Running EveryWare on the Computational Grid *

Rich Wolski † John Brevik ‡ Chandra Krintz § Graziano Obertelli ¶
Neil Spring ‖ Alan Su **

November 6, 2001

Abstract

The Computational Grid [10] has recently 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 Grid applications. The toolkit, called EveryWare, enables an application to draw computational power transparently from the Grid. The toolkit 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 the first true Grid application.

1 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* [10], has recently been proposed to frame the software infrastructure required to implement high-performance applications using widely dispersed computational resources. Just as household appliances draw electricity from a power utility, the goal of this new architecture is to enable applications to draw compute cycles, network bandwidth, and storage capacity seamlessly from the Grid.

To realize this vision, the application programming environment must be able to

- leverage *all* potentially useful resources that the user can access,
- exploit the heterogeneity of the resource pool to the program's advantage, and
- manage the effects of dynamically changing resource performance characteristics caused by contention, reconfiguration, and federated administration.

Researchers have developed several innovative and powerful software infrastructures to support the Grid paradigm [9, 16, 35, 4] and several pilot projects [13, 18, 24] have been launched to investigate the efficacy of Grid computing. Each of these technologies, however, is designed assuming that the individual technology is ubiquitous with respect to the resources at hand. We note, however, that the resource pool available to an application is generally specific to its user

* 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 Tennessee – email: rich@cs.utk.edu
‡ University of California, Berkeley – email: brevik@math.berkeley.edu
§ University of California, San Diego – email: ckrintz@cs.ucsd.edu
¶ University of California, San Diego – email: graziano@cs.ucsd.edu
‖ University of Washington – email: nspring@cs.washington.edu
** University of California, San Diego – email: alsu@cs.ucsd.edu

and not a particular infrastructure. A single user may be granted access to resources owned by different organizations which do not agree to support a single, unifying infrastructure. Moreover, the time at at which a given infrastructure may become available on a particular resource is also a problem. There may be a significant time lag between the time a particular machine becomes available, and when these more sophisticated infrastructures can be ported to it.

In this paper, we describe *EveryWare* − a user-level software toolkit for writing Grid programs. EveryWare allows a user to write Grid programs to take maximal advantage of the functionality that is present at each resource to which he or she has access. As such, EveryWare is an embarrassingly portable set of processes and libraries which can "glue" different locally-available infrastructures together so that a program may draw upon these resources seamlessly. In addition, we offer our experiences in the development of the EveryWare prototype and the first true Grid application. The EveryWare toolkit enabled the development of this application and provided a framework in which to study and address the problems associated with writing Computational Grid programs.

In an experiment entered as a contestant in the High-Performance Computing Challenge [17] at SC98 in November of 1998, we were able to use this prototype to leverage Globus [9], Legion [16], Condor [35], NetSolve [4], Java [15], Windows NT [37], and Unix simultaneously in a single, globally distributed application. The application, a program that searches for Ramsey Number counter examples (discussed in Section 3), does not use exhaustive search, but rather requires careful dynamic scheduling for efficiency. Moreover, by focusing on enhancing the interoperability of the resources in our pool, we were able to combine the Tera MTA [36] and the NT Supercluster [25] − two unique and powerful resources − with a raft of other, more commonly available systems including parallel supercomputers, PC-based workstations, shared-memory multiprocessors, and Java-enabled desk-top browsers. As such, it 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. The EveryWare entry was awarded "Best Acceleration" by the panel judging the contest.

The dynamic variability of resource performance in a Computational Grid setting makes writing high-performance Grid programs problematic. EveryWare allows the programmer to consider explicitly the potential variability of resource type (heterogeneity), connectivity, and dynamic load. However, the details associated with managing this variability are separated from the application code by well defined APIs.

In the following section, we describe the components in the EveryWare toolkit. In Section 3, we explain the application we selected as the first Grid program and describe its implementation using the toolkit. In Section 4 we show the performance results and execution characteristics of the Grid program during the SC98 contest. We then detail our experiences with the individual infrastructures in Section 5. We conclude with a future work section in 6 and summary in 7.

2 The EveryWare Toolkit

To realize the performance offered by the Grid computing paradigm, a program must be ubiquitous, adaptive, robust, and scalable. Ubiquity is required because the resources are federated: the owners of the resources allow them to participate in the Grid, but maintain ultimate authority over their use. As such, the resource pool may change without notice as resources are added, removed, replaced, or upgraded. If the program is not compatible with all potentially available software infrastructures, operating systems, and hardware architectures it will not be able to draw some of the "power" that the Grid can provide. Adaptivity is required to ensure performance. If the resource pool is changing, or the performance of the resources are fluctuating due to contention, the program must be able to choose the most profitable resource combination from the resources that are available at any given time. Similarly, if resources become unavailable due to reclamation, excessive load, or failure, the program must be able to make progress. Scalability, in a Grid setting, allows the program to use resources efficiently. The greater degree to which the program can be dispersed, the greater the flexibility the Grid system has in being able to meet its performance needs.

The EveryWare toolkit is composed of three separate software components:

- a **portable *lingua franca*** that is designed to allow processes using different infrastructures and operating systems to communicate,

- a set of **performance forecasting services** that can 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 portability of the *lingua franca* allows the program to run ubiquitously. Dynamic forecasting services permit the program to anticipate performance changes and adapt execution accordingly. The distributed state-exchange services provide a mechanism for synchronizing and and replicating important program state to ensure robustness and scalability.

The toolkit we implemented for SC98 was strictly a prototype designed to expose the relevant programming issues. As such, we do not describe the specific APIs supported by each component (they will 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.

2.1 *Lingua Franca*

For the SC98 experiment, 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 vendor, and in some cases, operating system release-level specific. We tried to avoid controlling portability 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 [33] 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 Window NT). More immediately, many of the currently available Grid communication infrastructures, such as Legion [16] and Nexus [11], 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` [19]. The original implementation of the EveryWare packet layer comes directly from the current Network Weather Service (NWS) [39] implementation, where it has been stress-tested in a variety of Grid computing environments.

2.2 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 (see [38] for a complete description of the NWS forecasting methodology). The NWS collects performance measurements from Grid computing resources (processors, 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 passing 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 applied the forecasting modules to dynamic benchmarks to predict the response time of each EveryWare state-exchange server. We instrumented each server to record the time required to get a response to a request made to each of the other servers, for each message type. To do so, we identified each place in the server code where a request-response pair occurred, and tagged each of these "events" with an identifier consisting of address where the

request was serviced, and the message type of the request. By forecasting how quickly a server would respond to each type of message, we were able to dynamically adjust the message time-out interval to account for ambient network and CPU load conditions. This dynamic time-out discovery proved crucial to overall program stability. Using the alternative of statically determined time-outs, the system frequently misjudged the availability (or lack thereof) of the different EveryWare state-management servers causing needless retries and dynamic reconfigurations (see subsection 2.3 below for a discussion of EveryWare state-exchange functionality). At SC98, network performance on the exhibit floor varied dramatically, particularly as SCINet [29] was reconfigured "on-the-fly" to handle increased demand, thereby exacerbating this problem.

In general, the forecasting services and dynamic benchmarking allow both the EveryWare toolkit, and the application using it, to dynamically adapt itself to changing load and performance response conditions. We trimmed down and adapted the NWS forecasting subsystem so that it may be loaded as a library with application and EveryWare code. We also added a tagging methodology so that arbitrary program events could be identified and benchmarked. We used 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 [23] and Pablo [7] will yield higher-fidelity measurements. While we were unable to leverage these technologies in time for SC98, the prototype EveryWare forecasting interface is compatible with them.

2.3 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. The application component must register 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). The *Gossip* compares that state (using the previously registered comparator function) 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 as a distributed service. Second, that the number of application components wishing to synchronize is small. Lastly, 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. At SC98, the EveryWare *Gossip*s dynamically partitioned the responsibility for querying and updating application components amongst themselves. 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 allowed the *Gossip* pool to fluctuate. New *Gossip* processes registered themselves with one of the well-known sites and were announced to all other functioning *Gossip*s. Within the *Gossip* pool, we used the NWS clique protocol [39] (a token-passing protocol based on leader-election [12, 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 second and third assumptions 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 we implemented for SC98 can be substantially optimized, (or replaced by a more sophisticated mechanism) and 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 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 it did during SC98. EveryWare does not allow any application to become an effective Grid application. Rather, it facilitates the deployment of applications whose characteristics are Grid suitable so that they may draw computational power ubiquitously 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. At SC98, 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.

3 Example Application: Ramsey Number Search

The application we chose to implement as part the EveryWare experiment at SC98 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 higher R_n 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 was especially attractive as a first test of EveryWare because of its synchronization requirements and its resistance to exhaustive search. 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 ([28]). There are $2^{903} > 10^{270}$ different two-colored graphs on 43 vertices which making it infeasible to try all possible colorings. Therefore, we must use heuristic techniques to control the search process making the process of counter-example identification related to distributed "branch-and-bound" state-space searching. Note that this search strategy requires individual processes to communicate and synchronize as they prune the search space implying potentially large communication overheads.

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 it did during SC98. EveryWare does not allow any application to become an effective Grid application. Rather, it facilitates the deployment of applications whose characteristics are Grid suitable so that they may draw computational power ubiquitously from a set of fluctuating resources.

3.1 Implementing the Ramsey Number Search Algorithm with EveryWare

We used the implementation strategy discussed in the previous subsection to structure the Ramsey Number Search application a set of computational *clients* that request run-time management services from a set of application-specific *servers*. Figure 1 depicts the structure of the application. Application clients (denoted "A" in the figure) can execute in a number of different environments, such as NetSolve [4], Globus [9], Legion [16], Condor [35], etc. They communicate amongst themselves and with scheduling servers (marked "S" in the figure) to receive scheduling directives dynamically.

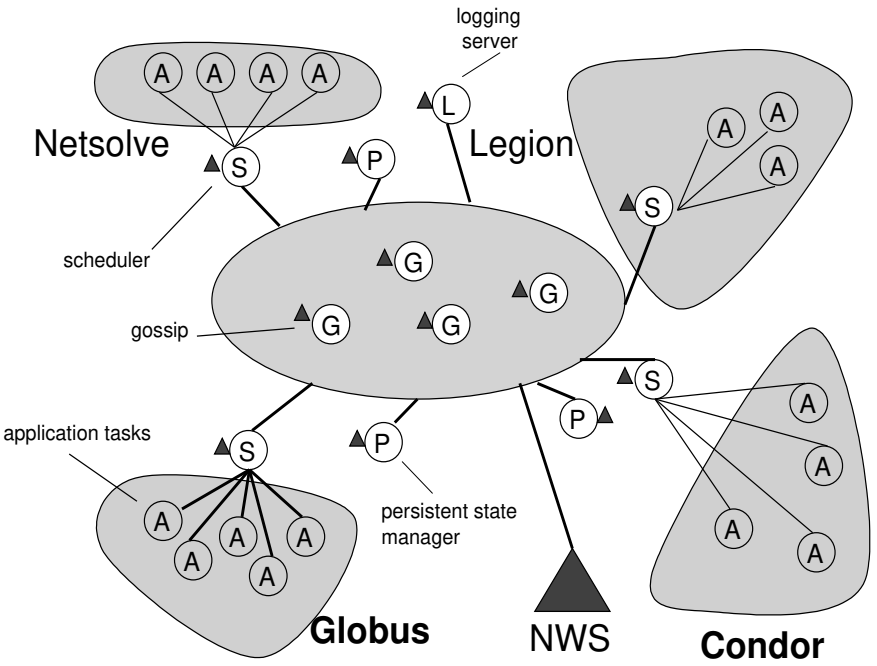


Figure 1: **Structure of Ramsey Number application at SC98**

Persistent state managers (denoted by "P" in the figure) control and protect any program state that must survive host or network failure. Logging servers (marked "L") allow arbitrary messages to be logged by the application. Finally, all application components use the *Gossip* service (marked "G") to synchronize state. To anticipate load changes, the various application components consult the Network Weather Service (NWS) − a distributed dynamic performance forecasting service for Computational Grids [39, 38, 26].

3.1.1 Scheduling

To schedule the Ramsey Number application, we use a collection of cooperating, but independent scheduling servers to control application execution dynamically. Each client periodically reports computational progress to a scheduling server. 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.

The scheduling servers are also responsible for migrating work based on forecasts of available resource performance levels. 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) [31, 3].

3.1.2 Persistent State Management

To improve robustness, we identify three classes of program state within the application: local, volatile-but-replicated, and persistent. Local state is 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 is passed between processes as a result of *Gossip* updates, but it is not written to persistent storage. For example, the up-to-date list of active servers is volatile-but-replicated state. Persistent state must survive the loss of all active processes in the application. The largest counter example that the application has yet to find, for example, is check-pointed as persistent state.

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 regular tape back-ups and industrial quality file system security. 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.

3.1.3 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 allows us to limit and control the storage load generated by the application.

4 Results

To estimate the performance of the Ramsey Number Search application, we instrumented each client to maintain a running count of the computational operations it performs. The bulk of the work in each of the heuristics (see Section 3) 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 SC98.

4.1 Sustained Execution Performance

As a Computational Grid experiment, we wanted to determine if we could obtain high application performance levels from widely distributed, heavily used, and non-dedicated computational resources. In Figure 2, 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 at at SC98 on November 12, 1998[1]. The $x - axis$ shows the time of day, Pacific Standard Time[2], and the $y - axis$ shows the average computational rate over a five-minute time period. The highest rate that the application was able to sustain was 2.39 billion integer operations between 09:51 and

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

[2] 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.

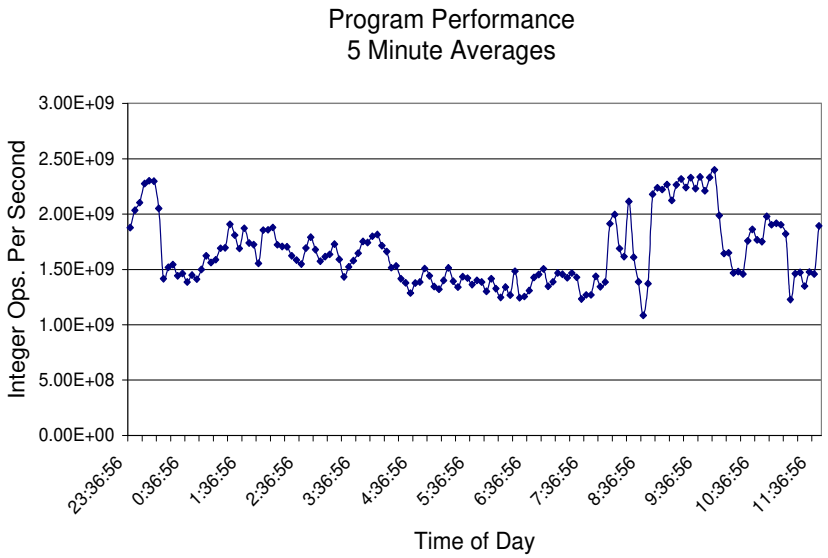


Figure 2: **Sustained Application Performance**

09:56 during a test an hour before the competition (right-hand side of the graph). 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 we claimed by other applications) as 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.

4.2 Performance Response

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 3 and 4 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 3 makes this comparison on a linear scale and Figure 4 shows the same data on a log scale so that the wide range of performance variability may be observed.

In Figures 3a and 4a 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 Figure 3b, we show the host availability from each infrastructure for the same time period. Together, these graphs provide insight into the diversity of the resources we used in the SC98 experiment.

In Figure 3c we reproduce Figure 2 for the purpose of comparison. Figure 4c 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 the first true Grid program.

5 Computational Grid Experiences Using EveryWare

In this section, we describe the way in which we implemented the Ramsey Number Search application using different Grid computing infrastructures with EveryWare as a coordinating tool. Our goal in using these infrastructures was to attempt to leverage the functionality from each that was most useful for the application. In addition, we wished to gain programming experience with these technologies in a "live" Grid setting.

5.1 Unix

We developed the reference implementation of EveryWare and the Ramsey Number Search application for Unix and Unix sockets. Our goal was to target high-performance resources located at the NSF Partnerships for Computational Infrastructure sites. When we began development in the summer of 1998, these resources were entirely Unix based. By starting with a Unix implementation, we believed that we would be able to leverage the greatest number of systems quickly. Our plan was to then use this implementation as the basis for the port to other infrastructure types.

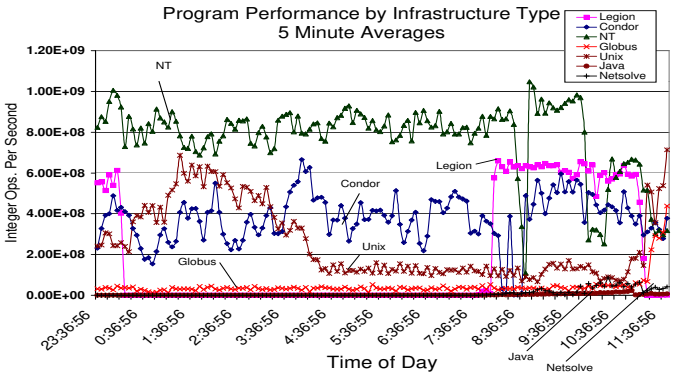
To support this development and deployment strategy, the Unix implementation had to be portable and unparameterized by variables from the execution environment. Portability ensured that we would be able to migrate the functionality, not only between Unix systems, but to other infrastructure types such as Java and NT. Similarly, each application component had to be self-configuring so that we could leverage as many different process invocation mechanisms as possible.

For portability, we identified "basic" Unix functionality that we believed would be commonly supported by most implementations. In particular, we relied upon common semantics for

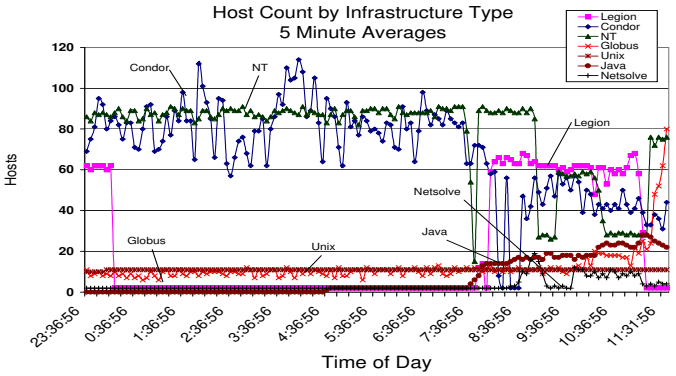
- the socket system calls `send()`,`recv()`,`connect()`, `listen()`,`bind()`, and `accept()`,
- the `select()` file-descriptor synchronization call, and
- the `setitimer()` system call.

In addition, we assumed that there would be some potentially vendor-specific way of obtaining clock readings (e.g. via `gettimeofday()`) with one-second resolution. Conspicuously absent from this list are thread manipulation calls. All of the application-specific services were single threaded which complicated their implementation, but greatly enhanced their portability. Similarly, we chose not to rely upon the `fork()` system call. While the semantics of `fork()` are universal with respect to Unix implementations, we did not believe that all of the execution environments would support `fork()` in the same way. The chief problem we had to overcome was that the socket calls for TCP/IP could block a calling thread of execution indefinitely. On the receiving end, we used the `select()` system call to portably implement a receive with time-out. For sending or connecting, however, we initially relied upon a forked "watchdog" process to send a blocked sender a Unix signal after a time-out. This functionality was difficult to re-implement in non-Unix based environments such as provided by Legion and Java. Instead, we used the self-alarm functionality provided by the `setitimer()` system call which, interestingly, turned out to be relatively portable.

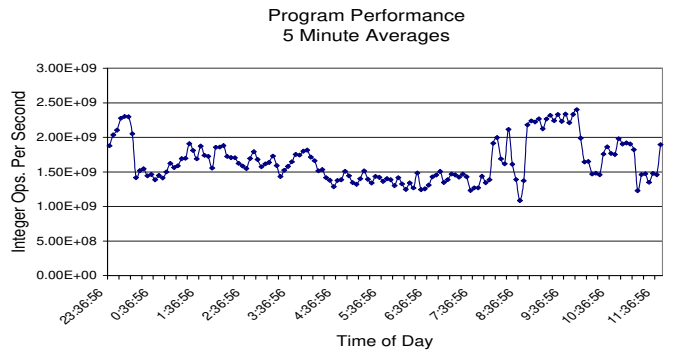
Process invocation semantics presented another impediment to portability and wide deployment. Each infrastructure exported its own interface for launching and terminating processes. The problem was particularly cumbersome in batch



(a)

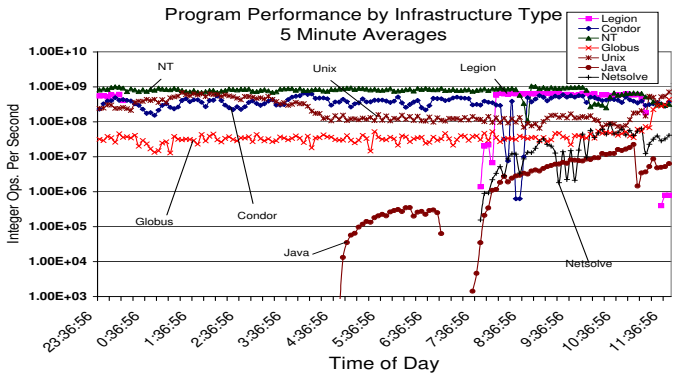


(b)

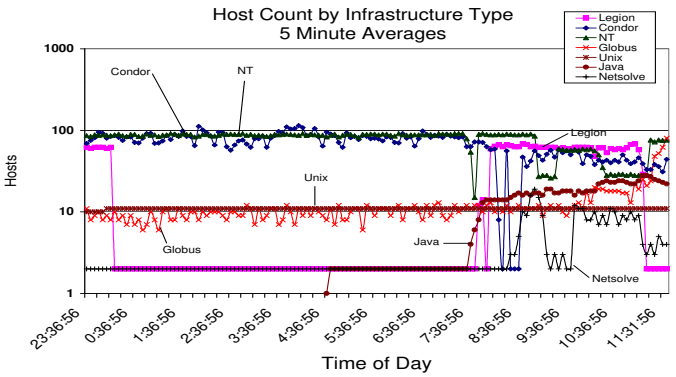


(c)

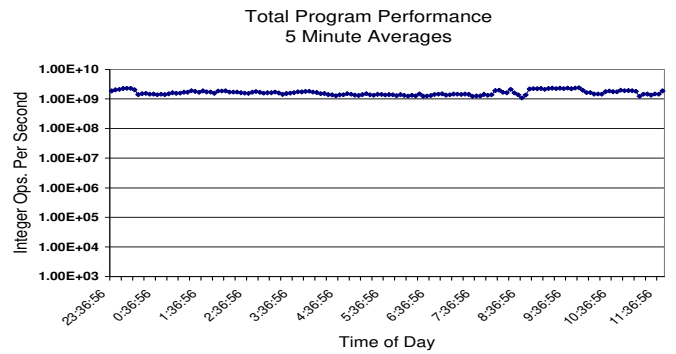
Figure 3: Sustained Processing Rate by Infrastructure (a), Host Count by Infrastructure (b), and Total Sustained Rate (c)



(a)



(b)



(c)

Figure 4: Log Scale: Sustained Processing Rate by Infrastructure (a), Host Count by Infrastructure (b), and Total Sustained Rate (c)

controlled environments where the appropriate invocation semantics were often user and site specific. To get the appropriate level of service from some of these systems, a user must specify the correct account and target queue information. The Globus GRAM [9] interface provided a partial solution (see Section 5.2 for a discussion of our experience using Globus with EveryWare) but, in general, we were unable to use a single set of semantics in all environments. To minimize the problem, all components received any necessary parameters via messages. When any component started, it immediately attempted to contact a scheduler running at a well-known location to receive start-up parameters and execution instructions (see Section 3.1.1 for details). As a result, the application did not need to rely on infrastructure-specific parameter-passing mechanisms or shell environment variables at start-up.

We also used the GNU `autoconf` utility extensively. Header file placement for common library and system call packages is often vendor-specific. While our choice of system calls was relatively universal, using `autoconf` we were able to achieve complete source-code portability across all Unix platforms and between Unix and NT via the CygWin emulation environment (see Section 5.5 for a description of EveryWare for NT).

5.2 Globus

The Globus Project is an on-going research effort to create an infrastructure that allows aggregation of distributed resources into Grids. The Globus Metacomputing Toolkit [9, 14] provides several user-level facilities to build "Globus-enabled" Grid applications.[3]

Each component of the Globus toolkit may be used independently of or in concert with the other services. Figure 5 illustrates the Globus services used by the Ramsey Number Search application. Our principal design goal was to enable *light switch* functionality, which provides the notion of a *single point of control* for activating and deactivating the Globus-enabled application components. The Ramsey Number Search application uses the process control/creation (via the Globus Resource Allocation Manager), persistent storage (via the Global Access to Secondary Storage), and metacomputing directory services from the Globus toolkit. This light switch abstraction hides much of the complexity of the each of these components and the underlying Globus infrastructure.

The Globus Resource Allocation Manager (GRAM) mechanism provides process creation and control capabilities. GRAM is a gatekeeper that first creates certificates of authenticity for each user that enable access to remote compute resources. Once processes are executing GRAM provides the user with means to check job statuses, kill jobs, or read output from jobs. Our Ramsey Number application used the GRAM interface to launch computational clients at various sites running Globus gatekeepers. Once the client was started, output and exit statuses were irrelevant, since the client was not designed to run to completion. Essentially, the GRAM interface was being used as a remote process invocation mechanism in this application.

The Global Access to Secondary Storage (GASS) interface allows applications access to common persistent storage areas. GASS servers essentially allow remote processes (using the GASS client utilities or library functions) to access local file systems. A GASS server acts as a simple file server by binding to a port and transferring files to or from its local file system, driven by requests which are received on that port by remote processes.

We configured a GASS server (running on a well-known host) to act as a repository for pre-compiled computational client binary images for various platforms. Requests sent to gatekeepers would then reference files in the repository, rather than files on the gatekeepers' local file systems. Furthermore, we took advantage of the gatekeepers ability to substitute values for some pre-defined variable names representing the execution platform. This facilitated platform-independent access to the GASS server. By doing so, we used the gatekeeper as a *grappling hook* onto the machine, automatically loading the appropriate binary through GASS.

The Globus Metacomputing Directory Service (MDS) [8] is based on the Lightweight Directory Access Protocol (LDAP) [40]. It serves as a general-purpose repository for information about resources in the Globus testbed. Among other data, the MDS stores information about where each gatekeeper is running, how to contact it (i.e. TCP/IP port number), and how many nodes are free on the resource it manages. The Ramsey Number application used the metadata stored in the MDS to perform crude, but effective, resource discovery. It queried the MDS for a list of potential execution

[3] For clarity, these services are described here as they existed at the time of our experiments. Globus is, by nature, a constantly changing system. Readers interested in changes that may have occurred since SC98 should refer to documentation and technical papers available at [14].

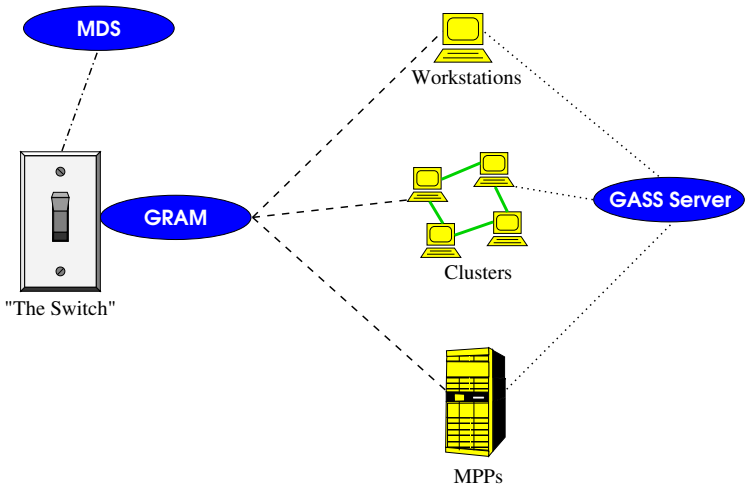


Figure 5: **Ramsey Number application on the Globus infrastructure.** Depicted is the relationship between the Ramsey Number Search application control site, the Grid resources, and the Globus subsystems used. The following Globus mechanisms used by the application and EveryWare toolkit are shown: Globus Resource Allocation Manager (GRAM) the Global Access to Secondary Storage (GASS), and the Metacomputing Directory Service (MDS).

sites and then exercised a relatively lightweight, *authenticate-only* operation to determine if the application's user is authentic and authorized to run remote processes on each gatekeeper that is listed. At the same time, the MDS metadata specified the architecture type of each target machine so that an appropriate GRAM and GASS specification could be made.

5.3 Legion

Legion is an object-based, meta-systems software project at the University of Virginia [16, 22]. It implements a distributed object model that is scalable, easy-to-program, fault-tolerant, and secure. Legion's object-oriented invocation semantics motivated us to develop a "stateless" application client for the Ramsey Number Search application. Legion offers both "stateful" clients in which a specified backing store is used for storage of required information; and "stateless" clients in which operations are performed and results returned independent of prior executions. The advantage from the application's perspective of a stateless client is that Legion implements automatic resource discovery and process migration for stateless objects.

To test our ability to develop application-specific services using EveryWare, we also implemented the scheduling and persistent state services using Legion. Our goal was to permit the Legion environment to be partitioned from the rest of the EveryWare application dynamically in the event of a network connectivity failure. If we had not implemented the Ramsey Number Search services for Legion, we would have had to ensure that another infrastructure was always connected to the pool of Legion resources so that these services could execute.

To communicate with the other infrastructures, we implemented a translator object for the *lingua franca*. As an alternative, we could have loaded each Legion object with the *lingua franca* library, but we found that having a single message translator greatly aided the debugging process. In particular, it gave us a single monitoring point for all messages headed to and from Legion application components. If the translator had become a bottleneck, however, our design would have supported the alternative approach.

To construct the translator, we implemented the Legion versions of the scheduler and persistent state manager as a single object and configured to be passive (to function upon request). The role of the translator was to invoke an appropriate Legion method based on message receipt. In effect, the translator implemented an event model for the

Legion application components that permitted them to respond to events that occurred in other infrastructures.

5.4 Condor

Condor provides reliable guest access to federated workstation resources [35, 5]. The goal of the Condor system is to support high-throughput computing [2] by consuming otherwise idle CPU cycles from a workstation pool. Workstation owners allow Condor to monitor keyboard and process activity to determine when a workstation becomes idle. Idle workstations may be claimed by Condor and used to run guest processes. When workstation activity indicates that the resource is being reclaimed by its owner, the guest process is either checkpointed and migrated to a workstation of the same type, or killed.

For the EveryWare experiment, we used a heterogenous collection of Condor-managed workstations. Since the migration facilities could not move program state between pools of different resource types, we chose to use the "vanilla" Condor universe [5] for the Ramsey Number Search application. In the vanilla universe, guest jobs are terminated without warning when a resource is reclaimed by its owner. Application clients, therefore, checkpointed their persistent and volatile-but-replicated state through the EveryWare *Gossip* mechanisms. Since the EveryWare schedulers are stateless, they were also executed within the Condor pool. When a scheduler was killed by Condor, its clients could automatically switch to another viable scheduler. Scheduler birth and death information was circulated via the *Gossip* protocol so application clients could learn of the currently viable schedulers.

In practice, the overhead associated managing the location transparency of rapidly moving (birthing and dying) schedulers proved prohibitive. Since clients request scheduling service, they could only learn of server death at the time when they attempt to make contact. In this dynamic configuration, clients spent an appreciable amount of time simply locating a viable server. We, therefore, opted for a more stable configuration in which the Condor application clients only contacted schedulers that were located outside of the Condor pools. Since scheduler failure occurred much less frequently than resource reclamation, the overall performance improved.

5.5 NT and Win32

To leverage NT-based systems we used the CygWin [6] compiler, emulation library, and execution environment. The reference Unix implementation used only the most "vanilla" set of system calls possible which were all supported by the emulation library. The overall effort required to complete the port of EveryWare to the Win32 system was minimal, consisting of changes to include files, the inclusion of a less specific random number generation function, and a static definition of where to direct error and log output. By developing a port to NT, we were able to leverage the NT Superclusters located at NCSA and UCSD as well as a variety of PC resources that would have effectively been unusable otherwise.

Execution on the NT Superclusters was straightforward, with the exception of the following minor issues. First, each of the machines in the cluster had to be configured to resolve Domain Name System (DNS) host names. This configuration change was necessary to support communication between computational client processes and the EveryWare schedulers. Since no one expected that the NCSA Supercluster would be used in cooperation with other resources, the ability to resolve host names was not a part of the default configuration. NCSA support personnel quickly resolved this configuration problem for us at SC98.

A more subtle challenge was presented by the batch scheduling system used on the Superclusters, Local Sharing Facility (LSF). To prevent a large set of new worker processes from presenting a excessive instantaneous load to a particular EveryWare scheduler upon startup, we designed each worker process to sleep for a randomized amount of time before soliciting instructions from the rest of the system. LSF seemed to interpret the lack of cpu usage by assuming the process is dead, reclaiming the processor for use by other distributed processes. We reduced the sleep time duration, sacrificing our goal of reduced scheduler load, in order to effectively use the Supercluster processors.

5.6 Java

We implemented a lightweight version of the application in Java in order to take advantage of the ubiquity of Internet-based Java Virtual Machines. This choice was motivated by the desire to allow any user connected to the Internet to contribute processor cycles without downloading and installing any one execution environment or porting the toolkit. Using this framework, a user can download an applet version of the application and immediately participate in the distributed execution.

Java's portability and ease of use comes with considerable tradeoff in performance. The threaded nature of the language however, enabled us to overlap computation with communication. In addition, we implemented a lightweight version of the Ramsey Search heuristics with limited graphics to improve performance. Our results show that the applet version of the Ramsey application is still much slower than when using the other frameworks, but the additional (otherwise unused) cycles still aid computation.

Just-In-Time compilers [20, 34, 32], Java-to-C-code translators [27, 30], and other Java research [21] offer hope for improved performance for future EveryWare applications. For example, during SC98, an interpreted version of the applet on a 300 Mhz Pentium II performed 111,616 integer operations per second on average; a JIT-compiled version performed 12,109,720 integer operations per second on average. Even though the JIT-compiled version is still slower than many of the other hosts from different frameworks in our study, as Java improves in performance, it will be a practical and important gateway to the use of idle cycles.

5.7 NetSolve

The NetSolve [4] infrastructure developed at the University of Tennessee provides brokered remote procedure invocation service in a distributed environment. Computational servers communicate their capabilities to brokering agents. Application clients gain access to remote services through a strongly typed procedural interface.

At SC98, we used NetSolve to test the extensibility of the EveryWare approach. The EveryWare development team[4] had extensive implementation experience with all of the other infrastructures we employed in the study. To test the ability of EveryWare to leverage an infrastructure we had not previously encountered, we appealed to the NetSolve group and asked if they would be willing to develop an EveryWare implementation of the Ramsey Number Search code for NetSolve. Dr. Henri Casanova based his implementation on the Legion version since, at the time of the experiment, the NetSolve invocation interface was functional.

6 Future Work

The EveryWare experiment verified an important conjecture: that programs can be written which realize the Computational Grid paradigm. Our focus in the future will be towards developing EveryWare as a programming tool, and using it to enable Grid application programming.

While the Ramsey Number Search application was an effective test of the EveryWare and Grid computing concepts, it was a difficult program to write. We, the application programmers, had to design and implement both application clients (performing the actual computations) and the application-specific services that were required for robustness and adaptivity. The EveryWare toolkit made such an implementation possible, but not easy. One of our primary future objectives is to develop the software necessary to make EveryWare a useful tool for more than the dedicated and the brave. In particular, we plan to exploit commonalities in the various service designs to provide an application-specific service framework or template. Programmers could then install control modules within the framework that would be automatically invoked by each server.

We plan to study the applicability of EveryWare to a variety of Grid applications. We will continue to enhance Ramsey Number Search application, both as a study of EveryWare and to improve the known bounds of classical Ramsey numbers. Our experience at SC98 showed that to search for R_6, we will need to parallelize some of the individual heuristics, each of which we will implement as a computational client within the application. As a result, we

[4] The members of the EveryWare development team are the authors of this paper.

will develop ways in which EveryWare can be used to couple tightly synchronized parallel codes running on parallel computers with other Grid application components.

We also plan to characterize types of applications that can take advantage of the Grid infrastructure using the EveryWare toolkit. Two characteristics discovered during our SC98 experience that may achieve improved performance using this development environment are applications with coupled master/slave and data parallelism and non-trivial communication and synchronization requirements. To determine the performance levels achievable using the EveryWare toolkit on such applications, we plan to implement an image reconstruction tool called Positron Emission Tomography (PET) and a data mining application called NOW G-Net.

7 Conclusions

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 [10](page 18) the authors describe the 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 high-performance, 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, 1998.
- **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 of the resources used by the Ramsey Number Search application were non-dedicated and accessed via a non-privileged user login.

To our knowledge, EveryWare is the first Grid software effort that has been able to successfully meet these criteria, and to demonstrate the degree to which they are met quantitatively.

8 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 Dimitri Zagorod 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 at 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. Cosimo Anglano of Dipartimento di Informatica, Università di Torino provided us with intercontinental capabilities and tremendously spirited encouragement. 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.

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] J. Basney, M. Livny, and T. Tannenbaum. Deploying a high throughput computing cluster. In R. Buyya, editor, *High Performance Cluster Computing*, volume 1, chapter 5. Prentice Hall, 1999.

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

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

[5] Condor home page – `http://www.cs.wisc.edu/condor/`.

[6] Cygnus. `http://sourceware.cygnus.com/cygwin/`.

[7] L. DeRose, Y. Zhang, and D. Reed. SvPablo: A multi-language performance analysis system. In *Proceedings of 10th International Conference on Computer Performance Evaluation*, September 1998.

[8] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, August 1997.

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

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

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

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

[13] Gusto webpage at `http://www-fp.globus.org/testbeds/`.

[14] Globus. `http://www.globus.org`.

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

[16] 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.

[17] The high-performance computing challenge at sc98 – `http://www.supercomp.org/sc98/hpcc/`.

[18] The NASA information power grid – `http://science.nas.nasa.gov/Pubs/NASnews/97/09/ipg.html`.

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

[20] A. Krall and R. Graft. `http://www.complang.tuwien.ac.at/java/cacao/`.

[21] C. Krintz, B. Calder, H. Lee, and B. Zorn. Overlapping execution with transfer using non-strict execution for mobile programs. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[22] Legion. `http://www.cs.virginia.edu/~mentat/legion/legion.html`.

[23] 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.

[24] 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.

[25] The NT Supercluster at NCSA – `http://www.ncsa.uiuc.edu/General/CC/ntcluster/`.

[26] The network weather service home page – `http://nws.npaci.edu`.

[27] T. Probstein, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson. `http://www.cs.arizona.edu/sumatra/toba/`.

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

[29] SCINet – SC98 Networking `http://www.supercomp.org/history/1998/index.html`.

[30] N. Shaylor. `http://www.geocities.com/CapeCanaveral/Hangar/4040/jcc.html`.

[31] N. Spring and R. Wolski. Application level scheduling: Gene sequence library comparison. In *Proceedings of ACM International Conference on Supercomputing 1998*, July 1998.

[32] Sun Microsystems, Inc. `http://www.sun.com/`.

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

[34] Symantec Corporation. Just-in-time compiler for Windows 95/NT, 1996.

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

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

[37] Microsoft Windows NT Operating System – `http://www.microsoft.com/ntserver/nts/techdetails/overview/WpGlobal.asp%/`.

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

[39] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems (to appear)*, 1999. available from `http://www.cs.utk.edu/~rich/publications/nws-arch.ps.gz`.

[40] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol, March 1995. RFC 1777.