Near-real-time Satellite Image Processing: Metacomputing in CC++ [*]

Craig A. Lee
Computer Systems Division
The Aerospace Corporation

Carl Kesselman
The Beckman Institute
California Institute of Technology

Stephen Schwab
Computer Systems Division
The Aerospace Corporation

Abstract

Metacomputing entails the combination of diverse, heterogeneous elements to provide a seamless, integrated computing service. We describe one such metacomputing application using Compositional C++ that integrates specialized resources, high-speed networks, parallel computers, and VR display technology to process satellite imagery in near-real-time. From the virtual environment, the user can query an `InputHandler` object for the latest available satellite data, select a satellite pass for processing by `CloudDetector` objects on a parallel supercomputer, and have the results rendered by a `VisualizationManager` object which could be a simple workstation, an ImmersaDesk or a CAVE. With an ImmersaDesk or CAVE, the user can navigate over a terrain with elevation and through the cloudscape data as it is being pumped in from the supercomputer. We discuss further issues for the development of metacomputing capabilities with regards to the integration of run-time systems, operating systems, high-speed communication, and display technologies.

Keywords: Metacomputing, parallel and distributed languages, high-speed networks, virtual environments.

1 Introduction

Satellite constellations typically require large, distributed ground systems for processing and dispersing data products. The performance re-

[*] *Appears in IEEE Computer Graphics & Applications, July 1996.*

quirements of these ground systems depend on the number of satellites in the constellation, the bandwidth of the bits "hitting the ground" and the complexity of the application domain processing. In addition, after the bits hit the ground, there can be a hard deadline after which data products must be available. When coupled with end-user demands for data with higher temporal and spatial resolution, ground system performance requirements are increasing dramatically.

To achieve this level of performance efficiently, compute resources must be integrated to a level far beyond what is typically seen today. Unique data sources, processing power, display technologies and networks must be integrated into a seamless resource that can be easily managed within one framework. These are the requirements of *metacomputing*, a uniform method for handling a collection of heterogeneous resources.

In this paper, we present a case study in metacomputing: a cloud detection and visualization application for infrared and visible light satellite images that integrates high-speed networks, parallel and distributed computing, and stereoscopic visualization using Compositional C++ (CC++) [1], a simple yet powerful extension of C++, and its run-time system, Nexus [5]. We begin by discussing the problem of metacomputing in a little more depth.

2 Elements of Metacomputing

Metacomputing adds another dimension to the system building problem. In addition to making sure that an application is implemented correctly, one must be able to do *configuration management* over a potentially arbitrary collection of resources. This means that the application must be able to (1) identify available resources, (2) acquire any such resources, (3) initialize the computation on them, and eventually (4) terminate. Initialization can be further categorized into independent and dependent initialization in a static or dynamic topology. For example, after a compute node is acquired, the application must do initialization that is local to that compute node and independent of other nodes, e.g., find local data files and build internal data structures. In order to integrate that compute node into the computation, the application must do initialization that is dependent on other nodes, e.g., exchange various operating parameters and references to remote objects. This often requires an order to independent and dependent initialization phases as compute resources and application objects are integrated into the computation. For many applications, a static topology of resources is sufficient. As this technology matures, however, application topologies will be increasingly dynamic and may change as the computation proceeds.

Once installed on some configuration of resources, an application must be able to do *resource management* effectively. This means composing not only different machines, but (1) composing different types of parallelism, (2) managing both synchronous and asynchronous control flow among compute nodes, (3) allowing for both control-oriented and data-oriented synchronization, and (4) managing data locality in order to minimize communication and latency. Communication among compute nodes is an important resource that can take many forms in a heterogeneous environment. This could be shared memory, hardware message-passing, or network message-passing such as ethernet, ATM, HIPPI, etc. All of these media must be controllable by the application to manage the available bandwidth and tolerate variable latencies. When combined with unique resources, such as satellite downlink stations and stereoscopic visualization systems, the metacomputer can be a geographically disperse, networked, parallel and distributed, heterogeneous system. Controlling such a system is greatly facilitated by a uniform method whereby one can easily express the desired computational behavior.

3 CC++ and Nexus

Compositional C++ (CC++) is a small, uniform extension to C++ for handling parallel and distributed computation. Since the language is essentially *architecture-independent*, it can be used to handle heterogeneous resources. While CC++ gives the implementor a great deal of flexibility, its run-time system, called Nexus, manages the actual transfer of data and control among the compute nodes on whatever bitways are available almost transparently at the language level. This is an effective combination for managing a large heterogeneous system. We briefly describe CC++ prior to describing Nexus.

The CC++ extension consists of only six new keywords and their attendant semantics: `global, par, parfor, spawn, atomic` and `sync`.

The keyword `global` is essentially used to denote and control *locality* – the minimal necessary concept that must appear at the language level to support metacomputing. Applying `global` to a class denotes a *processor object*; an object that has its own address space, is associated with some (at least logical) location and, for practical purposes, an executable binary. Applying `global` to a pointer denotes a pointer that can be used to reference any data type regardless of location. Hence, a global pointer can be used to access data and function members of a remote object. For example,

```
local_var = gp->remote_data_member;
```

assigns the value of a remote data member referenced by the global pointer `gp` to a local variable. The statement

```
ret_val = gp->remote_member_func(args);
```

locally evaluates `args`, transfers them to the remote location where the remote member function is evaluated, and eventually assigns the return value to the local variable.

The other five keywords are used to control parallelism and synchronization. Structured and unstructured control parallelism is specified using `par, parfor`, and `spawn`. A `par` block denotes parallel execution of a fixed number of statements known at compile-time. A `parfor` statement denotes parallel execution over an index space whose range does not need to be known until run-time. Both of these statements offer structured parallelism since each statement terminates when all of the constituent statements or loops have terminated. The `spawn` statement allows unstructured parallelism since any `spawn`ed statement executes in parallel without necessarily any subsequent synchronization.

Both control- and data-oriented synchronization are supported. Applying `atomic` to a member function denotes that it can be executed by only one thread of control at a time. Threads attempting to enter an "occupied" atomic function block until the occupying thread leaves. Applying `sync` to a data member denotes *single-assignment semantics*; threads referencing this data member block if the member has not yet been assigned a value.

All of these language level semantics are supported at the run-time level by Nexus. Nexus associates a processor object name at the language level with an executable binary. Processor objects are created using the standard `new` syntax along with the *optional placement syntax.* In C++, the optional placement syntax can be used to specify memory alignment. In CC++, it can also be used to specify *location*, i.e., a host name or number where Nexus should execute the binary file that corresponds to the new processor object. Nexus currently uses a relatively static *resource database* to determine where the desired host actually is and how to bootstrap a new processor object, or in Nexus terminology, a *context* on it. If the target host is in the same trusted domain and on the same filesystem, Nexus can use a simple `rsh` command to start the executable binary. If this is not the case, an `.rhosts` file can be used in conjunction with the `rsh`. To achieve better security, a *nexus server* can be used that listens on a specified network port number and uses PGP authentication to verify requests to create new contexts (start executables) on that host. Once the child context is established, the parent and child will communicate using the method specified in the database, e.g., shared-memory, or local message-passing hardware or bitways such as ethernet, ATM, etc.

Global pointer operations between objects are translated into *remote service requests* (RSRs) between contexts. At creation time, an object registers a set of *entry functions* with its local Nexus that are used to service remote requests for retrieving the value of data members or executing function members locally. When an object initiates a global pointer operation, the arguments are evaluated locally and then marshaled by the Nexus sending the RSR. The receiving Nexus looks-up and executes the appropriate entry function. If there is a return value from the member function, the CC++ compiler generates code for a return RSR to the originating Nexus and object.

If the size and structure of an argument type for a remote member function is known at compile-time, then the compiler can generate the appropriate marshaling code for the RSR. If the argument's size is dynamic and not known until run-time, or if the argument type involves pointers, then the CC++ program must supply a *void shift operator* to tell Nexus how to move data from one context to another. (CC++ uses the concept of shifting data *into the void* from one context and *out of the void* into another context.) When supplied with the properly typed shift operators, the CC++ compiler can direct Nexus to transparently manage the transfer of arbitrary data types between contexts.

For the parallel and synchronization language features, Nexus provides a threaded environment. Parallel threads of execution are created for `par, parfor`, and `spawn` statements. Condition and mutex variables are used to implement the synchronization semantics of `atomic` and `sync`. The semantics implemented by Nexus threads are fully integrated with the global pointer and RSR machinery such that any number of threads can interact with fair scheduling among any number of contexts.

4 An Application Case Study

Having described the functionality that CC++ offers for heterogeneous parallel and distributed computing, this section describes the use of that functionality in building a large-scale application. We begin by briefly illustrating what the application's *problem architecture* looks like and our implementation architecture.

4.1 The Application

NEPH is a near-real-time cloud detection code for satellite imagery. (Nephology is the study of clouds.) The main input to the code is two-dimensional infrared and visible light images from on-board satellite sensors. The image pixels are first geo-located (assigned a latitude and longitude value). Infrared data is corrected for previously known temperature climatology depending on geography type (land, water, desert, etc.), time of day and time of year. Visible data is corrected for previously known background brightness according to geography type, time of day and time of year. The corrected data and other historical data is used to determine dual thresholds in both channels (visible and infrared) to decide if a pixel is clear, partially cloudy or completely cloudy. This leads to a 3×3 decision matrix for pixel cloudiness. False colors can be used to indicate which matrix element a pixel falls in except the "clear" pixel which is colored according to geography type. (See Figure 1.) In an operational system that is processing data on a regular basis, the results of each computation are stored and used, in part, to determine the thresholds used on future data sets.

Previous implementations of this system use a *sub-analysis box*,

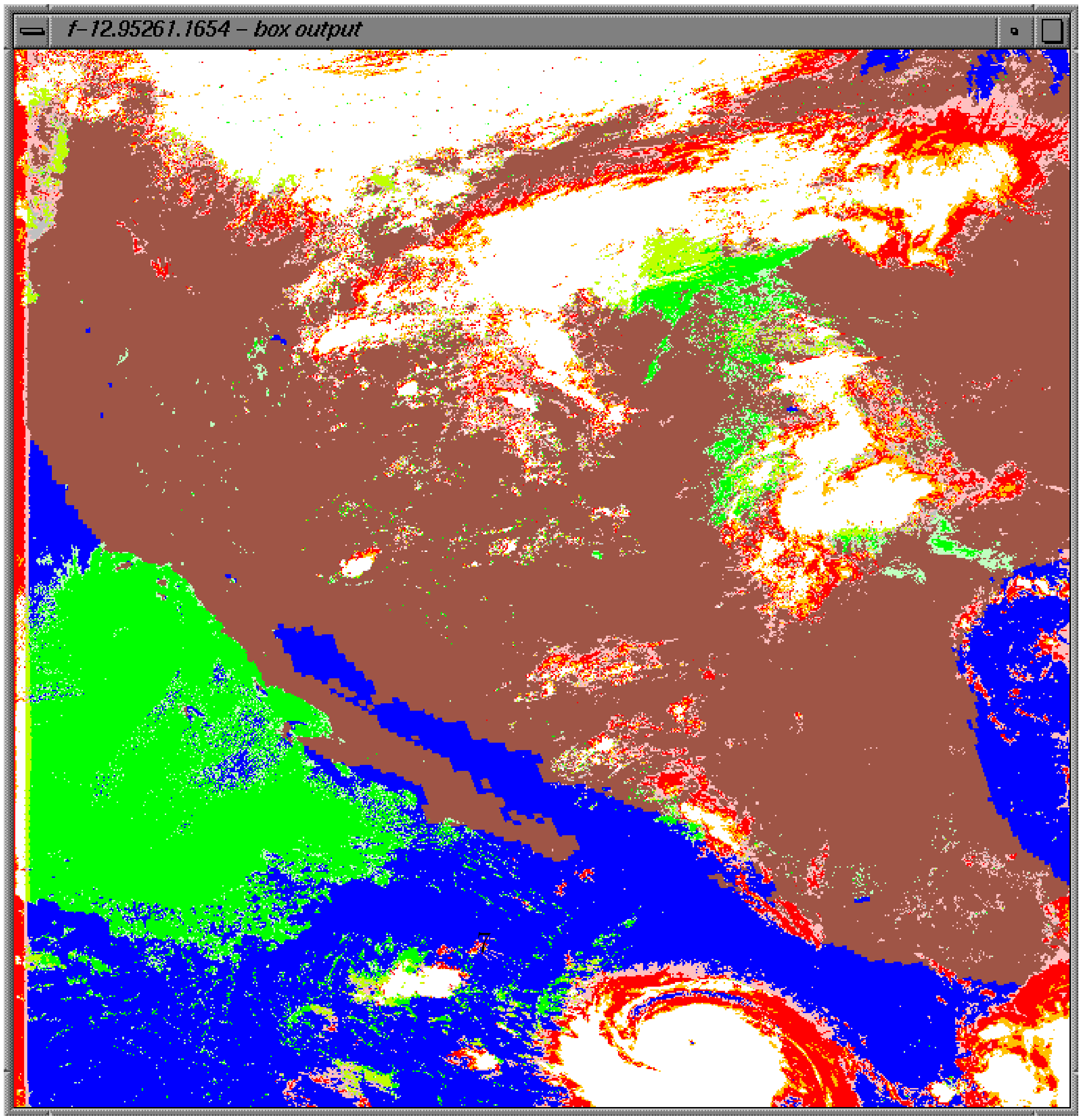

Figure 1: *Output Cloud Mask. Red indicates cloudy pixels detected only in the infrared image. Green indicates cloudy pixels detected only in the visible light image. White and gray indicate agreement. Others colors are used for others elements of the decision matrix.*

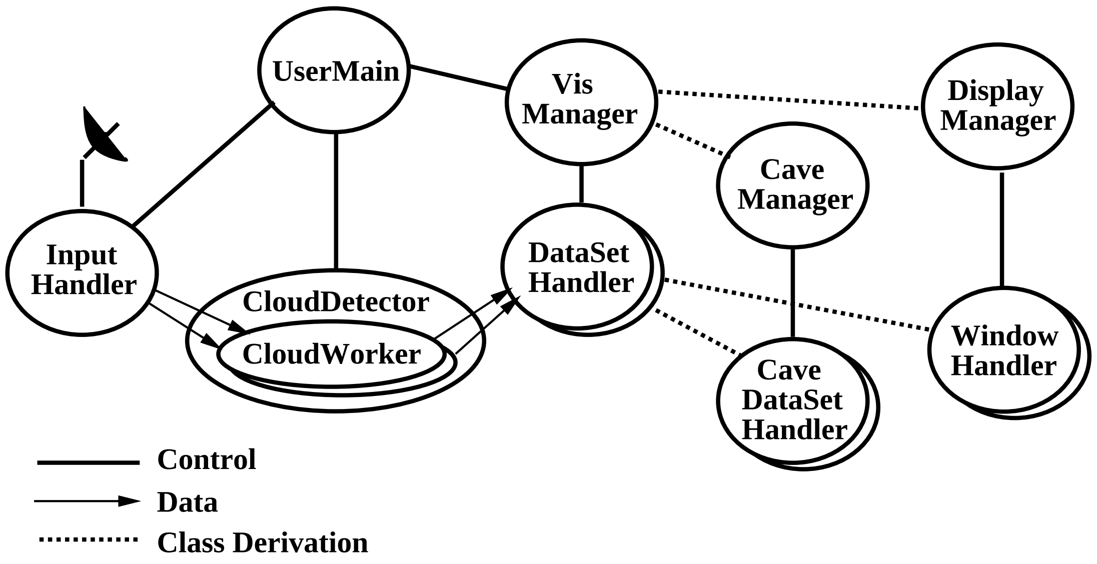

Figure 2: *NEPH architecture.*

where whole images are decomposed into smaller subregions on which the analysis algorithms are run. Hence, this allows for a straightforward data decomposition approach for extracting parallelism which is "embarrassingly parallel" or "parallel machine friendly". Typically a sub-analysis box is 32×32 pixels. Since even a low resolution image is typically 1440×1440 pixels, a simple outer loop parallelization can extract a reasonable amount of parallelism where stripes of an image are given to different processors and the output is recomposed for display.

This application has a natural pipeline structure. Since satellite data obviously has to come from a satellite, the input machine must have a downlink capable of receiving, decrypting and storing the raw satellite telemetry. Once this data hits the ground, it can be streamed from the input machine to any set of machines capable of running the analysis algorithms. The output of these cloud detector computations can then be reassembled for display on an appropriate display machine.

The entire application structure is illustrated in Figure 2. The

UserMain processor object is started by the user and is responsible for acquiring all other resources, managing independent and dependent initialization and clean termination. The InputHandler processor object acquires current satellite data (or opens a file of previously acquired data) that are partitioned for subsequent processing. Each CloudDetector processor object manages one or more CloudWorkers which share common conversion tables, etc., within a CloudDetector. The InputHandler streams image partitions to the CloudWorkers by invoking a CloudWorker member function through a global pointer. While parallelism is realized across multiple CloudDetectors, parallelism is also realized by multiple CloudWorkers within a CloudDetector when the detector is hosted on a shared-memory multiprocessor.

A variety of display technologies are possible for this application. The simplest display is a two-dimensional pixel map using grey scale or false color (as already shown in Figure 1) that is updated whenever processed data becomes available. Another possibility, however, is a three-dimensional rendering. Since the higher a cloud is, the colder it is, the infrared data can be used to derive the altitude of the cloud tops. Hence, each pixel can be located at some altitude over a specific latitude and longitude allowing for a three-dimensional rendering of the cloud tops.

To accommodate this functionality, a simple class hierarchy was developed. NEPH uses a VisualizationManager abstract class which controls any number of abstract DataSetHandlers. Each DataSetHandler reassembles data partitions from the CloudWorkers. For two-dimensional displays, a DisplayManager and WindowHandler are derived from VisualizationManager and DataSetHandler, respectively. For stereoscopic displays, such as a CAVE or ImmersaDesk, a CaveManager and CaveDataSetHandler are derived. Needless to say, the virtual member functions called by the CloudWorkers via global pointer are implemented differently in the two derived classes. VisualizationManager semantics require that any derived class implement a basic event loop such that the user can interact with the application and the display technology as appropriate.

4.2 The I-WAY Host

During development, NEPH was hosted on a variety of workstation clusters and symmetric multiprocessors. At Supercomputing '95, NEPH

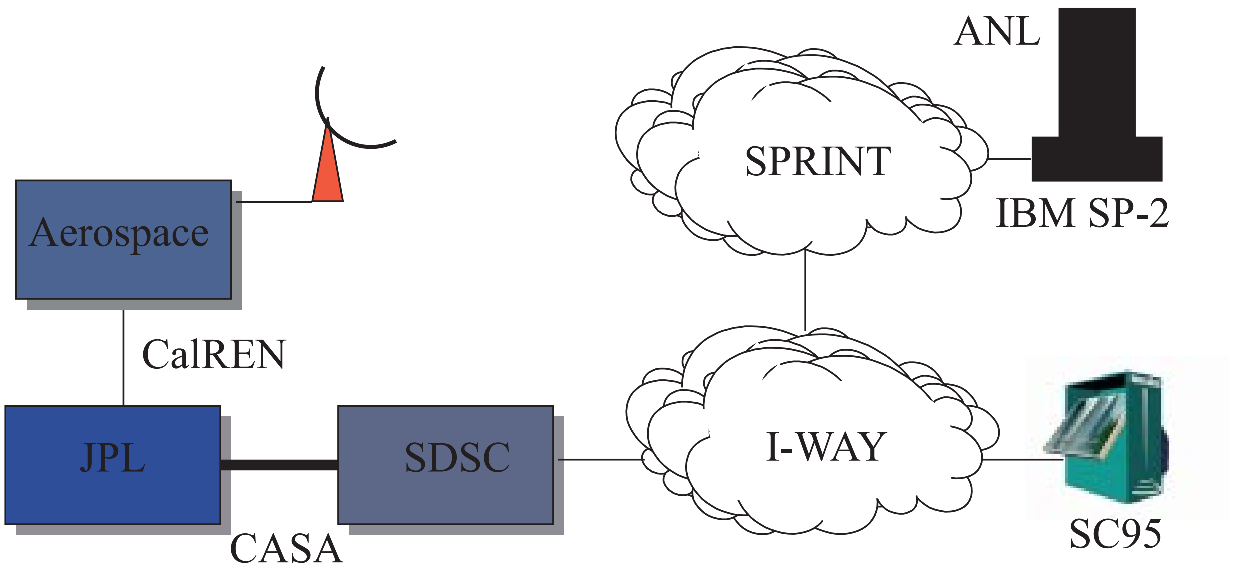

Figure 3: *NEPH's I-WAY host.*

was hosted and demonstrated on the I-WAY [2]. (See Figure 3.) The `InputHandler` was hosted on a specialized Sparc-10 at The Aerospace Corporation near Los Angeles. This machine is configured with a satellite dish and special-purpose hardware to receive, decrypt and store raw telemetry. (This is why NEPH is termed “near-real-time”; the data first lands on a disk prior to being partitioned for processing.) `CloudDetector`s and `CloudWorker`s were hosted on twenty nodes of the IBM SP-2 at Argonne National Laboratory outside Chicago. `UserMain` and the `CaveManager` were hosted on an SGI Onyx running an ImmersaDesk at SC95 in San Diego. Nexus servers were used to acquire these resources and ATM virtual circuits were used to communicate among them.

Figure 4 shows one of the data sets processed over the I-WAY. This is a stereoscopic view looking over the top of a hurricane north towards Baja California and Mexico from the same data set shown in Figure 1. Using the multithreading capabilities supported by CC++ and Nexus, the `CaveManager`’s basic event loop was written to allow simultaneous interaction between CAVE events, such as wand movement and button clicks, and input from the `CloudWorker`s. Hence, a simple CAVE graphical interface was developed such that the user in San Diego could query the `InputHandler` near Los Angeles for the latest available satellite data passes, select a pass for processing by the `CloudWorker`s outside Chicago, and then navigate through a stereoscopic cloudscape as partitions of processed cloud data arrived back in San Diego. The graphical interface also allowed the user to change thresholds, color maps, and terminate the application.

The bandwidth demands placed on the I-WAY were actually quite modest since the Sparc-10 could only support the input demands for a limited number of `CloudWorker`s. The `InputHandler` can, however, stage the data for later processing from a faster machine. Communication bandwidth demand can also vary since the data volume involved in a satellite pass can vary. A satellite that passes directly over the input machine has the longest contact time and, hence, can download the most data. For this demonstration, only low-resolution data was used. This means that a typical NEPH dataset that includes infrared, visible light, latitude and longitude data is approximately 15 MB which is partitioned into stripes of 46 KB that are delivered to the `CloudWorker`s on demand. High-resolution data will increase the data volume by 25$\times$ which could use a communication bandwidth of

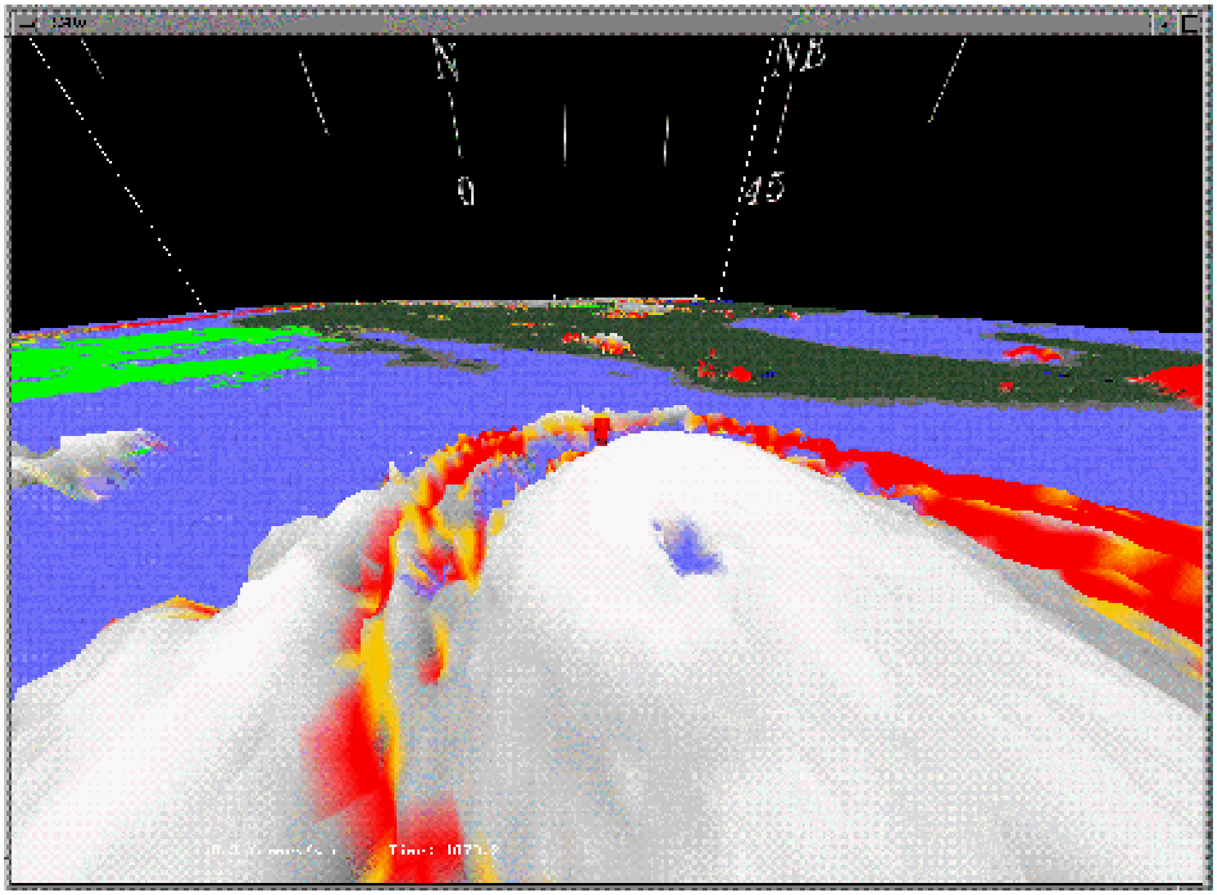

Figure 4: *Stereoscopic view over top of hurricane towards Baja California and Mexico. Vertical scale is exaggerated.*

$\sim$ 184 Mbits/second depending on the exact number of `CloudWorker`s used.

5 Discussion

The metacomputing style offered by CC++ and the functionality supported by Nexus worked exceedingly well for this application. The object oriented semantics of CC++ allowed the application to be built according to its natural structure. The flexibility encapsulated by Nexus allowed us to easily host the application on a variety of compute resources. There are, however, many application and research groups are implementing large, heterogeneous systems and working on issues related to those being addressed by CC++ and Nexus. A fundamental issue is what advantage does metacomputing using global pointers at the language level have over some library level paradigm such as Remote Procedure Call (RPC), a client/server approach, or message-passing with PVM [6] or MPI [4]? How is calling a remote member function with call-by-value arguments significantly different from marshaling arguments for an outgoing message and perhaps blocking for a return message?

The first answer is that of *uniformity* and *transparency*. The same functional interface is used for remote control and communication that is used for a local function call in sequential code. In addition, arguments are *typed* in a way that is meaningful to the application. There is no artificial run-time enumeration of message "types" that must correspond to the type of data being sent or the remote processing that must be done with it. Since remote member function arguments are truly typed, the compiler can enforce consistency. The second answer is that global pointers are a *first-class data type*. Global pointers can be freely passed among processor objects and used to access data and functions regardless of location while still being subject to the rules of C++ member access. This is a much more structured and powerful way of managing control and data.

Another fundamental comparison can be made between global pointers and *global variables*. Global variables typically imply a shared name-space or data-space ([9], for example) which can be easier to use and very efficient when supported in shared-memory hardware. Supporting this paradigm in a distributed environment, however, requires the same kind of caching, consistency and access control but in software rather than hardware. Global pointers force the application designer to be explicit about data locality. Virtual shared memory, on the other hand, does not and can incur a high and nondeterministic overhead if an inappropriately sized data region is cached or if the data region does not correspond to logical data objects. Virtual shared memory can also encourage programming techniques that require a finer granularity than what can be supported efficiently.

A related but different approach is taken by Legion [11]. Legion provides a global name-space but *only for objects.* Legion handles this internally by building *Legion Object Identifiers* which must be bound to physical *Object Addresses* that the underlying run-time system can use to actually interact with a target object. Hence, direct references to members of a global object are implemented as references through its physical Object Address. This is much the same internal functionality as a global pointer; the difference being that a CC++ global pointer contains all the information that Nexus needs to access a remote object whereas a Legion Object Identifier initially needs to be bound to a physical Object Address by a *Binding Agent* that may need to consult a *LegionClass* object to determine the binding.

Besides these issues of the fundamental approach to metacomputing, much work is being done in the area of integrating object-oriented parallelism, threaded environments, high-speed communication and virtual reality (VR) technologies. A number of parallel languages based on C++ exist [12]. While CC++ is does not directly support data-parallelism, data-parallel class libraries can be built that have much the same semantics as concurrent aggregates [10]. Threads and communication have also been extensively studied [8, 7]. In contrast to some of these systems, threads are not explicitly used at the application level in CC++ and since the CC++ compiler translates source code into vanilla C++, it can be hosted on any machine with a C++ compiler allowing greater portability. Steps in the integration of communication and VR have been taken by systems such as CAVEcomm [3]. CAVEcomm is based on a single-threaded subset of Nexus and is used to connect supercomputers with a message-passing interface that is tailored for virtual environments. The work reported in this paper encompasses all of these areas by integrating metacomputing and VR in a multithreaded environment at the language level.

6 Future Work

Future work in these situations is always divided between application issues and infrastructure issues. Application issues include the processing of high resolution data (a 25× increase in data volume), cloud layering and typing (to derive more information about the cloud structure), and volumetric cloud rendering.

The infrastructure or metacomputing issues are quite fundamental. The flat, static resource database currently used by Nexus is only adequate for flat, static application configurations. Capabilities must be introduced into Nexus for *resource identification* and *resource discovery*. This must be a general, possibly hierarchical and distributed method whereby an application can query remote sites for available resources of a given type and configuration and then negotiate for acquisition of those resources. Besides meaning simply some number of processors, "resources" can mean memory space, disk space, access to globally named storage objects (file systems) and bandwidth between sites. Resources could also mean properties such as *security*, *fault tolerance*, and *quality of service* on a given bitway.

The integration of these capabilities at the language level in fact requires a closer integration of the run-time system, the operating system, and communication in addition to display technologies. Threads of control that relate directly to application operations should be tied more closely to communication protocols and OS scheduling policies in order to maximize performance. Remote communications can be streamlined by using one-copy and zero-copy user-space device drivers. Reduced communication latencies, direct support of control operations, and generally making network performance as close as possible to that of a backplane will be necessary to support this style of metacomputing at the language level. When this is accomplished, we will be able to interact with a metacomputer through a virtual environment in ways that have yet to be imagined.

Acknowledgments

The authors thank the Defense Meteorological Satellite Program for access to their equipment and data, 1st Lt. Brian Loomis USAF for assisting in the development of the graphics code for NEPH, Ian Foster of Argonne National Lab, Cheryl DeMatteis and Bob Lindell of The Aerospace Corporation for reviewing an earlier draft of this manuscript, and Matt Szymanski of the Electronic Visualization Lab for loaning us the "compass halo" code that was integrated into the `CaveManager` and appears in Figure 4. More information on CC++ and Nexus can be obtained from http://www.compbio.caltech.edu/ccpp/ccpp-lib.

References

[1] K.M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In *Languages and Compilers for Parallel Computing, 6th International Workshop Proceedings*, pages 124–44, 1993.

[2] T. DeFanti et al. Overview of the I-WAY: Wide area visual supercomputing. *Intl. J. Supercomputing Applications*, 10(2), 1996. (in press).

[3] T. Disz et al. Sharing visualization experiences among remote virtual environments. In *Proceedings of the International Workshop on High Performance Computing for Computer Graphics and Visualization*, 1995.

[4] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, May 1994.

[5] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *J. Parallel and Distributed Computing*, (to appear).

[6] Geist et al. PVM3 user's guide and reference manual. Technical report, Oak Ridge National Laboratory, May 1993. TR ORNL-TM-12187.

[7] S.C. Goldstein, D.E. Culler, and K.E. Schauser. Lazy threads, stacklets, synchronizers: Enabling primitives for compiling parallel languages. Technical report, University of California at Berkeley, 1995.

[8] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: a talking threads package. In *Supercomputing '94*, pages 350–359, 1994.

[9] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[10] C.F. Kesselman. Implementing parallel programming paradigms in CC++. *Proceeding of the Workshop on Parallel Environments and Tools*, 1994.

[11] M. Lewis and A. Grimshaw. The Core Legion Object Model. Technical report, University of Virginia, 1995. TR CS-95-35.

[12] *Parallel Object Oriented Methods and Applications*, 1996. http://www.acl.lanl.gov/Pooma/intro.html.