* where the work unit state and messages are serialized and consistency convergence is achieved on the back-end services for the returning work unit. *Application* denotes the time spent executing application-dependent simulation code. The *XwYm* indicates *X* work unit state servers and *Y* message state servers used in Aurora2 multi-server tests. The *XpYwZm* notation represents *X* proxy threads, *Y* work unit state server threads, and *Z* message state server threads in the multi-threading tests.

A. Torus Queuing Network Simulation

In these torus queuing network simulations, servers can be aggregated into subnets as LPs which can then be mapped to single work units. The first test examines the system performance of *coarse* work units where the number of servers within each work unit is high and the lookahead is favorable. This *coarse* scenario is configured as a 250,000 server 500x500 closed torus network with 625 partitions as available work units of 25x25 torus subnets. The internal links between servers within each work unit have a delay of 10 microseconds while delays between servers spanning work units have a delay of 1 millisecond. There are 20,000 initial jobs generated with 50% of the jobs destined for servers within each work unit and 50% destined for servers external to the work unit. Job service times were exponentially distributed with a mean of 5 microseconds. The following tests were performed with 42 clients consisting of 2 *Pentium-III* nodes and 13 *Xeon* nodes.

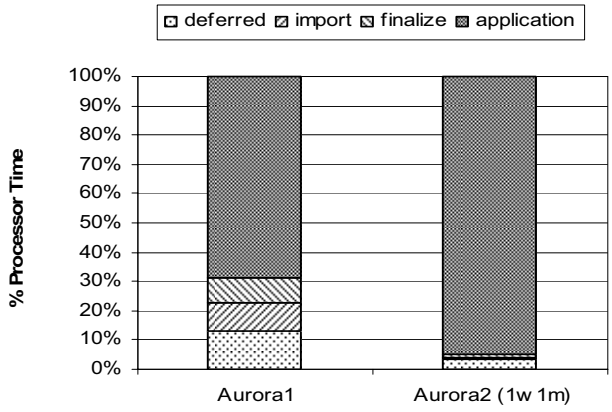

Fig. 7 *Coarse* Scenario Aurora Comparison

Fig. 7 shows the contribution of each portion of the work unit lifecycle to overall processor time usage. The Aurora1 system shows 68.6% of the time spent doing useful computation while 31.4% of CPU time is spent in overhead or waiting for a valid work unit lease. The Aurora2 case with one work unit server and one message state server shows approximately 94.9% of the processor time dedicated towards the application and only 5.1% for overhead. This particular simulation exhibited high concurrency and relatively large amount of time spent in simulation computation per work unit lease. Moreover, the amount of message state shifted per work unit lease averaged approximately 60MB. Two improvements in the Aurora2 system are shown here. First, the XML processing overhead for large transfers increases the import and finalize times for the Aurora1 system. Second, the optimized Aurora2 back-end services with multi-threading allows concurrent service requests while potentially large memory copies are being processed.

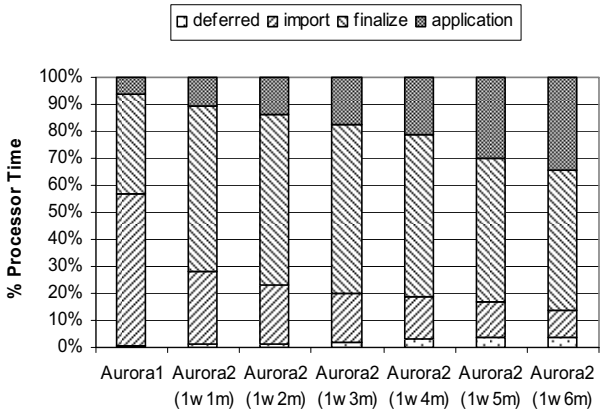

Fig. 8 *Fine* Scenario Aurora Comparison

The next test modifies the torus queuing network simulation parameters to unfavorable conditions with low concurrency even for traditional PDES systems. This *fine* scenario consists of a 150x150 closed torus queuing network containing 22,500 servers partitioned into 225 10x10 subnets which can be leased as work units. The external work unit to work unit delay has been reduced by an order of magnitude over the coarse scenario to 0.1 milliseconds. All other parameters remain the same. This test was run over 64 clients consisting of 4 *Pentium-III* nodes and 16 *Xeon* nodes.

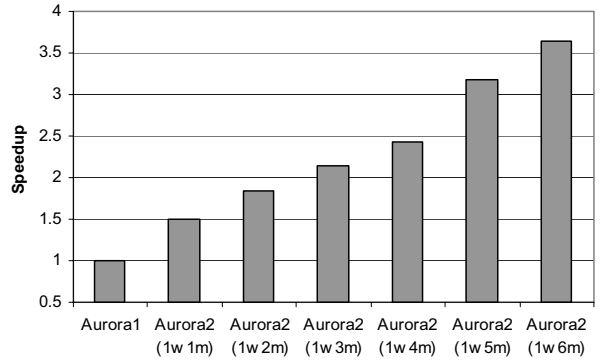

Fig. 9 *Fine* Scenario Speedup

This *fine* scenario exhibits lower concurrency in the model due to the reduction in lookahead compared to the *coarse* model. Fig. 8 shows a steady reduction in overhead as an increasing number of messaging servers are added to the back-end system. Due to the negligible amount of state vectors in this simulation, it is suitable to only add message state servers instead of a mixture of both storage services. At the sixth messaging server added, 28.1% of total processor time is recovered from overhead over the first generation Aurora1 system. In terms of perceived performance gain, e.g.

wallclock runtime reduction, Fig. 9 shows this more clearly. The Aurora2 back-end system with a single work unit server and message server is 49.6% faster than its previous generation counterpart. The final test which includes six message state servers, a speedup of 3.65x over Aurora1 is achieved in this low concurrency scenario.

B. Hybrid Shock Simulation

This simulation models shockwave propagation using electromagnetic hybrid algorithms with fluid electrons and kinetic ions [6]. The simulation space is partitioned into cells containing an initial amount of ions. These cells are then aggregated into work units which can be leased to clients. To test the effect of multi-threading in the Aurora2 system the back-end was configured with one work unit state server and message state server along with the proxy service. The threads were then varied for the test case with the following parameters for the hybrid shock simulation: 4000 cells with 100 initial ions per cell, 0.11 lookahead, and a cell width of 0.00025. The simulation was partitioned into 200 work units.

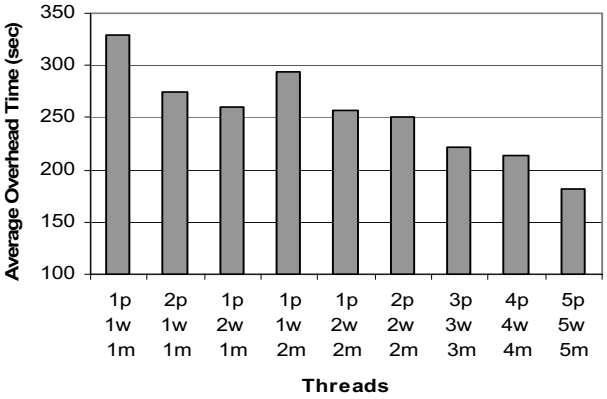

Fig. 10 Effect of Multi-threading on Overhead

Fig. 10 illustrates the trend of decreasing average total overhead time per client across 64 clients (4 *Pentium-III* nodes, 13 *Xeon* nodes) for this high overhead, low concurrency simulation. The ability of the Aurora2 back-end to process requests concurrently is advantageous in reducing average overhead time per client particularly in the areas of work unit import and work unit finalization.

The final test case involves a large-scale Hybrid Shock model that contains both large amounts of state vector data and large amounts of messages being shifted each work unit cycle. The configuration for this Hybrid Shock model increase the number of cells to 10,000 and 800 initial ions per cell. The other parameters remain the same. This simulation was run over 64 clients on 32 *Xeon* nodes.

Steady performance improvement is shown in Fig. 11. Although adding just message state servers produces great initial gains, the performance levels off after the third message state server if the work unit state server is restricted to only one instance. The addition of similar numbers of work unit and message state servers allows for the best scaling improved performance. Application CPU time improves from 14.6% to 49.3% in the *3w4m* case.

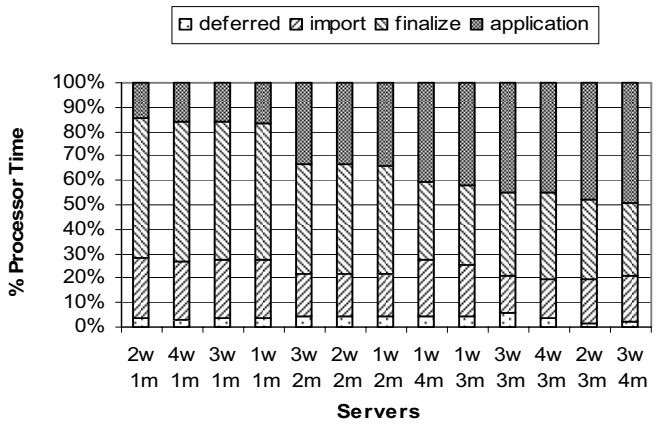

Fig. 11 Hybrid Shock Multi-server Performance

Performance is slightly degraded when adding more work unit servers initially over message state servers. This is due to the phenomena as work unit state server request latency drops, the request rate into the message state server increases reducing the responsiveness of that service further. This implies that the initial bottleneck is not the state server but the message state service for this particular simulation application due to large amounts of processor time dedicated to message binning, and further proven that the addition of one message state server improves performance more than two, three, or four work unit servers.

VI. Related Work

Federating HLA-compliant simulations across a grid using grid-based technologies such as Globus [7] have been explored in projects such as IDSim [8] and HLAGrid [9]. The Aurora system has a less restrictive execution environment and requirements on clients than these systems. In addition, the clients are uncoupled entities and no communication between clients is assumed. PIRS [10] is an effort to speed up parallel simulation through replication on machines with varying workloads. The Aurora system is fundamentally different than these efforts in that the framework allows PDES programs to be run on non-dedicated machines without guarantees that the client may ever return a result.

There has been recent literature describing efforts in harnessing computing power through global computing and desktop grids. Examples include XtremWeb [11], Unicorn [12], InteGrade [13], Harmony [14], DIRAC [15], and BOINC [16]. Several distributed computing systems such as Condor [17], Parallel Virtual Machine (PVM) [18], and Condor MW [19] are systems that pool processing power from various machines into a single computing resource. These systems schedule jobs on machines when they are not in use. The Aurora system utilizes a similar style of work distribution as employed by Condor MW but for computationally intensive message passing data-parallel PDES codes rather than task-parallel applications [20]. The principal difference between these existing distributed computing infrastructures is that the Aurora system is specifically tailored for PDES execution, offering services such as time and LP management.

VII. Future Work

The Aurora2 system has potential for greater performance gains in a variety of areas, some of which are under active development. Scalability can be improved further by reducing the time spent in overhead. One of the main approaches to this issue is allowing client-side caching of states and messages. This will not only reduce serialization, de-serialization, and transmission times but will also reduce bandwidth and memory requirements of the back-end services. Optimistic synchronization has the potential to unlock large performance gains by allowing work units to run past their safe processing bounds. Work units with invalid results can be simply discarded and the time management and LP managers can roll back to a previously known good work unit state.

Another area for performance improvement is load balancing with regard to work unit states and message state queues. Balancing work load is a non-trivial task. If a simulation package is uploaded while other simulations are currently in progress, evaluating the activity of each server prior to work unit distribution could provide balanced load between servers. Dynamic load balancing techniques could also be used to distribute not only processing load but memory load as simulations are active. This technique can prove valuable in reducing latency of the storage services and improving overall responsiveness of the back-end services.

Other areas of development include incorporating first generation Aurora fault tolerance in Aurora2, exploring more advanced and robust communications middleware, the addition of group communications support, and an HLA interoperability layer to provide a bridge to the many parallel and distributed simulations using that interface.

VIII. Conclusions

The Aurora2 architecture has made significant strides in improving performance for large-scale PDES applications on a master/worker paradigm. The distributed back-end system with a multi-threaded proxy, work unit state, and message state services can reduce overhead time and improve request latency. The ability to instance multiple storage services across many machines enhances the scalability of the Aurora2 back-end system. The improvements to the client by separating auxiliary Aurora-specific tasks and the computation thread reduce overhead times.

The first generation Aurora system provided a unique approach to PDES execution through a web services communication middleware. However, this prototype system suffered shortcomings in providing services for large-scale PDES programs. The Aurora2 system offers a more robust application independent framework that builds upon the principles of the first generation Aurora system delivering scalable higher performance. This improved architecture allows large scale PDES applications to take advantage of the master/worker style of parallel workload distribution and execution across desktop grids and public resource computing infrastructures.

References

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: an experiment in public-resource computing," *Commun. ACM,* vol. 45, pp. 56-61, 2002.

[2] D. P. Anderson and G. Fedak, "The Computational and Storage Potential of Volunteer Computing," in *Cluster Computing and the Grid (CCGrid)*, Singapore, 2006, pp. 73-80.

[3] A. Park and R. M. Fujimoto, "Aurora: An Approach to High Throughput Parallel Simulation," in *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*: IEEE Computer Society, 2006.

[4] R. A. van Engelen and K. A. Gallivan, "The gSOAP toolkit for Web services and peer-to-peer computing networks," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Germany, 2002, pp. 117-124.

[5] R. A. van Engelen, "Pushing the SOAP Envelope with Web Services for Scientific Computing," in *Proceedings of the International Conference on Web Services*, Las Vegas, Nevada, 2003.

[6] K. Perumalla, R. Fujimoto, and H. Karimabadi, "Scalable Simulation of Electromagnetic Hybrid Codes," in *International Conference on Computational Science*, University of Reading, UK, 2006, p. 8.

[7] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," in *IFIP International Conference on Network and Parallel Computing*, 2005, pp. 2-13.

[8] J. B. Fitzgibbons, R. M. Fujimoto, D. Fellig, S. D. Kleban, and A. J. Scholand, "IDSim: an extensible framework for Interoperable Distributed Simulation," in *Proceedings of the IEEE International Conference on Web Services*, San Diego, CA, 2004, pp. 532-539.

[9] Y. Xie, Y. M. Teo, W. Cai, and S. J. Turner, "Servicing Provisioning for HLA-Based Distributed Simulation on the Grid," in *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation* Monterey, CA: IEEE Computer Society, 2005.

[10] Y.-B. Lin, "Parallel independent replicated simulation on a network of workstations," in *Proceedings of the eighth workshop on Parallel and distributed simulation* Edinburgh, Scotland, United Kingdom: ACM Press, 1994.

[11] G. Fedak, C. Germain, V. Neri, and F. Cappello, "XtremWeb: a generic global computing system," in *Proceedings of the first IEEE/ACM International Symposium on Cluster Computing and the Grid*, Brisbane, Qld., 2001, pp. 582-587.

[12] T. M. Ong, T. M. Lim, B. S. Lee, and C. K. Yeo, "Unicorn: voluntary computing over Internet," *SIGOPS Oper. Syst. Rev.,* vol. 36, pp. 36-51, 2002.

[13] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra, "InteGrade: object-oriented Grid middleware leveraging the idle computing power of desktop machines," *Concurr. Comput. : Pract. Exper.,* vol. 16, pp. 449-459, 2004.

[14] V. K. Naik, S. Sivasubramanian, D. Bantz, and S. Krishnan, "Harmony: a desktop grid for delivering enterprise computations," 2003, pp. 25-33.

[15] A. Tsaregorodtsev, V. Garonne, and I. Stokes-Rees, "DIRAC: a scalable lightweight architecture for high throughput computing," in *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, 2004, pp. 19-25.

[16] D. P. Anderson, "BOINC: a system for public-resource computing and storage," in *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, PA, 2004, pp. 4-10.

[17] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - A hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, CA, 1988, pp. 104-111.

[18] V. S. Sunderam, "PVM: a framework for parallel distributed computing," *Concurrency: Pract. Exper.,* vol. 2, pp. 315-339, 1990.

[19] J.-P. Goux, J. Linderoth, and M. Yoder, "Metacomputing and the Master-Worker Paradigm," Mathematics and Computer Science Division, Argonne National Laboratory ANL/MCS-P792-0200, February 2000 2000.

[20] D. Kondo, M. Taufer, C. L. Brooks, H. Casanova, and A. A. Chien, "Characterizing and evaluating desktop grids: an empirical study," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, 2004, p. 26.