Writing Programs that Run EveryWare on the Computational Grid [*]

Rich Wolski, [†] John Brevik, [‡] Graziano Obertelli, [§] Neil Spring, [¶] and Alan Su [‖]

ABSTRACT

The Computational Grid [12] has been proposed for the implementation of high-performance applications using widely dispersed computational resources. The goal of a Computational Grid is to aggregate ensembles of shared, heterogeneous, and distributed resources (potentially controlled by separate organizations) to provide computational "power" to an application program.

In this paper, we provide a toolkit for the development of globally deployable Grid applications. The toolkit, called EveryWare, enables an application to draw computational power transparently from the Grid. It consists of a portable set of processes and libraries that can be incorporated into an application so that a wide variety of dynamically changing distributed infrastructures and resources can be used together to achieve supercomputer-like performance. We provide our experiences gained while building the EveryWare toolkit prototype and an its use in implementing a large-scale Grid application.

Keywords: Computational Grid, EveryWare, Ramsey Number search, grid infrastructure, ubiquitous computing, distributed supercomputer

I. INTRODUCTION

Increasingly, the high-performance computing community is blending parallel and distributed computing technologies to meet its performance needs. A new architecture, known as *The Computational Grid* [12], has recently been proposed which frames the software infrastructure required to implement high-performance applications using widely dispersed computational resources. The goal of a Computational Grid is to aggregate ensembles of shared, heterogeneous, and distributed resources, potentially controlled by separate organizations, to provide computational "power" to an application program. Applications should be able to draw compute cycles, network bandwidth, and storage capacity seamlessly from the Grid[*] in a way analogous to the way in which household appliances draw electrical power from a power utility.

The framers of the Computational Grid paradigm identify four qualitative criteria for the concept to be realized. According to [12] (page 18), a Computational Grid must deliver *consistent*, *dependable*, *pervasive*, and *inexpensive* cycles to the end user. In this paper, we outline five quantitative requirements which, if met, fulfill the qualitative criteria from [12]. We also describe *EveryWare* — a toolkit for constructing Computational Grid programs — and evaluate quantitatively how well an example EveryWare program fulfills the Computational Grid vision.

Our evaluation is based on five quantitative metrics:

1. **Execution Rate**: measures the sustained computational performance of the entire application. Although not mentioned explicitly as a criterion, the Grid must be able to deliver efficient execution performance which we measure in terms of sustained execution rate.
2. **Adaptivity**: measures the difference between the performance variability exhibited by the underlying resources and the performance variability exhibited by the application itself. If program execution is stable, independent of fluctuations in resource performance (i.e. the program adapts to varying performance conditions successfully) we suggest that the program is able to sustain *consistent* execution.
3. **Robustness**: measures the overall duration of continuous program execution in the presence of resource failures. A program that can continue to execute effectively in the presence of unpredictable resource failure is a *dependable* program.
4. **Ubiquity**: measures the the degree of heterogeneity a program can exploit in terms of the number of different resource types used by the application. If a program can execute using any and all available resources (both software and hardware) it is a *pervasive* program.
5. **Expense**: measures the cost of the resources necessary to implement the infrastructure. This metric maps directly to the the expense criterion described in [12].

[*]Supported by the National Partnership for Advanced Computational Infrastructure (NPACI), NSF grant ASC-9701333, Advanced Research Projects Agency/ITO under contract #N66001-97-C-8531.

[†]University of California, Santa Barbara – rich@cs.ucsb.edu

[‡]Holycross University – brevik@math.holycross.edu

[§]University of Calfornia, San Diego – graziano@cs.ucsd.edu

[¶]University of Washington – nspring@cs.washington.edu

[‖]University of California, San Diego – alsu@cs.ucsd.edu

[*]We will capitalize the word "Grid" when referring to "Computational Grid" throughout this paper.

Therefore, a program that achieves a high *execution rate*, which is able to *adapt* to rapidly changing performance conditions, which is *robust* to resource failures, which can execute *ubiquitously* and which requires little added *expense* over a single-machine program possess all of the qualities described in [12] that a Grid program must possess.

EveryWare is a software toolkit consisting of of three separate components:

- a **portable** *lingua franca* that is designed to allow processes using different infrastructures and operating systems to communicate,
- a set of **performance forecasting libraries** that enable an application to make short-term resource and application performance predictions in near-real time, and
- a **distributed state exchange** service that allows application components to manage and synchronize program state in a dynamic environment.

The goal is to allow a user to write Grid programs that combine the best features of different Grid infrastructures such as Globus [11], Legion [19], Condor [36], or NetSolve [6] as well as the native functionality provided by Java [18], Windows NT [27], and Unix to the performance advantage of the application. EveryWare is implemented as a highly portable set of libraries and processes that can "glue" different locally available infrastructures together so that a program may draw upon these resources seamlessly. If sophisticated systems such as Globus, Legion, or Condor are available, the EveryWare program must be able to use the features provided by those systems effectively. If only basic operating system functionality is present, however, an EveryWare program should be able to extract what ever functionality it can, realizing that these resources may be less effective than those supporting better infrastructure. The ability to execute ubiquitously with respect to all of the resources accessible by the user is key to meeting the pervasiveness criterion. By leveraging the most performance-efficient infrastructure that is present on those resources, an EveryWare application can ensure the best possible execution performance and the greatest degree of robustness possible.

Designed to be quickly and easily portable, EveryWare is intended to be the thinnest middleware layer capable of unifying heterogeneous resources with various software infrastructures to accomplish a computational task. In a Grid environment with several incompatible software infrastructure choices, it has been challenging to build a distributed application running everywhere, until EveryWare.

We have implemented a prototype toolkit to test the efficacy of the EveryWare approach. In an experiment entered as a contestant in the High-Performance Computing Challenge [22] at SC98, we were able to use this prototype to leverage Globus, Legion, Condor, and NetSolve Grid computing infrastructures, the Java language and execution environment, native Windows NT, and native Unix systems *simultaneously* in a single, globally distributed application. The application, a program that searches for Ramsey Number counter-examples, does not perform an exhaustive search, but instead uses search heuristics such as simulated annealing to negotiate the enormous search space. Effectively implementing this approach requires careful dynamic scheduling to avoid substantial communication overheads. Moreover, by focusing on enhancing the interoperability of the resources in our pool, we were able to combine the Tera MTA[37] and the NT Supercluster[30] - two unique and powerful resources - with several more commonly available systems, including parallel supercomputers, PC-based workstations, shared-memory multiprocessors, and Java-enabled desk-top browsers. With non-dedicated access to all resources, under extremely heavy load conditions, the EveryWare application was able to sustain supercomputer performance levels over long periods of time. As such, the Ramsey Number Search application using EveryWare represents an example of a true Grid program - the computational "power" of all resources that were available to the application's user was assessed, managed, and delivered to the application.

In detailing our Computational Grid experiences, this paper makes four important contributions.

- It defines five quantitative metrics that can be used to measure the effectiveness of Grid applications.
- It demonstrates, using these quantitative metrics, the potential power of globally distributed Grid computing.
- It details our experiences using most of the relevant distributed computing technology available to us in the fall of 1998.
- It describes a new programming model and methodology for writing Grid programs.

In the next section we motivate the design of EveryWare in the context of current Computational Grid research. In Section III we detail the functionality of the EveryWare toolkit and describe the programming model it implements. Section IV discusses the Ramsey Number Search application we used in this experiment and in Section V, we detail the performance results we were able to obtain in terms of the four metrics described above. We conclude in Section VI with a description of future research directions.

II. Computing with Computational Grids

The goal of EveryWare is to enable the construction of true Grid programs – ones which draw computational power seamlessly from a dynamically changing resource pool. Since the field is evolving, a single definition of "Computational Grid" has yet to be universally adopted.[†] In this work, we will use the following definition.

[†]In [12], the authors define Computational Grids in terms of a set of criteria that must be met. We address these criteria in our work, but prefer the definition provided herein for the purpose of illustration.

Computational Grid : A heterogenous, shared, and federated collection of computational resources connected by a network that supports interprocess communication.

By "shared" we mean that it is impractical to dedicate all of the resources in a Computational Grid to a single application for an appreciable amount of time. The term "federated" means that each resource is expected to have local administration, local resource allocation policies, and local resource management software. No single overarching resource management policy can be imposed on all resources.

The resources housed at the National Partnership for Advanced Computational Infrastructure (NPACI) and National Computational Science Alliance (NCSA) constitute examples of Computational Grids. At these centers, machines and storage devices of various types are internetworked. Each resource is managed by its own resource manager (e.g. batch scheduler, interactive priority mechanism, etc.) and it is not generally possible to dedicate all resources (and the network links that interconnect them) at either site to a single application. Moreover, it is possible to combine NPACI and NCSA resources together to form a larger Computational Grid, that has the same characteristics. In this larger case, it is not even possible to mandate that a uniform software infrastructure be present at all potentially useful execution sites.

To maximize application performance on a Computational Grid, a program must be *scalable* (able to exploit concurrency for performance), *adaptive*, *robust*, and *ubiquitous*.

Other work has met these requirements to different degrees. AppLeS [4] (*App*lication *Le*vel *S*cheduling) agents have enabled applications to meet these requirements in environments where a single infrastructure is present and the scheduling agent does not experience resource failure. An AppLeS agent dynamically evaluates the performance that all available resources can deliver to its application, and crafts a schedule that maximizes the application's overall execution performance. EveryWare supports this principle but also extends it to wide-area lossy environments in which several infrastructures may be available. Note also that the AppLeS agent is a specialized application component that performs a single application management function: scheduling. EveryWare generalizes this notion to other application management functions in the form of application-specific services. In Section IV, we describe application-specific scheduling, persistent-state management and performance logging for the Ramsey Number search application in EveryWare.

The MPI (Message Passing Interface) [10], and PVM (Parallel Virtual Machine) [17] implementations for networked systems allow distributed clusters of machines to be programmed as a single, "virtual" parallel machine, allowing applications to *scale*. In addition, portable implementations that do not require privileged (super-user) access for installation or execution [20], [17] are available, promoting their *ubiquity*. However, they do not manage resource heterogeneity on behalf the program nor do they expose it to the programmer so that it may be managed explicitly, so are not *adaptive*.

Grid computing systems such as Globus, Legion, Condor, and HPC-Java [21] include support for resource heterogeneity as well, but they are not yet *ubiquitous*. As they gain in popularity, we anticipate these systems to be more widely installed and maintained. However, we note that their level of sophistication makes porting them to new and experimental environments labor intensive.

EveryWare is similar to Globus [11] in that application components communicate via different well-defined protocols to obtain Grid "service." EveryWare extends this "sum of service" approach to provide tools for the Grid programmer to develop application-specific protocols and services so that the application, and not just the underlying infrastructure, can be robust and ubiquitous.

It also supports information hiding and location transparency in the same way object-oriented systems such as Legion [19] and CORBA [31] do. Indeed, it is possible to leverage the salient features from these object-oriented systems via EveryWare where advantageous to the application.

In particular, we were able to build an application-specific process location service using EveryWare that is similar in concept to the functionality provided by JINI [3]. JINI relies on broadcast and multicast facilities, however, making it difficult to use in wide-area environments. Using the EveryWare *Gossip* protocol, we were able to overcome this limitation, although it is possible that JINI could be used to implement part of the *Gossip* infrastructure.

EveryWare complements the functionality provided by Condor [36] by providing a robust messaging layer. Adaptive and robust execution facilities permit Condor to kill and restart EveryWare processes at will. However, in order to provide an automatic and seamless checkpointing facility, Condor only provides a way for tasks to be migrated between machines of the same architecture. EveryWare's Gossip protocol enables a programmer to write an explicit state-saving facility which is both application and platform neutral. In conjunction with Condor's checkpointing facility, this enables EveryWare programs to span Condor pools based on different architectures.

Dynamically schedulable *adaptive* programs that are capable of tolerating resource performance fluctuations have been developed by the Autopilot [33], Prophet [38], Winner [2] and MARS [16] groups. Most of these systems rely on a centralized scheduler for each application, sacrificing *robustness*. If the scheduler fails or becomes disconnected from the rest of the application, the program is disabled. In addition, having a single scheduling agent impedes scalability as communication with the scheduler becomes a performance bottleneck.

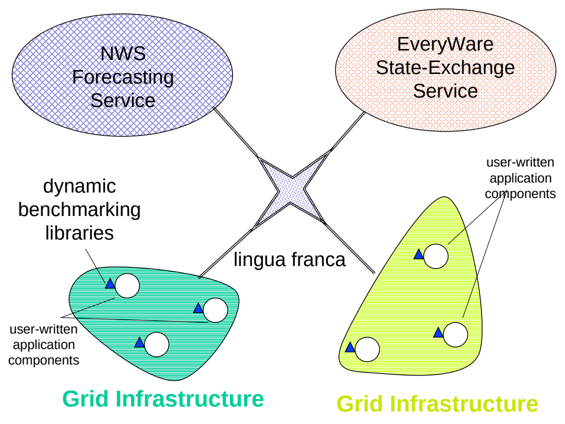

NWS Forecasting Service
EveryWare State-Exchange Service
user-written application components
dynamic benchmarking libraries
lingua franca
user-written application components
Grid Infrastructure
Grid Infrastructure

Fig. 1. **EveryWare Components**

EveryWare is designed as a portable "toolkit" for linking together program components running in different environments. Individual program components may use what ever locally available infrastructure is present. In addition, we provide a low-level "bare-bones" implementation that is designed to use only basic operating system functionality. In this way, an EveryWare application does not assume that any single operating system or infrastructure, except it's own, will be accessible from every resource. Borrowing from the AppLeS [4] project, EveryWare applications characterize all resources in terms of their quantifiable impact on application performance. In this way, heterogeneity is expressed as the difference in deliverable performance to each application. The EveryWare toolkit includes support for process replication and performance forecasting so that an EveryWare application can adapt to dynamically changing resource conditions. We leverage the Network Weather Service [40], [39] forecasting facilities to provide both heterogeneity management and adaptive resource performance prediction.

III. The EveryWare Toolkit

The EveryWare toolkit is composed of three separate software components: a portable *lingua franca* that allows processes using different infrastructures and operating systems to communicate, a set of performance forecasting services and libraries that enable an application to make short-term resource and application performance predictions in near-real time, and a distributed state exchange service that allows application components to manage and synchronize program state in a dynamic environment. Figure 1 depicts the relationship between these components. Application components that are written to use different Grid infrastructure features can communicate amongst themselves, with the EveryWare state exchange service, and with other multi-infrastructure services such as the Network Weather Service [40] using the *lingua franca*. NWS dynamic forecasting libraries (small triangles in the figure) can be loaded with application components directly. These libraries, in conjunction with the performance forecasts provided by the NWS, permit the program to anticipate performance changes and adapt execution accordingly. The distributed state-exchange services provide a mechanism for synchronizing and replicating important program state to ensure robustness and scalability.

The toolkit we have implemented is strictly a prototype designed to expose the relevant programming issues. As such, we do not describe the specific APIs supported by each component (we expect them to change dramatically in our future implementations). Rather, in this section, we motivate and describe the functionality of each EveryWare component and discuss our overall implementation strategy. Our intent is to use the prototype first to implement a variety of applications so that we may determine what functionality is required, and then to provide a "user-friendly" implementation of EveryWare for public release.

A. Lingua Franca

The *lingua franca* provides a base set of resource control abstractions that are portable across infrastructures. They are intended to be simple, easy to implement using different Grid technologies, and highly portable. Initially, we have developed simple *process*, *datagram message*, and *storage buffer* abstractions for EveryWare. The *process* abstraction creates and destroys a single execution thread on a target resource that is capable of communicating via both the EveryWare datagram message abstraction, and any local communication facilities that are present. EveryWare *datagram messages* are sent between processes via non-blocking send and blocking receive calls and processes can block waiting for messages from multiple sources. Processes can also create and destroy *storage buffers* on arbitrary resources (e.g. by creating "memory" processes that respond to read and write requests to their own memory).

We implemented the *lingua franca* using C and TCP/IP sockets. To ensure portability, we tried to limit the implementation to use only the most "vanilla" features of these two technologies. For example, we did not use non-blocking socket I/O nor did we rely upon keep-alive signals to inform the system about end-to-end communication failure. In our experience, the semantics associated with these two useful features are specific to the vendor and, in some cases, to the operating system release level. We tried to avoid controlling the portability of EveryWare through C preprocessor flags whenever possible so that the system could be ported quickly to new architectures and environments. Similarly, we chose not to rely upon XDR [35] for data type conversion for fear that it would not be readily available in all environments. Another important decision was to strictly limit our use of signals. Unix signal semantics are somewhat detailed and we did not want to hinder the portability to non-Unix environments (e.g. Java and Windows NT). More immediately, many of the currently available Grid communication infrastructures such as Legion [19] and Nexus [14] take over the user-level signal mechanisms to facilitate message delivery. Lastly, we avoided the use of threads throughout the architecture as differences in thread semantics and thread implementation quality has been a source of incompatibility in many of our previous Grid computing efforts.

Above the socket level, we implemented rudimentary packet semantics to enable message typing and delineate record boundaries within each stream-oriented TCP communication. Our approach takes its inspiration from the publicly available implementation of `netperf` [23]. However, the actual implementation of the messaging layer comes directly from the current Network Weather Service (NWS) [40], where it has been stress-tested in a variety of Grid computing environments.

Note that the EveryWare *lingua franca* differs from other message passing implementations such as PVM [17] or MPI [20] in several important ways. First, these other interfaces are designed to support arbitrary parallel programs in environments where resource failure is rare (i.e. on parallel machines). As such, they include useful primitives (such as global barrier synchronization) that make them attractive programming facilities, but sometimes difficult to implement in Grid environments. Often, when a resource fails, the entire PVM or MPI program must be restarted. EveryWare assumes that resource availability will be dynamically changing. As such, all primitives obey user-specified time outs, success and failure status is reported explicitly, and only those primitives that can fail individually (that is, without affecting more than the process calling them) are implemented. We do not intend the *lingua franca* to replace any of the existing message-passing or remote invocation systems that are available to Grid programmers. Rather, we will provide the minimal functionality required to allow these infrastructures to interoperate efficiently so that programs can span Grid infrastructures. We expect that the portability of the *lingua franca* will also benefit from this minimalist approach.

B. Forecasting Services

We also borrowed and enhanced the NWS forecasting modules for EveryWare. To make performance forecasts, the NWS applies a set of light-weight time series forecasting methods and dynamically chooses the technique that yields the greatest forecasting accuracy over time [39]. The NWS collects performance measurements from Grid computing resources (CPUs, networks, etc.) and uses these forecasting techniques to predict short-term resource availability. For EveryWare, however, we needed to be able to predict the time required to perform arbitrary but repetitive program events. Our strategy was to manually instrument the various EveryWare components and application modules with timing primitives, and then pass the timing information to the forecasting modules to make predictions. We refer to this process as *dynamic benchmarking* as it uses benchmark techniques (e.g. timed program events) perturbed by ambient load conditions to make performance predictions.

For example, we use the NWS forecasting modules and NWS dynamic benchmarking to predict the response time of each EveryWare state-exchange server. We first identify instances of request-response interactions in the state-server code. At each of these “events” we instrument the code to record an identifier indicating the address where the request is serviced and the message type, and time required to get the corresponding response. By forecasting how quickly a server responds to each type of message, we are able to dynamically adjust the message time-out interval to account for ambient network and CPU load conditions. This dynamic time-out discovery is crucial to overall program stability. Using the alternative of statically determined time-outs, the system frequently misjudges the availability (or lack thereof) of the different EveryWare state-management servers causing needless retries and dynamic reconfigurations.

In general, the NWS forecasting services and NWS dynamic benchmarking allow both the EveryWare toolkit, and the application using it, to dynamically adapt itself to changing load and performance response conditions. We use standard timing mechanisms available on each system to generate time stamps and event timings. However, we anticipate that more sophisticated profiling systems such as Paradyn [28] and Pablo [9] can be incorporated to yield higher-fidelity measurements.

C. Distributed State Exchange Service

To function in the current Grid computing environments, a program must be robust with respect to resource performance failure while at the same time able to leverage a variety of different target architectures. EveryWare provides a distributed state exchange service that can be used in conjunction with application-level checkpointing to ensure robustness. EveryWare state-exchange servers (called *Gossip*s) allow application processes to register for state synchronization by providing a contact address, a unique message type, and a function that allows a *Gossip* to compare the “freshness” of two different messages having the same type. All application components wishing to use *Gossip* service must also export a state-update method for each message type they wish to synchronize.

Once registered, an application component periodically receives a request from a *Gossip* process to send a fresh copy of its current state, identified by message type. Using the previously registered comparator function, the *Gossip* compares the current state with the latest state message received from other application components. When the *Gossip* detects that a particular message is out-of-date, it sends a fresh state update to the application component that originated the out-of-date message.

To allow the system to scale, we rely on three assumptions: first, that the *Gossip* processes cooperate; second, that the number of application components wishing to synchronize is small; finally, that the granularity of synchronization events is relatively coarse. Cooperation between *Gossip* processes is required so that the workload associated with the synchronization protocol may be evenly distributed. *Gossip*s dynamically partition the responsibility for querying and updating application components amongst themselves. For the SC98 experiment, we stationed several *Gossip*s at well-known addresses around the country. When an application component registered, it was assigned a responsible *Gossip* within the pool of available *Gossip*s whose job it was to keep that component synchronized.

In addition, we allow the *Gossip* pool to fluctuate. New *Gossip* processes register themselves with one of the well-known sites and are announced to all other functioning *Gossip*s. Within the *Gossip* pool, we use the NWS clique protocol [40] (a token-passing protocol based on leader-election [15], [1]) to manage network partitioning and *Gossip* failure. The clique protocol allows a clique of processes to dynamically partition itself into subcliques (due to network or host failure) and then merge when conditions permit. The EveryWare *Gossip* pool uses this protocol to reconfigure itself and rebalance the synchronization load dynamically in response to changing conditions.

The assumptions about synchronization count and granularity are more restrictive. Because each *Gossip* does a pair-wise comparison of application component state, N^2 comparisons are required for N application components. Moreover, if the overhead associated with state synchronization cannot be amortized by useful computation, performance will suffer. We believe that the prototype state-exchange protocol can be substantially optimized (or replaced by a more sophisticated mechanism) and that careful engineering can reduce the cost of state synchronization over what we were able to achieve. However, we hasten to acknowledge that not all applications or application classes will be able to use EveryWare effectively for Grid computation. Indeed, it is an interesting and open research question as to whether large-scale, tightly synchronized application implementations will be able to extract performance from Computational Grids, particularly if the Grid resource performance fluctuates as much as we have typically observed [41], [39]. EveryWare does not allow any application to become an effective Grid application. Rather, it facilitates the deployment of Grid-suitable applications, enabling them to ubiquitously draw computational power from a set of fluctuating resources.

Similarly, the consistency model required by the application program dramatically affects its suitability as an EveryWare application, in particular, and as a Grid application in general. The development of a high-performance state replication facilities that implement tight bounds on consistency is an active area of research. EveryWare does not attempt to solve the distributed state consistency problem for all consistency models. Rather, it specifies the inclusion of replication and synchronization facilities as a constituent service. For the application that describe in the next Section (Section IV), we implemented a loosely consistent service based on the *Gossip* protocol. Other, more tightly synchronized services can be incorporated, each with its own performance characteristics. We note, however, that applications having tight consistency constraints are, in general, difficult to distribute while maintaining acceptable performance levels. EveryWare is not intended to change the suitability of these programs with respect to Grid computing, but rather enables their implementation and deployment at what ever performance level they can attain.

D. The EveryWare Programming Model

An EveryWare application is structured as a set of *computational application clients* that request run-time management services from a set of *application-specific servers*. Application clients perform the actual "work" within the application using the features of a native Grid infrastructure. They may themselves be parallel or distributed programs, and they are not constrained to use only the *lingua franca* for communication, process control, and storage management. For operations that require more global control, such as scheduling, user interaction, etc., the computational application clients appeal to application-specific servers, also written by the application programmer. Like the clients, the application-specific servers are not constrained to use any single communication or process control mechanism - they may be written to use any native Grid infrastructure. However, using the *lingua franca* enables arbitrary client and server interaction and ensures portability across infrastructures.

Figure 2 depicts the structure of an application. Application clients (denoted "A" in the figure) can execute in a number of different environments, such as NetSolve, Globus, Legion, Condor, etc. They communicate with application-specific scheduling servers to receive scheduling directives dynamically. Persistent state managers, tuned for the application, control and protect any program state that must survive host or network failure. Application performance logging servers allow arbitrary messages to be logged by the application. Finally, all application components use the EveryWare *Gossip* service to synchronize state. To anticipate load changes, the various application components consult the Network Weather Service (NWS).

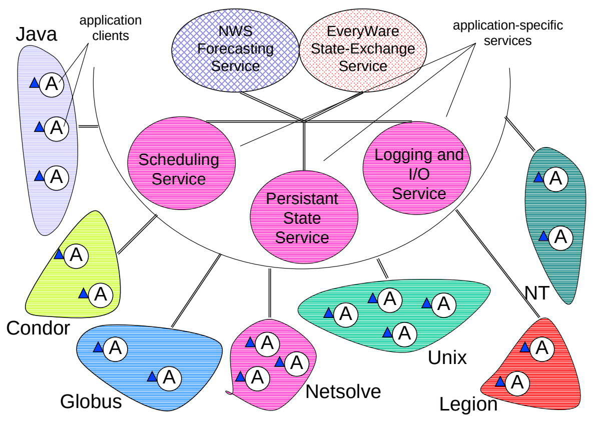


Fig. 2. **EveryWare Application Structure**

This application architecture offers several advantages. First, the overall program can be constructed incrementally. Most concurrent programs are not structured so that some parts may execute while others are being revised, enhanced, or debugged. By structuring an EveryWare program as a communicating set of application-specific services, however, it is possible to interface new pieces of code with the running application. The adaptive nature of the code allows new processes to join and others to drop out while the code continues to execute. Since we do not have to restart the application every time we wish to add a new program component, we can improve and evolve the running application dynamically. Another advantage is that it allows us to implement infrastructure-specific clients that can get the best possible performance by running in "native" mode. Since the clients need only speak the protocol required by each server, we do not need to put a complete software veneer between the computational code and the native infrastructure.

Note that the EveryWare programming model is fundamentally different from that used by most procedure oriented Grid infrastructures such as NetSolve [6], NINF [29], CORBA [31] and NEOS [26]. These infrastructures typically support applications structured as a single controlling client that makes remote-procedure calls to remote computational servers. Under the EveryWare programming model, computation is centered at the clients and program control is coordinated by a set of cooperating application-specific servers. Since the roles or client and server are reversed, we term this application architecture an *inverted client-server* model. This novel application structure offers EveryWare applications greater scalability and robustness than a single-client approach.

IV. Example Application: Ramsey Number Search

The application we chose to implement to test the effectiveness of EveryWare attempts to improve the known bounds of classical Ramsey numbers. The n^{th} *classical* or *symmetric Ramsey number* $R_n = R_{n,n}$ is the smallest number k such that any complete two-colored graph on k vertices must contain a complete one-colored subgraph on n of its vertices. It can be proven in a few minutes that $R_3 = 6$; it is already a non-trivial result that $R_4 = 18$, and the exact values of R_n for $n > 4$ are unknown.

Observe that to show that a certain number j is a lower bound for R_n, one might try to produce a particular two-colored complete graph on $(j-1)$ vertices that has no one-colored complete subgraph on any n of its vertices. We will refer to such a graph as a "counter-example" for the n^{th} Ramsey number. Our goal was to find new lower bounds for Ramsey numbers by finding counter-examples.

This application addresses an unsolved problem in combinatorics using new search techniques. It is not a "classic" scientific application, however, since it does not model real-world phenomena, nor does it provide better applied mathematical or computational techniques for such modeling. However, this application was especially attractive as a first test of EveryWare because of its loose synchronization requirements and its resistance to exhaustive search techniques like those employed in cryptographic factoring [24], [5].

This resistance arises from the combinatorial complexity of the problem. For example, if one wishes to find a new lower bound for R_5, one must search in the space of complete two-colored graphs on 43 vertices, since the known lower bound is currently 43 ([32]). Since such a graph has $\binom{43}{2} = 903$ edges, there are $2^{903} > 10^{270}$ different two-colored graphs on 43 vertices. Even if one could examine 10^{12} configurations every second, an exhaustive search would take over 10^{250} years.

Therefore, we must use heuristic techniques to control the search process. The process of counter-example identification is related to distributed "branch-and-bound" state-space searching.

A. Application Clients

Our goal was to create a dynamically changing population of computational processes executing different heuristics. Heuristic design is an active area of research in combinatorics [32]. As such, we designed the application to be able to incorporate different heuristic algorithms concurrently, each of which implemented as a single application client. The clients would then use the *lingua franca* to communicate with a set of application servers to receive scheduling directives and state management services.

The heuristics that we used all involved *directed search*, by which we mean the following: On the search space of two-colored complete graphs of a particular size, there is a numerical "score" which assigns to each graph the degree to which it fails to be a counter-example in some suitable sense. There is also a set of manipulations called "moves" (transformations) that one can perform on a particular graph to produce other graphs. The algorithm, then, is roughly to start with an arbitrary graph and perform a sequence of moves with a view toward lowering the score by each successive move. Note that in any such heuristic, it is necessary to provide some possibility of making a move that worsens the score; otherwise, there is the danger that the search will get trapped at a local minimum which is not a global minimum.

In our case, the score assigned to a two-colored graph is simply the number of "violations," or complete one-colored subgraphs on n vertices, that it possesses; thus a graph is a counter-example if and only if its score is 0.

Various algorithms employed used slightly different definitions for their moves. The simplest and most common was to change the color of a single edge. Thus, for a graph on 43 vertices possessing 903 edges, there are 903 possible moves that can be made from any given graph. In other algorithms, a move comprised changing the colors of 3 edges. Still other algorithms worked in restricted search spaces which partitioned the edges and only considered those graphs for which all the edges in any given partition were the same color; in such a case a move comprised changing the colors of all the edges within a particular partition.

The two classes of search heuristics employed were those based on *tabu search* [32] and *simulated annealing*. In a tabu search, the algorithm keeps a list (the tabu list) of a fixed length recording the most recent moves that have been made. From a given configuration, it examines all moves not in the tabu list, finds the one that gives the lowest score, and makes and records this move. The tabu list is in place to avoid loops; in practice, some element of randomness is necessary in order to avoid large loops. We employed two variants of the tabu search, namely one that allowed a particular move to be made no more than twice on the list and another that allowed a particular move onto the list if its last appearance was with a different predecessor.

The simulated annealing heuristic mimics the physical behavior of a mass as it undergoes cooling; in this case, the score of a configuration is analogous to the temperature of the mass. Generally, from a given configuration the algorithm chooses a move at random and makes the move if it results in a lower score; otherwise, it rejects the move and chooses another at random from the same configuration. However, the algorithm will accept a random move, regardless of the resulting score, with a small probability that decreases as the score drops; here again, this randomness has the effect of keeping the algorithm from getting trapped in a local minimum.

A.1 Scheduling Service

To schedule the EveryWare Ramsey Number application, we use a collection of cooperating, independent scheduling servers to control application execution dynamically. Each computational client periodically contacts a scheduling server and reports its algorithm type, the IP address of the machine on which it is running, the progress it has made since it last made a scheduling decision, and the amount of time that has elapsed since its last contact. Servers are programmed to issue different control directives based on the type of algorithm the client is executing, how much progress the client has made, and the most recent computational rate of the client.

Scheduling servers are also responsible for migrating work. Clients report the number of violations in the graph they are testing when they check in. If the number is low the server will ask the client for a copy of the graph it is currently considering. If it is high, the server sends the client a better graph and directs it to continue from a different point in the search space. The clients are programmed to randomize their starting point in different ways to prevent the system from dwelling irrevocably in a local minimum. In addition, the thresholds for identifying a "good" graph (one with a low number of violations), a bad one, and the number of times a good one can be migrated to serve as a new starting point in the search space, are tunable parameters.

The schedulers also make decisions based on dynamic performance forecasting information. If a scheduler predicts that a client will be slow based on previous performance, it may choose to migrate that client's current workload to a machine that it predicts will be faster. Rather than basing that prediction solely on the last performance measurement for each client, the scheduler uses the NWS lightweight forecasting facilities to make its predictions. Note that this methodology is inspired by some of our previous work in building application-level schedulers (AppLeS) [34], [4]. AppLeS is an agent-based approach in which each application is fitted with a customized application scheduler that dynamically manages its execution. For the Ramsey Number Search application, however, a single scheduling agent would have been insufficient to control the entire application, both because it would limit the scalability of the application and because the agent would constitute a single-point-of-failure. We designed an application-specific scheduling service that forms organized and robust, but dynamically changing groups of cooperating processes that can make progress if and when the network partitions. As such, we term this type of scheduling Organized Robust AutoNomous Group Scheduling (ORANGS). ORANGS and AppLeS are, indeed, similar in that they use NWS performance forecasts to make application-specific scheduling decisions. However, the distributed and robust nature of the ORANGS service made it a more appropriate choice for the Ramsey Number Search application.

Notice that, for the Ramsey Number search application, the scheduling service considers the use of *all* available resources. When an application client checks in with a scheduling server, the server evaluates the client in terms of the performance it will be able to deliver to the application (using the forecasting services) and decides on the amount and type of work that client should receive. In *all* cases, the Ramsey Number search clients receive some amount of work to perform. For other applications, however, the scheduling service may decide that the use of a particular resource will hinder rather than aid performance and, hence, should be excluded. Therefore, while resource selection is not an issue for Ramsey Number search, the EveryWare programming model supports its implementation.

Schedulers within the scheduling service communicate non-persistent state amongst themselves via the *Gossip* service. In particular, the IP addresses and port numbers of all servers are circulated so that new server instances can be added dynamically. Clients are furnished with a list of active servers when they make contact so that they can contact alternates in the event of a failed server communication. Similarly, scheduling servers learn of different *Gossip* servers, persistent state managers, and logging servers via *Gossip* updates.

A.2 Persistent State Management Service

To improve robustness, we identify three classes of program state within the application:

Local : State that can be lost by the application due to machine or network failure (e.g. local variables within each computational client).

Volatile-but-replicated : State that is passed between processes as a result of *Gossip* updates, but not written to persistent storage (e.g. the up-to-date list of active servers).

Persistent : State that must survive the loss of all active processes in the application (e.g. the largest counter-example that the application finds).

We use a separate persistent state service for three reasons. First, we want to limit the size of the file system footprint left by the application. Many sites restrict the amount of disk storage a guest user may acquire. By separating the persistent storage functionality, we are able to dynamically schedule the application's disk usage according to available capacities. Secondly, we want to ensure that persistent state is ultimately stored in "trusted" environments. For example, we maintained a persistent state server at the San Diego Supercomputer Center because we were assured of reliable storage and regular tape back-ups. Lastly, we are able to implement run-time sanity checks on all persistent state accesses. If a process attempts to store a counter-example, for example, the persistent state manager first checks to make sure the stored object is, indeed, a Ramsey counter-example for the given problem size. This is a significant advantage to application-specific state management.

To implement this functionality, all persistent state objects must be typed. For each persistent type used in the program, the state manager needs a set of sanity-checks (performed when an object is accessed) and a comparator operator so that the state may be synchronized by the *Gossip* service. We acknowledge that developing this functionality for all Grid applications may not be possible. However, we note that many Computational Grid infrastructures currently support mechanisms that can be used to implement the state management functionality we require for Ramsey Number search. For example, the sanity checks performed by the state manager were implemented, primarily, to prevent errant or malicious processes from damaging program state. Instead, Globus authentication mechanisms [13] could be used to provide access control so that only trusted processes may modify persistent state. Similarly, the Legion class management system [25] tracks object instances in a way that could be used to identify stale state. We wanted to ensure that all application components (computational clients and application-specific servers) would be portable to any environment so we did not choose to rest any of the application's functionality on a particular infrastructure. Future versions of the Ramsey Search application may relax this restriction to further benefit from maturing Computational Grid technologies.

A.3 Logging Service

To track the performance of the application dynamically, we implemented a distributed logging service. Scheduling servers base their decisions, in part, on performance information they receive from each computational client. Before the information is discarded, it is forwarded to a logging server so that it can be recorded. Having a separate service, again, allows us to limit and control the storage load generated by the application. For example, NPACI loaned our group a pair of file servers so that we could capture a performance log that spanned the time of the conference.

As with the persistent state managers and the scheduling servers, the logging servers register themselves with the *Gossip* service. Any application process wishing to log performance information learns of a logging server through the server list that is circulated. The logging servers do not register a state synchronization function, however. They use the *Gossip* service only to join the running application.

V. Results

To test the efficacy of our approach, we deployed the Ramsey Number search application on a globally distributed set of resources during SC98. As part of the test, we entered EveryWare in the High-performance Computing Challenge [22] (an annual competition held during the conference) as we believed that the fluctuating loads generated by our competitors would test the capabilities of our system vigorously.

We instrumented each application client to maintain a running count of the computational operations it performs so that we could monitor the performance of Ramsey Number search application. The bulk of the work in each of the heuristics (see Section IV) are integer test and arithmetic instructions. Since each heuristic has an execution profile that depends largely on the point in the search space where it is searching, we were unable to rely on static instruction count estimates. Instead, we inserted counters into each client after every integer test and arithmetic operation. Since the ratio of instrumentation code to computational code is essentially one-to-one (one integer increment for every integer operation) the performance estimates we report are conservative. Moreover, we do not include any instrumentation instructions in the operation counts nor do we count the instructions in the client interface to EveryWare - only “useful” work delivered to the application is counted. Similarly, we include all communication delays incurred by the clients in the elapsed timings. The computational rates we report include all of the overheads imposed by our software architecture and the ambient loading conditions experienced by the program during SC98. That is, all of the results we report in this section are conservative estimates of the sustained performance *delivered* to the application during the experiment.

A. Execution Rate

As a Computational Grid experiment, we wanted to determine if we could obtain high application performance from widely distributed, heavily used, and non-dedicated computational resources. In Figure 3, we show the sustained execution performance of the entire application during the twelve-hour period including and immediately preceding the judging of our High-performance Computing Challenge entry at SC98 on November 12, 1998.[‡]

The *x-axis* shows the time of day, Pacific Standard Time,[§] and the *y-axis* shows the average computational rate over a five-minute time period. The highest rate that

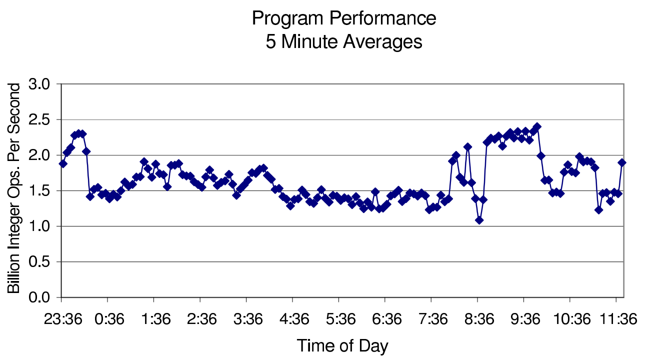


Fig. 3. **Application Speed**

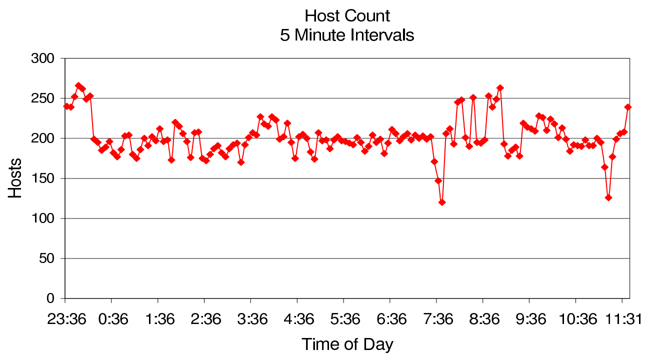


Fig. 4. **Application Host Count**

the application was able to sustain was 2.39 billion integer operations between 9:51 and 9:56 during a test an hour before the competition The judging for the competition itself (which required a “live” demonstration) began at 11:00. As several competing projects were being judged simultaneously, and many of our competitors were using the same resources we were using, the networks interlinking the resources suddenly experienced a sharp load increase. Moreover, many of the competing projects required dedicated access for their demonstration. Since we deliberately did not request dedicated access, our application suddenly lost computational power (as resources were claimed by and dedicated to other applications) and the communication overheads rose (due to increased communication load). The sustained performance dropped to 1.1 billion operations as a result. The application was able to adapt to the performance loss and reorganize itself so that by 11:10 (when the demonstration actually took place), the sustained performance had climbed to 2.0 billion operations per second.

This performance profile clearly demonstrates the potential power of Computational Grid computing. With non-dedicated access, under extremely heavy load conditions, the EveryWare application was able to sustain supercomputer performance levels.

In Figure 4, we show the number of hosts used during the same time period. In this figure, each data point rep-

[‡]We demonstrated the system for a panel of judges between 11:00 AM and 11:30 AM PST.

[§]SC98 was held in Orlando, Florida which is in the Eastern time zone. Our logging and report facilities, primarily located at stable sites on the west coast, used Pacific Standard Time. As such, we report all time-of-day values in PST.

resents the number of hosts checking in during the corresponding five-minute period.[¶] Note that the maximum host count (266) occurs at 23:51 as we ran a large scale test of the system the night before the competition. However, the maximum host count does not correspond to the maximum sustained rate. While we were able to incorporate many new and powerful resources on the morning of the competition, we lost some of the workstations that were loaned to us by Condor during the night. Also, these host count numbers are based on unique IP addresses (and not process id) making them very conservative. Since some systems use the same IP address for all hosts (e.g. the NT Supercluster) the actual host population was much higher. However, we could not distinguish between multiple processes on different hosts with the same IP address, and multiple process restarts due to eviction for the combined host population. As a result, we report the more conservative estimates.

B. Adaptivity

We also wanted to measure the smoothness of the performance response the application was able to obtain from the Computational Grid. For the Grid vision to be implemented, an application must be able to draw "power" uniformly from the Computational Grid as a whole despite fluctuations and variability in the performance of the constituent resources. In Figures 5 and 6 we compare the overall performance response obtained by the application (graph (c) in both figures) with the performance and resource availability provided by each infrastructure. Figure 5 makes this comparison on a linear scale and Figure 6 shows the same data on a log scale so that the wide range of performance variability may be observed. In Figures 5a and 6a we detail the number of cycles we were able to successfully deliver from each Grid infrastructure during the twelve hours leading up to the competition. Similarly, in Figures 5b and 6b, we show the host availability from each infrastructure for the same time period. Together, these graphs show the diversity of the resources we used in the SC98 experiment.

Specifically, Condor supports a dynamic loan-and-reclaim resource usage model. Users agree to loan idle workstations to the Condor system for use by other processes. When a user-specified keyboard activity or load threshold is exceeded, the resource is declared busy and any Condor jobs that are running at the time are evicted. Note that Condor processing power and host count fluctuated through the night and then fell off as the day began in Wisconsin and user activity caused their workstations to be reclaimed. For Java, the performance trajectory was the opposite. We fitted the Java applets with the necessary logging features at approximately 4:30 AM, although we

[¶]The maximum time between check-ins for any computational client was set to five minutes during the test.

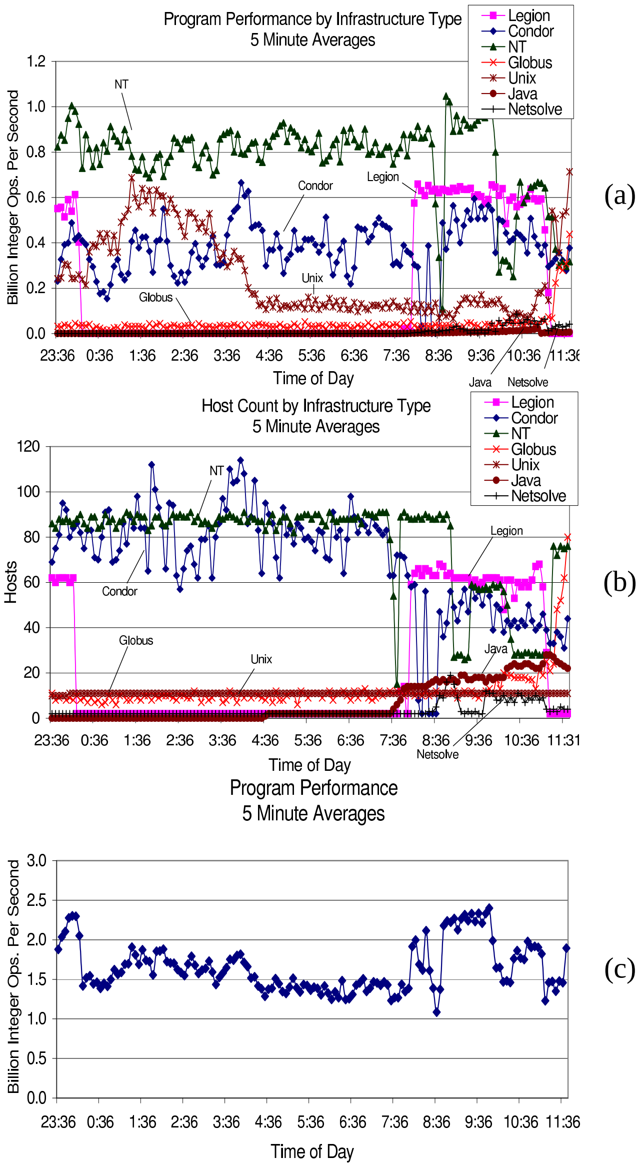
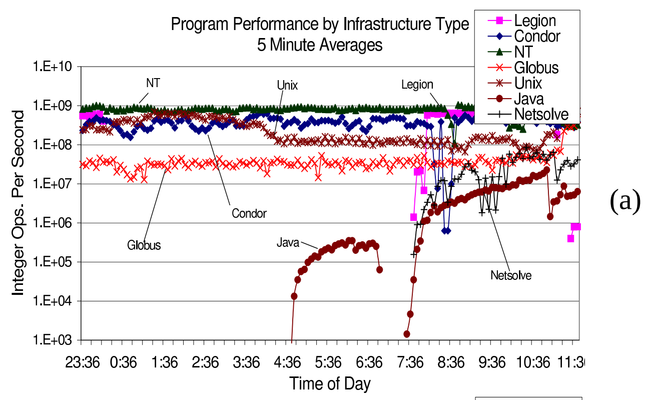
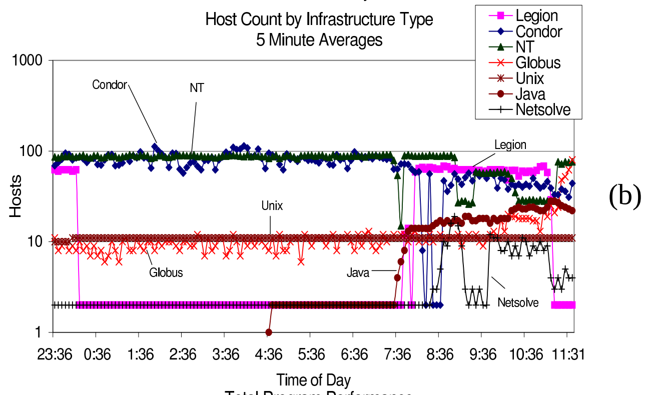


Fig. 5. Execution Rate by Infrastructure (a), Host Count by Infrastructure (b), and Total Sustained Execution Rate (c)

had a small number of test hosts running before then. At approximately 8:00 AM, we announced the availability of the Java implementation and solicited participation from "friendly" sites. In addition, we began to execute the Java applet using HotJava [18] on workstations that had been brought to SC98 for general use by conference attendees. At about the same time, Legion (which had been down since approximately midnight) became available again and the application immediately began to take advantage of the newly available resources. Our Globus utilization, however, was low until just after the competition ended at 11:30 AM, when it suddenly spiked. The Globus group entered the High-performance Computing Challenge with two separate entries. As we did not request dedicated access or special access priority for the demonstration, our application was able to leverage these resources only after higher-priority Globus processes finished. NetSolve





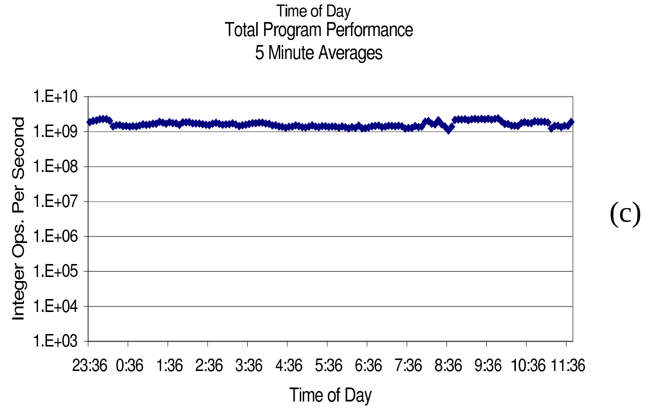


Fig. 6. Log Scale – Sustained Processing Rate (a) and Host Count (b) by Infrastructure, and Total Sustained Rate (c)

gave us access to the student workstation laboratories and several resources in the Innovative Computing Laboratory at the University of Tennessee. We detected a bug in the performance logging portion of the NetSolve implementation at approximately 8:00 AM, hence we have no reliable performance numbers to report for the period before then. The bulk of the NT hosts we were able to leverage came from the Superclusters [30] located at the National Computational Science Alliance (NCSA) and in the the Computer Systems Architecture Group [7] (CSAG) located at the University of California, San Diego. These systems used batch queues to provide space-shared access to their processors. Unix host count remained relatively constant throughout the experiment, but performance jumped at the end as the Tera MTA (the fastest Unix host) was added to the resource pool.

In Figure 5c we reproduce Figure 3 for the purpose of comparison. Figure 6c shows this same data on a log scale.

By comparing graphs (a) and (b) to (c) on each scale we expose the degree to which EveryWare was able to realize the Computational Grid paradigm. **Despite fluctuations in the deliverable performance and host availability provided by each infrastructure, the application itself was able to draw power from the overall resource pool relatively uniformly**. As such, we believe the EveryWare example constitutes the first application to be written that successfully demonstrates the potential of high-performance Computational Grid computing. It is one of the first examples of a truly adaptive Grid program.

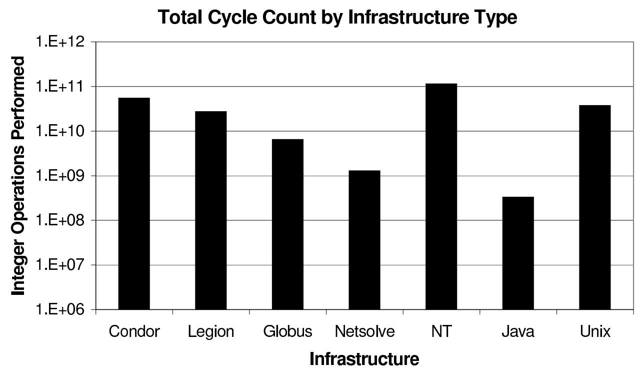


Fig. 7. **Total Cycle Count by Infrastructure**

C. *Aggregate Performance*

Figure 7 shows the total number of integer operations the application was able to obtain during the twelve hours before the competition (on a log scale). With the exception of Java and NetSolve, all infrastructures were within an order of magnitude in terms of the cycles they delivered. Interpreted Java applet performance was typically between 3 and 5 times slower than native binary execution, and the NetSolve computational servers were shared by other NetSolve jobs and student projects.

D. *Robustness*

High-performance computer users often complain about application sensitivity to resource failure in distributed environments. Figure 8 shows the total number of hosts and processes controlled by each infrastructure that were used by the application during the twelve hours leading up to the competition. Comparing the number of processes to hosts gives an indication of the process failure and restart rate during the experiment. Each computational client was programmed to run indefinitely; therefore, in the absence of process failure, the number of processes would equal the number of hosts. We implemented several “ad-hoc” process restart mechanisms for the environments in which they were not automatic. However, most of the process restarts were due either to deliberate termination on our part while debugging, or dynamic resource reclamation by resource owners. On the Condor system, we ran

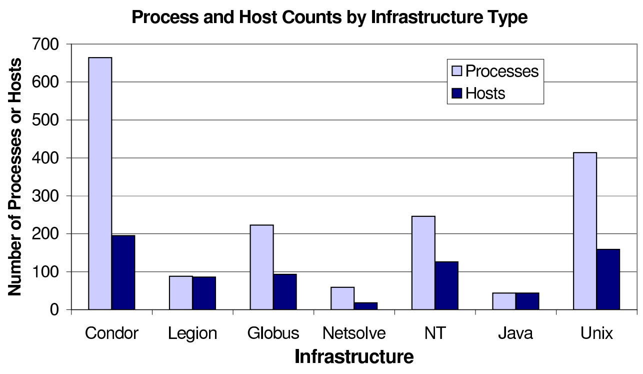


Fig. 8. Total Host and Process Count by Infrastructure

each computational client as a "vanilla" job which is terminated without notice when the resource on which it is running is reclaimed, and subsequently restarted when another suitable resource is free. It is interesting that, despite the midweek daytime usage, process restart due to resource reclamation was relatively infrequent in the Condor environment during the experiment. The Globus comparison illustrates the power of the GRAM interface [11]. Globus allows all processes to be launched and terminated through a single GRAM request. During the time leading up to the competition, we were improving and debugging our Globus implementation. Having a single control point allowed us to restart large batches of processes easily. Under Legion, the concept of process is not defined. Instead, class "instances" move between blocked and running states (and vice versa) so we simply report the number of instances we used during the demonstration. As a result this level of process restart activity is an estimate. The numbers are accurate for the Globus, Condor, and Unix environments but somewhat ambiguous for the other infrastructures. Despite the level of process failure we were able to detect, we were able to obtain the sustained processing rates shown in Figure 3 during the same time period.

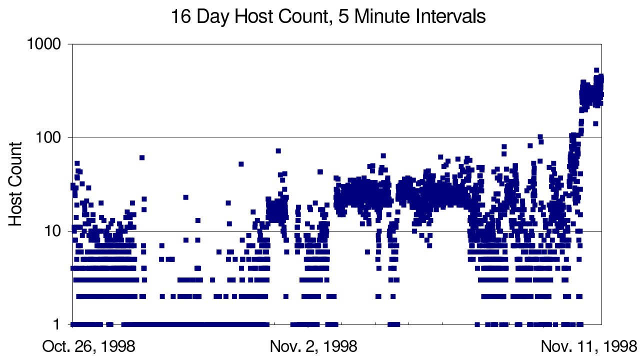


Fig. 9. **Sixteen-day Host Counts**

Indeed, EveryWare and the application design we used proved to be quite robust. In Figure 9 we show host counts over five-minute intervals during the 17 days prior to the judging on November 12. Some portion of the application was executing, more or less continuously during the entire period. As we concentrated our initial efforts on developing the EveryWare toolkit and new Ramsey search heuristics, we did not add performance logging to the running system until October 26. The program had actually been running continuously since early June of 1998; however, we only have performance data dating from the end of October. Note that we were able to add, and then completely revise, the performance logging service while the program was in execution.

E. Ubiquity

For the Computational Grid paradigm to succeed, all useful resources must be accessible by the application. Metaphorically, all profitable methods of power generation must be usable by any power consumer. Figure 10 compares the delivered performance from the fastest host controlled by each infrastructure. The values not only benchmark our code on various architectures, but also show the wide range of resource options we were able to leverage during the experiment. In each case, we attempted to use the native, vendor-specific C compiler (as opposed to GNU `gcc`) with full optimization enabled. On the top half of the figure, we compare the best performance from each infrastructure. The fastest Unix machine was the Tera MTA [37]. We report only the single-processor performance; however, the Tera was also able to parallelize the code automatically and achieve an almost linear speed-up on two processors. The fastest NT-based machine was located at the University of Wisconsin, but we are unable to determine its architectural characteristics. An unknown participant downloaded the NT binary from the EveryWare home page when we announced that the system was operational on Wednesday morning. The fastest Condor machine was a Pentium P6 running Solaris, also located at the University of Wisconsin. Single-processor Pentium P6 performance was particularly good (second only to the Tera) for the integer-oriented search heuristics we developed. The fastest Legion host was a Digital Equipment Corporation Alpha processor running Red Hat Linux, located at the University of Virginia and the fastest Globus machine was an experimental Convex V class host located at the Convex development facility in Richardson, Texas. Surprisingly, the fastest Java execution was faster than the fastest NT, Legion, and Globus machines. An unknown participant at Kansas State University loaded the applet using Microsoft's Internet Explorer on a 300Mhz dual-processor Pentium II machine running NT. We speculate that a student used some form of just-in-time compilation technology to achieve the execution performance depicted in the figure, although we are unable to ascertain how this performance level was reached.

On the bottom half of the figure, we show the best single-processor performance of other interesting and popular machines. The NT Superclusters at UCSD and NCSA

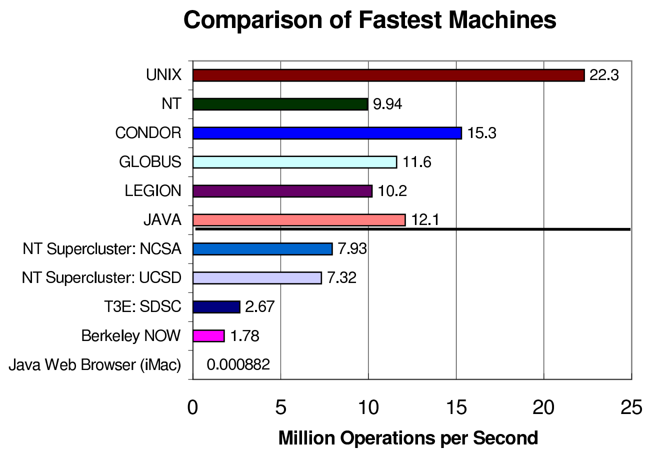


Fig. 10. **Host Speeds**

generated almost identical per-node processing rates. A single node of the Cray T3E located at the San Diego Supercomputer Center was able to run only slightly faster than a single node of the Berkeley NOW [8]. This comparison surprised us since the T3E is space shared (meaning that each process had exclusive access to its processor once it made it through the batch queue) and the NOW (which is timeshared) was heavily loaded. The bottom-most entry shows the speed of a publicly accessible Apple iMac workstation located in a coffee shop on the UCSD campus which is typical of the interpreted Java performance we were able to achieve.

In addition to detailing the relative performance of different architectures and infrastructures, Figure 10 demonstrates the utility of EveryWare. It would not have been possible to include experimental (and powerful) resources such as the Tera MTA and the NT Superclusters without the EveryWare toolkit. At the time of the experiment, none of the existing Grid infrastructures had been ported to either architecture. We were able to port EveryWare to both systems quickly (under 30 minutes for the Tera) allowing us to couple them with other, more conventional hosts that did support some form of Grid infrastructure. By providing execution ubiquity, EveryWare was able to leverage resources that no other Grid computing infrastructure could access. As such, **the Ramsey Number Search application is the first program to couple the Tera MTA, both NT Superclusters, and the Berkeley NOW with parallel supercomputers such as the Cray T3E, workstations, and desktop web browsers**.

VI. Conclusions and Future Work

By leveraging a heterogeneous collection of Grid software and hardware resources, dynamically forecasting future resource performance levels, and employing relatively simple distributed state management techniques, EveryWare has enabled the first application implementation that meets the requirements for Computational Grid computing. In [12], the authors describe qualitative criteria that a Computational Grid must fulfill as the provision of *pervasive*, *dependable*, *consistent*, and *inexpensive* computing.

- **Pervasive**: At SC98, we were able to use EveryWare to execute a globally distributed program on machines ranging from the Tera MTA to a web browser located in a campus coffee shop at UCSD.
- **Dependable**: The Ramsey Number Search application ran continuously from early June, 1998, until the High-Performance Computing Challenge on November 12.
- **Consistent**: During the twelve hours leading up to the competition itself, the application was able to draw uniform compute power from resources with widely varying availability and performance profiles.
- **Inexpensive**: All the resources used by the Ramsey Number Search application were non-dedicated and accessed via a non-privileged user login.

We plan to study how EveryWare can be used to implement other Grid applications as part of our future efforts. In particular, we plan to use it to build Grid versions of a medical imaging code written at the University of Tennessee, and a data mining application from the University of Torino. We also plan to extend ORANGS to include storage scheduling directives and memory constraints. Finally, we plan to leverage our experience with EveryWare to build new Network Weather Service sensors for different Grid infrastructures.

VII. Acknowledgements

It is impossible to acknowledge and thank adequately all of the people and organizations that helped make the EveryWare demonstration at SC98 a success. As such, we miserably fail in the attempt by expressing our gratitude to the AppLeS group at UCSD for enduring weeks of maniacal behavior. In particular, we thank Fran Berman for her moral support during the effort, and Marcio Faerman, Walfredo Cirne, and Dmitrii Zagorodnov for launching EveryWare on every conceivable public email and Java workstation at SC98 itself. We thank NPACI for supporting our High-performance Challenge entry in every way and, in particular, Mike Gannis for enthusiastically making the NPACI booth at SC98 ground-zero for EveryWare. Rob Pennington at NCSA left no stops unpulled on the NT Supercluster so that we could run and run fast, and Charlie Catlett, once again, made it all happen at “The Alliance.” We inadequately thank Miron Livny (the progenitor of Condor and the University of Wisconsin) for first suggesting and then insisting that EveryWare happen. Henri Casanova, at UCSD, single-handedly ported EveryWare to NetSolve after an off-handed mention of the project was carelessly made by a project member within his range of hearing. Steve Fitzgerald at Cal State Northridge and ISI/USC introduced us to the finer and more subtle pleasures of Globus, as did Greg Lindahl for analogously hedonistic experiences with Legion. Brent Gorda and Ken

Sedgewick at MetaExchange Corporation donated entirely too much time, space, coffee, good will, more coffee, sound advice, and patience to the effort. Allen Downey and the Colby Supercomputer Center provided us with cycles, encouragement, and more encouragement. Cosimo Anglano of Dipartimento di Informatica, Università di Torino provided us with intercontinental capabilities and tremendously spirited support. Lastly, we thank EveryOne who participated anonymously via our web interface and downloads. We may not know who you are, but we know your IP addresses, and we thank you for helping us through them.

Biographies

Rich Wolski is an assistant professor of Computer Science at the University of California, Santa Barbara. His research interests include Computational Grid computing, distributed computing, scheduling, and resource allocation. In addition to the EveryWare project, he leads the Network Weather Service project which focuses on on-line prediction of resource performance, and the G-commerce project studying computational economies for the Grid.

John Brevik completed his Ph.D. in Mathematics in 1996 at the University of California at Berkeley under Robin Hartshorne. His interest in Ramsey numbers arose from a combination of caffeine jitters, a slightly flippant response to a challenge from the lead author to produce a combinatorially challenging mathematics problem, and a gross underestimation of the formidability of the problem.

Graziano Obertelli received his Laurea degree from the Dipartimento di Scienze dell'informazione at the Universitá di Milano. Since joining the Department of Computer Science and Engineering at UCSD, he has been conducting research in distributed and parallel computing. Currently, he is working in the Grid Lab at UCSD. His hobbies include strong coffee and fast bikes.

Neil Spring is a Ph.D. student at the University of Washington. He received his B.S. in Computer Engineering from the University of California, San Diego in 1997, and his M.S. in Computer Science from the University of Washington in 2000. His research interests include congestion control, network performance analysis, distributed operating systems, adaptive scheduling of distributed applications, and operating system support for networking.

Alan Su is a Ph.D. candidate in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests have ranged from database systems to scheduling problems in Computational Grid environments. Currently, he is focusing on improving the performance of Grid applications which exhibit dynamic performance characteristics.

References

[1] H. Abu-Amara and J. Lokre. Election in asynchronous complete networks with intermittent link failures. *IEEE Transactions on Computers*, 43(7):778–788, 1994.

[2] O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo. Scheduling parallel applications in networks of mixed uniprocessor/multiprocessor workstations. In *Proc. ISCA 11th Conference on Parallel and Distributed Computing*, September 1998.

[3] K. Arnold, B. O. Sullivan, R. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

[4] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proc. Supercomputing 1996*, 1996.

[5] The Bovine RC5-64 project – http://distributed.net/rc5/.

[6] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *Intl. Journal of Supercomputer Applications and High Performance Computing*, 1997.

[7] Concurrent systems architecture group – http://www-csag.ucsd.edu/.

[8] D. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel computing on the berkeley NOW. In *9th Joint Symposium on Parallel Processing*, 1997. http://now.CS.Berkeley.EDU/Papers2.

[9] L. DeRose, Y. Zhang, and D. Reed. Svpablo: A multi-language performance analysis system. In *Proc. 10th Intl. Conference on Computer Performance Evaluation*, September 1998.

[10] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report CS-94-230, University of Tennessee, Knoxville, 1994.

[11] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. Journal of Supercomputer Applications*, 1997.

[12] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[13] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. Security architecture for computational grids. In *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[14] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 1996.

[15] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):49–59, Jan 1982.

[16] J. Gehrinf and A. Reinfeld. Mars - a framework for minimizing the job execution time in a metacomputing environment. *Proc. Future general Computer Systems*, 1996.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[18] J. Gosling and H. McGilton. The java language environment white paper, 1996.

[19] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, 1994.

[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[21] T. Haupt, E. Akarsu, G. Fox, and W. Furmanski. Web based metacomputing. Technical Report SCCS-834, Syracuse University Northeast Parallel Architectures Center, 1999. http://www.npac.syr.edu/techreports/html/0800/abs-0834.html.

[22] High-performance computing challenge at SC98 – http://www.supercomp.org/sc98/hpcc/, November 1998.

[23] R. Jones. Netperf: a network performance monitoring tool – http://www.cup.hp.com/netperf/NetperfPage.html.

[24] A. Lenstra and M. Manasse. Factoring by electronic mail. In *Advances in Cryptology – EUROCRYPT '89*, pages 355–371, 1990.

[25] M. J. Lewis and A. S. Grimshaw. Using dynamic configurability to support object-oriented programming languages and systems in legion. Technical Report CS-96-19, University of Virginia, 1996.

[26] J. M. M. Ferris, M. Mesnier. Neos and condor: Solving optimization problems over the internet. Technical Report ANL/MCS-P708-0398, Argonne National Laboratory, March 1998. http://www-fp.mcs.anl.gov/otc/Guide/TechReports/index.html.

[27] Microsoft Windows NT. http://www.microsoft.com/ntserver/nts/techdetails/overview/WpGlobal.asp.

[28] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, Nov. 1995.

[29] H. Nakada, H. Takagi, S. Matsuoka, U. Nagashima, M. Sato, and S. Sekiguchi. Utilizing the metaserver architecture in the ninf global computing system. In *High-Performance Computing and Networking '98, LNCS 1401*, pages 607–616, 1998.

[30] NT SuperCluster. http://www.ncsa.uiuc.edu/General/CC/ntcluster.

[31] "OMG". The complete formal/98-07-01: The CORBA/IIOP 2.2 specification, 1998.

[32] S. Radziszowski. Small ramsey numbers. In *Dynamic Survey DS1 – Electronic Journal of Combinatorics*, volume 1, page 28, 1994.

[33] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive control of distributed applications. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, Aug 1998.

[34] N. Spring and R. Wolski. Application level scheduling: Gene sequence library comparison. In *Proc. ACM Intl. Conference on Supercomputing 1998*, July 1998.

[35] Sun Microsystems. XDR: External data representation, 1987. ARPA Working Group Requests for Comment DDN Network Information Center, SRI International, Menlo Park, CA, RFC-1014.

[36] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobbs Journal*, February 1995.

[37] The Tera MTA – http://www.tera.com.

[38] J. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *Concurrency: Practice and Experience*, 1(1), 1998.

[39] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1998. http://www.cs.utk.edu/∼rich/publications/nws-tr.ps.gz.

[40] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5):757–768, 1999. http://www.cs.utk.edu/∼rich/publications/nws-arch.ps.gz.

[41] R. Wolski, N. Spring, and J. Hayes. Predicting the cpu availability of time-shared unix systems on the computational grid. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing*, 1999. http://www.cs.utk.edu/∼rich/publications/nws-cpu.ps.gz.