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 lan-

guage 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

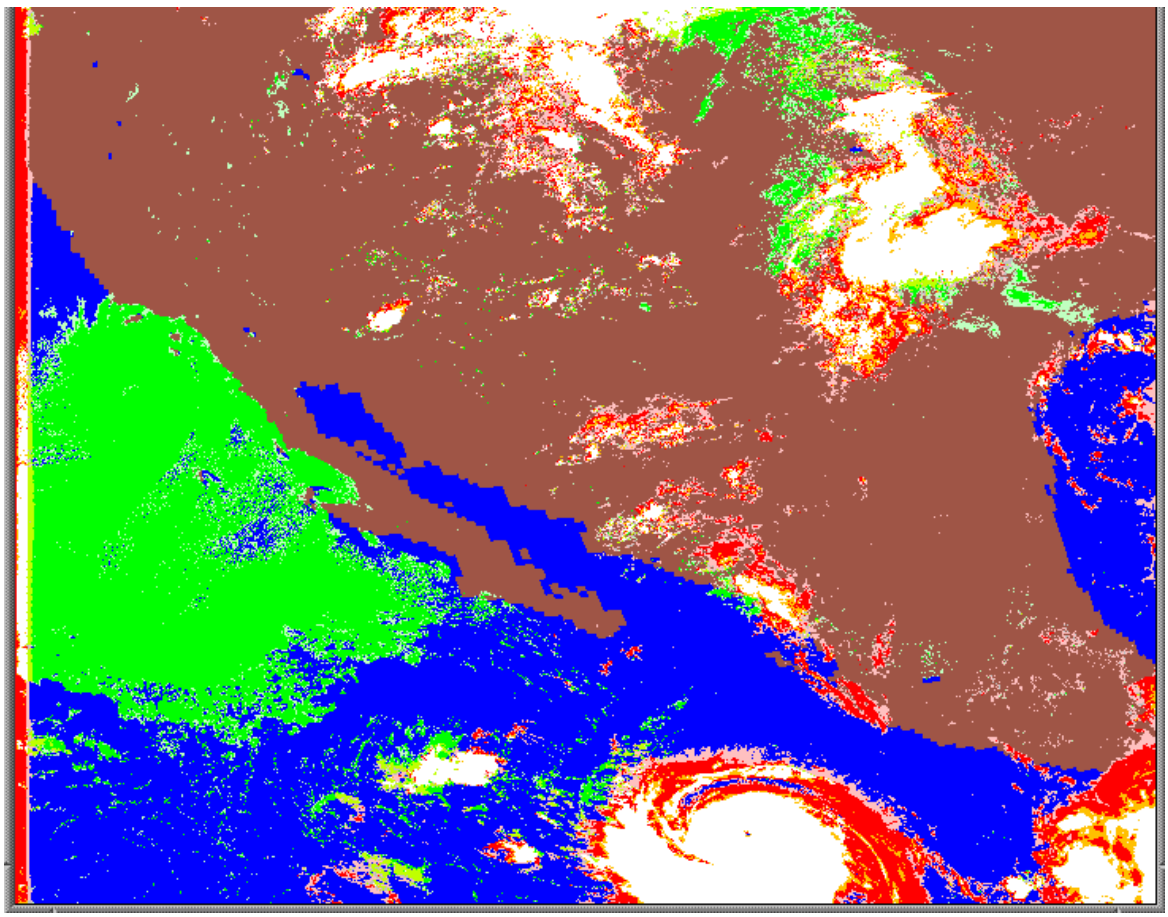
contexts.

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*,