NILE: Wide-Area Computing for High Energy Physics*

Keith Marzullo†, Michael Ogg‡, Aleta Ricciardi‡,
Alessandro Amoroso§, F. Andrew Calkins†, Eric Rothfus‡

1 Introduction

The CLEO project [2], centered at Cornell University, is a large-scale high energy physics project. The goals of the project arise from an esoteric question—why is there apparently so little antimatter in the universe?—and the computational problems that arise in trying to answer this question are quite challenging.

To answer this question, the CESR storage ring at Cornell is used to generate a beam of electrons directed at an equally strong beam of positrons. These two beams meet inside a detector that is embedded in a magnetic field and is equipped with sensors. The collisions of electrons and positrons generate several secondary subatomic particles. Each collision is called an *event* and is sensed by detecting charged particles (via the ionization they produce in a drift chamber) and neutral particles (in the case of photons, via their deposition of energy in a crystal calorimeter), as well as by other specialized detector elements. Most events are ignored, but some are recorded in what is called *raw* data (typically 8Kbytes per event). Offline, a second program called *pass2* computes, for each event, the physical properties of the particles, such as their momenta, masses, and charges. This compute-bound program produces a new set of records describing the events (now typically 20Kbytes per event). Finally, a third program reads these events, and produces a lossily-compressed version of only certain frequently-accessed fields, written in what is called *roar* format (typically 2Kbytes per event).

The physicists analyze this data with programs that are, for the most part, embarrassingly parallel and I/O limited. Such programs typically compute a result based on a projection of a selection of a large number of events, where the result is insensitive to the order in which the events are processed. For example, a program may construct histograms, or compute statistics, or cull the *raw* data for physical inspection. The projection is either the complete *pass2* record or (much more often) the smaller *roar* record, and the selection is done in an ad-hoc manner by the program itself.

Other programs are run as well. For example, a Monte Carlo simulation of the experiment is also run (called *monte carlo*) in order to correct the data for detector acceptance and inefficiencies, as well as testing aspects of the model used to interpret the data. This program is compute bound. Another important example is called *recompress*. Roughly every two years, improvements in detector calibration and reconstruction algorithms make it worthwhile to recompute more accurate *pass2* data (and hence, more accurate *roar* data) from all of the *raw* data. This program is compute-bound (it currently requires 24 200-MIP workstations running flat out for three months) and so must be carefully worked into the schedule so that it does not seriously impact the ongoing operations.

*Supported by NSF National Challenge Grant ASC-9318151
†Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA., USA.
‡Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX., USA.
§Dipartimento di matematica, Università di Bologna, Bologna, Italy

Making this more concrete, the current experiment generates approximately 1 terabyte of event data a year. Only recent *roar* data can be kept on disk; all other data must reside on tape. The data processing demands consume approximately 12,000 SPECint92 cycles a year. Improvements in the performance of CESR and the sensitivity of the detector will cause both of these values to go up by a factor of ten in the next few years, which will correspondingly increase the storage and computational needs by a factor of ten.

The CLEO project prides itself on being able to do big science on a tight budget, and so the programming environment that the CLEO project provides for researchers is innovative but somewhat primitive. Jobs that access the entire data set can take days to complete. To circumvent limited access to tape, the network, or compute resources close to the central disk, physicists often do preliminary selections and projections (called *skims*) to create private disk data sets of events for further local analysis. Limited resources usually exact a high human price for resource and job management and ironically, can sometimes lead to inefficiencies. Given the increase in data storage, data retrieval, and computational needs, it has become clear that the CLEO physicists require a better distributed environment in which to do their work.

Hence, an NSF-funded National Challenge project was started with participants from both high energy physics, distributed computing, and data storage, in order to provide a better environment for the CLEO experiment. The goals of this project, called Nile [7], are:

- to build a scalable environment for storing and processing High Energy Physics data from the CLEO experiment. The environment must scale to allow 100 terabytes or more of data to be addressable, and to be able to use several hundreds of geographically dispersed processors.
- to radically decrease the processing time of computations through parallelism.
- to be practicable. Nile, albeit in a limited form, should be deployed very soon, and evolve to its full form by the end of the project in June 1999.

Finally, the CLEO necessity of building on a budget carries over to Nile. There are some more expensive resources, such as ATM switches and tape silos, that it will be necessary to use. However, as far as possible we are using commodity equipment, and free or inexpensive software whenever possible. For example, one of our principal development platforms is Pentium-based PCs, interconnected with 100 Mbps Ethernet, running Linux and the GNU suite of tools.

2 Non-Problems and Problems

The Nile project resembles other ongoing scientific computational projects. However, details of the CLEO environment will allow Nile to avoid some of the problems that some other scientific computational projects must face. For example,

1. Fragmenting a physicist's analysis program (i.e., splitting it into data-parallel fragments) is simple to do, both for much of the legacy code (mostly in Fortran) and for new code (increasingly in C++). New programs written in C++ will be able to enjoy faster execution by using features of the new data model. We also expect to be able to execute all existing program with, at worst, no significant loss of performance.
2. There will be hundreds to thousands of processors that will be usable for computation, but any physical properties of the processors that one might wish to take into account when assigning (parallelized) programs to processors will be simple. The workstations will all run a standard Posix-compliant operating system (initially Unix) and will differ mainly in easily quantified ways, such as their speed, the amount of memory, which events are stored within its cluster, and so on. Thus, many of the complexities of heterogeneous computing will not occur here.

3. Synchronization will be straightforward. There will be no communication between different analysis programs, and an analysis program will require one barrier synchronization when it completes. The other tasks—event reconstruction (*pass2* and *recompress*) and *monte carlo*—have a well-defined data flow.

Problems that we are concerned with include:

1. Open system architecture. A closed system that performs a set of narrowly-defined tasks will not have the flexibility needed for the constantly evolving environment of CLEO. Besides, high energy physicists are by nature hackers, and they would ignore a closed system. The system must provide a framework that will allow individual physicists to extend the system, change the system, or apply it to some currently-unforeseen task.

2. Data storage, retrieval and transmission. The notion of skims can be captured in a data model. And, having fixed formats (*pass2* output and *roar*) is limiting: it would be very useful to be able to construct caches of other projections than these standard ones.

3. Job monitoring and feedback. Monitoring the progress of very geographically-dispersed programs is a challenge. The physicists will want to be able to look at partial results to determine whether or not to continue processing.

4. Resource allocation. While we believe that the scheduling of programs to processors will be relatively simple, we still expect that load balancing will prove to be useful. Programs will be spread across an internet of workstations, and poor internet interconnectivity, limited data replication and processor crashes will mean that the scheduling must be on-line and ongoing. Furthermore, the internet of workstations will be partitioned into different administrative domains, where each domain may wish to impose its own policies on how its workstations and data storage can be used.

We are addressing these problems as follows.

2.1 Open System Architecture

Nile is being designed as an object-oriented system and being built using CORBA. Doing so will allow us to leverage off a widely-growing set of tools and to allow the system to be extended in a relatively easy manner.

Adopting CORBA has proven to be harder than we first thought it would be. For example, we wish our system to be portable, and since CORBA is a standard, one would think that using CORBA would improve portability. However, each implementation of CORBA (the ORB) has its own idiosyncrasies that hinder portability across ORBs. A second problem is the relationship among processes, C++ objects and CORBA objects. Some of the simple designs we have tried have led to poor performance and even deadlock. We are trying to develop a methodology for making such associations as we develop Nile.

In order to make Nile more open, the architecture includes *policy objects*. Various Nile operations, such as scheduling, fragmentation, and recovery will invoke policy objects. Individuals will be able to supply their own policy objects and change them on the fly. For example, if an individual wishes, she could supply a policy object associated with fragmenting that applied a priority to the individual data runs based on her idea of which events would give her a good first cut. If, in running the application, the generated statistics look less promising than she hoped for, she could cancel the job rather than letting it run to completion.

2.2 Data Storage

We are designing a deep storage system grounded on an automated tape storage system and using our own data servers that keep cached projections of events. The data storage system will provide utilities that allow the users to create caches (much like current skims, but with minimal replication of data), to migrate or replicate of parts of caches for better locality, and to allow annotations to events to be created and stored locally. It will need to be globally manageable as well to be able to accommodate the large storage demands imposed by CLEO.

We are proceeding along two fronts in the design of this service. One approach is based on database systems, such as Illustra. This approach is attractive because it would give us a powerful language for an application to specify which events an application wishes to process. Very little bandwidth would be used in sending events to an application that it would later discard as not being of interest. Annotations are also easily implemented using databases. However, there is little support in database systems for the kind of caching we wish to have. Such caching would have to be done using explicit database queries across large tables read in from tape.

A second approach is to use persistent object bases. This approach is attractive for several reasons. We are building an object-oriented system, and so referring to events (or sequences of events) as objects is a natural design. Caching on disk of some attributes of objects stored on tape is straightforward to do. Events are written once, which makes event objects immutable, and thus amenable to wide area network usage. However, current persistent object bases share some problems of hierarchical data base systems, in that the expressiveness and performance of queries is strongly determined by the schema; that is, the separation of the physical and logical layers is not as clean as in relational databases. Hence, one loses flexibility when using persistent object stores.

We will most likely adopt an architecture that uses both databases and persistent object stores.

2.3 Job Monitoring and Feedback

We believe that the most important property is to have such monitoring and feedback be flexible. Towards this end, we are using a kind of wrapper abstraction to encapsulate the application programs for purposes of monitoring and control [6]. In this approach, each application inherits from a base class that defines a set of methods that expose the state of the application. Some of these methods (called *sensors*) allow the user to monitor the application, such as the accumulated statistics computed by the program, the amount of progress of each parallel component, and the state of the processors running the application. Other methods (called *actuators*) allow the user to modify the application's state, such as forcing a checkpoint, moving a parallel component from one processor to another, or restarting the computation with different event selection parameters.

2.4 Resource Allocation

Each administrative domain (which we call a *site*) will have its own scheduler. This scheduler will collect information about the machines in the site using the sensor mechanism mentioned above. Different sites will be able to use their own schedulers to meet their own local requirements. This service, however, must be highly available, and so must be able to tolerate crash failures. We are using ISIS to maintain the consistency of data replicated within a site.

Scheduling across sites, however, is a much more complex problem. Our experience with the internet has shown that partitioning is a real problem. One year ago, there were severe problems with the handoff between the carriers (such as between Sprint and MCI), but more recently connectivity problems are more likely to occur with local network providers. This situation should improve, but there will still be connectivity problems due to limited bandwidth into an organization. We expect temporary partitions to be a common enough failure model that we need to plan for it.

In the final deployment of NILE, the workstations will consist of tens of administrative clusters that will be interconnected by a high- speed[1] internet. Because of network partitioning, a single global resource manager built as a replicated ISIS process group is not a reasonable design. We are considering two designs for the global resource manager: a flat structure and a hierarchical structure.

Flat Structure Maintaining distributed state in the face of partitioning has been studied for over a decade (for example, [3, 9, 1]). The most recent work has addressed partitioning with group membership protocols that allow subsets of processors to behave consistently within changing partitions. Processes in a partition not only agree on the membership, but on messages they deliver between consecutive overlapping partitions. For example, if two processors believe themselves to be in one partition, m_1, and then both subsequently believe themselves to be in partition m_2, then the two processors will have delivered the same messages while in m_1.

As mentioned above, the resources within a cluster are managed by a replicated process group (called a *Site Manager Group*); the LAN-based replicas of the Site Manager Group will use ISIS and its usual virtual synchrony semantics. In the flat structure, each Site Manager Group will have a *proxy* representing it at each of the other sites. A Site Manager Group and its proxies will form another group, this time using some partitionable group membership semantics. A Site Manager Group will periodically update its proxies with statistical information about the cluster it manages, such as machine loads, assignment of jobs to processors, and current users. A proxy will be able to act on behalf of its site manager (*e.g.*, it can make a decision to allocate resources at its home site to jobs submitted at its host site), but if it finds itself partitioned away from its Site Manager, then its operations will either block (*e.g.*, it cannot start a program in that cluster) or be reversible (*e.g.*, only tentatively dedicating resources).

The main problems with this approach are the communication load needed to keep the proxies up-to-date enough to be useful, and the number of proxy processes at each site. The first problem should not be that severe, since the state being maintained by the proxies could be quite simple, such as the number of machines and the average load over the last 30 seconds. The seond problem is more troubling. It will be desirable to replicate the proxy (using ISIS-style group membership semantics) at each site for purposes of high availability. However, if each proxy replica were to run as its own process, this would result in an unacceptably large number of tightly-coupled processes at each site. We would need to replicate at the object level rather than the process level, and will create such objects upon a limited number of processes in each cluster. Doing so will force us to study the specification of services built using replicated CORBA objects. This in itself is not a problem, since we are using Electra, a CORBA ORB that implements replicated objects through ISIS [4].

Hierarchical Structure The normal approach to making a system scalable is to use a hierarchical strucuture. Hence, we could construct the global manager as a separate service that is updated by the site manager groups. The site manager group would pass a job up to the global manager, which would then make the first cut at partitioning the job into pieces and their associated sites. The pieces would be given to the appropriate site manager groups to do further partitioning and scheduling within its site.

The global manager should be available, but it may not need to be as available as the site managers; a site manager can operate autonomously until contact is re-established. Hence, one could implement the global manager as an ISIS-replicated service at a single site. Or, if there are only a few sites at which most of the computational resources are located, then the global manager could be loosely replicated at these few sites.

The main problem with this approach is that some partitions can cause a collection of sites not to be able to share resources because the global manager group is partitioned away. This could be addressed by adding further levels to the hierarchy.

[1]The original intention was to use the US National Information Infrastructure project, but changes in US policy mean that we have to plan for a long-distance network offering significantly less. In any case, early implementations of NILE will operate purely within one geographical domain.

In order to make progress, we are prototyping a hierarchical solution and will prototype a flat solution using very weak group membership semantics in order to understand what the scaling problems will be in the full NILE system.

3 Further Details

A prototype (*fast-track*) implementation of NILE has been completed, which has allowed us to experiment with architectural details in a homogenous (Digital AXP) environment. Data storage models are being built on top a persistent object manager (AIMnet) and an application-specific data server. Application management is being built on top of a C++ ORB, Electra, and ISIS on both SunOS and Linux. We have also prototyped on top of the ORBeline C++ ORB [8] (on SunOS). We have a working prototype of the sensors and actuators abstraction, and have a CORBA IDL description of the application management. Further details about the NILE project can be found on the NILE home page `http://www.nile.utexas.edu/`.

As well as the authors of this paper, the NILE collaborators include Ken Birman, David Cassel, Dan Riley (Cornell); Michael Athanas, Paul Avery, Prasun Gupta, Karp Jeong, Ted Johnson (Florida); Wei-Cheng Lai and Fabio Previato (Texas). The ideas presented here reflect the work of all these individuals.

References

[1] O. Babaoglu, R. Davoli, L. Giachini and P. Sabattini. The inherent cost of strong-partial view-synchronous communication. *Proceedings of WDAG 9*, Springer-Verlag LNCS 972, 1995, pp. 72–86.

[2] Cleo home page `http://www.lns.cornell.edu/`.

[3] A. El Abbadi and S. Toueg. Availability in partitioned replicated databases. *Proceedings Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1986, pp. 240–251.

[4] Electra home page `http://www.cs.cornell.edu/people/maffeis/`.

[5] Isis home page `http://www.stratus.com/`.

[6] K. Marzullo, R. Cooper, M. Wood and K. P. Birman. Tools for distributed application management. *IEEE Computer* 24(8):42-51, August 1991.

[7] Nile home page `http://www.nile.utexas.edu/`.

[8] ORBeline home page `http://www.visigenic.com/prod/orbpd.html`.

[9] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. *Proceedings of FTCS 23* (Toulouse, France), June 1993, pp. 534–543.