CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 1(1), 1–64 (AUGUST 1993)

Integrating Multiple Parallel Programming Paradigms in a Dataflow-based Software Environment

Gang Cheng and Geoffrey C. Fox
Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY 13244

SUMMARY
By viewing different parallel programming paradigms as essentially heterogeneous approach in mapping "real-world" problems to parallel systems, we discuss methodologies in integrating multiple programming models on an massively parallel system such as Connection Machine CM5. Using a dataflow based integration model built in a visualization software AVS, we describe a simple, effective and modular way to couple sequential, data-parallel and explicit message-passing modules into an integrated parallel programming environment on a CM5. A case study in the area of numerical advection modeling is given to demonstrate the integration of data-parallel and message-passing modules in the proposed multi-paradigm programming environment.

1. Introduction

One of the major issues in parallel processing concerns about homogeneous versus heterogeneous at both software and hardware levels. In parallel software development, currently there are two mainstream approaches to deal with this issue. One is to attempt to hide the heterogeneity of architectures from the programmer. This approach usually employs software technology, especially advanced compiler techniques, to present users with a homogeneous, high level programming model. This methodology leads to highly portable programs across multi-platforms of different architectures and programming models and greatly simplifies programming on parallel machines, while among its drawbacks are hard to obtain most efficient programs for a general application on a particular machine, and the extremely complex compiler design. Among examples of this approach are the FortranD project by Northeast Parallel Architectures Center at Syracuse University and Rice University[11], various automatic tools, and the efforts of HPF[13] and the emerging MPI message passing standard[7]. The other approach acknowledges the heterogeneity at both software and hardware levels, by extending existing programming languages and software environments into parallel environments best exploiting the high performance of underlying architectures. The advantage of the latter approach is the resultant highly efficient programs and an open software environment most flexible to be able to support an integrated heterogeneous processing environment which we will discuss in this paper.

While the first approach is promising(attractive) and represents a major direction in parallel programming language development, we believe that the future of high performance computing lies in the integration of existing and emerging parallel architectures and technologies into a heterogeneous networked computing environment, to be able to solve general classes of applications with various requirements in scientific computation, communication, programming models, hierarchy memory, I/O, data processing and visualization. Another advantage of this approach is in the adopting of existing programs where only those computationally intensive components need to be rewritten to run on

CCC 1040–3108/93/010001–20

Received 12 July 1995
Revised 4 September 1995

supercomputers. This is especially important in porting a large existing application system when a significant proportion of the sequential program are those "interface or management code", i.e. program segments do not perform real scientific computational tasks but simply manage various system resources, such as standard I/O, graphical user interface, system control interface, OS or file system interface and networking interface, to name a few. This heterogeneous approach, together with our view of parallel software as a mapping process between two complex systems - problems and parallel computers[10], would require integration methodologies and tools to facilitate multiple parallel programming models in an unified environment. System integration is expected to play a critical role in parallel software development and applications.

In this paper, we discuss issues in integrating programs written in different programming paradigms and languages on a MIMD machine Connection Machine CM5. Using a dataflow-based integration model built in a commercially available software Application Visualization System(AVS) to a case study of an atmospheric advection modeling application, we demonstrate a simple, effective and modular way to seamlessly couple sequential, data parallel and explicit message-passing parallel modules into an unified programming environment on CM5.

2. Multi-paradigm Parallel Programming Environments

2.1. Multi-paradigm Programming in Mapping Problem Domain to Parallel System

For this article, we shall consider that a *programming paradigm* is a model or an approach employed in solving a programming problem with a restricted set of concepts. Programming paradigms either induce or are induced by a class of languages, the programming environments that these languages are supported in, and the software engineering discipline one uses to produce systems within these environment. Typically, each class places different requirements on its associated programming environment[12].

On existing parallel systems, there are two dominant parallel programming paradigms:

1. Data parallel, in which more or less the same operations is performed on many data elements by many processors simultaneously. Data parallel program exploits parallelism in proportion to the quantity of data involved hence offers the highest potential for concurrency. Data parallel program is also easy to write, understand, debug and port.

2. Explicit message passing which allows two or more operations to be performed simultaneously. A message-passing program is free to arbitrarily distribute program tasks and data, using synchronization and data communication between nodes when it is necessary. Message passing paradigm allows programmer to have maximum flexibility and control in exploiting domain decomposition and control decomposition of the application problem so that maximum efficiency utilizing a MPP hardware can be achieved by careful design of explicit message passing parallel algorithms.

Like the trade-off among various conventional programming languages, expressiveness and efficiency are two major factors featuring parallel programming models and it is even more remarkable because the driving force using a parallel computer is the performance requirement, and programming on parallel machines tends to be more difficult and less portable. The programming languages for massively parallel processors must go beyond

Figure 1. *Computation and simulation as a series of mappings*

conventional languages in expressivity and cost/performance. Unfortunately, unlike on sequential machines and due to efficiency consideration on a particular SIMD/MIMD architecture with an unique interconnection topology, most parallel systems support only one single programming model, i.e. either data parallel or message passing. Given a targeted machine and an application problem, usually a decision must be made to choose which parallelism to solve the problem, early before looking into the problem, let alone at the implementation stage. This is not natural and should be avoided, as mostly for a large application problem some components may be well suited to data parallel, while others to message passing paradigm. We should exploit the parallelisms at each level of computations. There has been long debt in parallel community and industry on the issue of data-parallel versus message-passing programming, as well as SIMD versus MIMD machines.

As studied in [10], by viewing "real-world" problem and parallel computers both as complex systems, the fundamental issue in solving a problem on a parallel computer becomes whether we can easily and efficiently map a problem structure to a particular parallel architecture. A general mapping followed by a computer simulation typically consists of several mapping stages (shown in Figure 1). In this paper, we are interested in the software environment and programming models used to effectively map a large application onto a massively parallel system.

Parallel systems are built to solve large scale computationally intensive applications(e.g., "grand-challenge" problems). The broad range of HPCC applications may embody a rich diversity of program structures. It is anticipated that no one parallel programming model will satisfy all needs. Data parallel, task parallel, and object parallel have all found applicability to end user problems[21,15]. A useful parallel programming environment for MPPs must incorporate semantic constructs for delineating program parallelism, identifying locality and specifying data set partioning, as well as the normal requirements in conventional software engineering. It will have to permit user accessibility to a mix of parallelism models, either by incorporating them into a single schema or by providing means for a single application to use different languages or different parallelism models for separate parts of a user problem. In either case, a parallel software system will of necessity support interoperability among independently derived program modules, perhaps even running on separate computers in a distributed heterogeneous computing environment. This is

Figure 2. Integration of multiple programming models as a single mapping

especially important for irregular and dynamic problems, and for multidisciplinary applications, as it is often inevitable in those applications to require coordination or composition of multiple paradigms programming, e.g., high-level data-parallel and low level explicit message passing, considering the complex of problem structures to be expressed and solved by the software environment. [9] gives such an example of multi-model earth system in which an ocean model and an atmosphere model are loosely coupled each other.

While the homogeneous approach emphasizes a virtual programming model due largely to portability consideration, and high-level and low-level software are used in a strict successive manner in the mapping as shown in Figure 1, it is most likely that a large application problem would require programming models of different abstraction levels to be used in a hybrid fashion. For instance, Figure 2 gives another picture of how subtasks of an application task are mapped to a parallel system in a single mapping stage by multiple programming models.

A problem task may consist of subtasks requiring a set of computing services such as parallel computation/communication, parallel database management, parallel or sequential I/O, real-time visualization, sequential event-driven graphical user interface, networking interface, etc. This mapping process will require not only a "universal architecture" of the parallel computer to be able to efficiently support multiple programming models, it also requires a software environment capable of supporting task-level computation/communication and integrating different programming paradigms in a single framework. Software integration techniques are clearly required in interfacing data and control flows among components(modules) of different programming models in this single mapping process.

CM5 is one of a few today's parallel computers which support both data parallel and explicit message passing programming models. We choose it to evaluate the methodologies of multi-paradigm integration in this paper. However, we believe that many of the issues discussed in this paper are also applicable to the programming integration in a heterogeneous computing environment in which multiple parallel systems are connected by high-speed LAN and/or WAN networks.

2.2. Multi-paradigm Programming

In this section, we first examine integration techniques for multi-paradigm programming on a sequential machine and then discuss corresponding implementations under a parallel context such as on a Connection Machine CM5.

On a conventional uniprocessor machine, multi-paradigm integration can be in general divided into following four classes, based on the interface of how one paradigm is combined in the others:

1. Full integration — a single language has constructs supporting multiple programming paradigms. The transfer of program control and data among different language components at runtime are implicitly arranged by the compiler. The semantics of interfacing different paradigms is built in the language specification. Usually, a programming language evolves along this line, with a single paradigm at initial stage and new language components added in as applications grow and different paradigms are required. The evolvement of C++ from C(which in turn is evolved from Pascal), which now supports imperative, modular and object-oriented programming paradigms, can be thought as an example of this category. Another example can be found in [5] in which functional and logic programming paradigms are integrated in a single language. Among this approach in parallel programming language, compositional C++(CC++)[2] and FortranM[8] are examples attempted to integrate in a single language framework different paradigms of parallel programming such as data parallel, task-parallel and object-parallel paradigms, imperative and declarative programming, shared memory and message-based programs.

   This approach provides the most flexible and efficient hybrid integration of multiple models. However, new languages have to be developed and complex semantics of interfacing the multiple programming paradigms is usually inevitable. On a MIMD parallel system like CM5, because data-parallel and message-passing paradigms have dramatic implementation and performance differences in data mapping/processor allocation, task decomposition and execution synchronization, and require unique semantics inherently in both the programming language and underlying system architecture, this integration approach may incur interface overhead and unexpected complex semantics thus seems not feasible on a distributed-memory machine.

   The current CM programming environment provides some restricted ways to support this kind of integration, mostly through specialized library routines which encapsulate optimized implementation details of the functions provided. As an example, the CM Fortran Utility Library provides convenient access from CM Fortran to the capabilities of lower-level CM software. The purpose is typically to achieve functionality or performance beyond what is currently available from the compiler. CM Fortran programmers can use the library procedures in situations where one is normally tempted to make explicit calls to lower-level software. The utility procedures take CM Fortran array names and other CM Fortran data objects as arguments so that there is no need to convert CM Fortran objects into the data types by lower-level software. The CM Fortran Utility Library provides procedures for inquires, random number generation, dynamic array allocation, interprocessor and local data motion, and parallel I/O[23].

2. Embedded integration — master-slave style. The interface is a built-in construct (or

convention) in a master language to allow seamless invocation of subroutines(functions) written in a slave language. No global data can be shared directly by both languages, even they have the same addressing memory space. Data transfer among modules in master and slave languages is identical to that in a single language and is carried out by passing pointers to the same shared memory so that there is no data copy cost. Separate compilations of different languages are required and their object programs are finally linearly linked together. The executable program runs as a single process. This approach provides a simple, coherent yet efficient way to support the integration at language level. Among examples of this approach are interlanguage calls on most conventional platforms, such as from C calling Fortran77 or assembly, or vice visa. This is the common approach on most SIMD and MIMD machines to support extension of parallel programming in host sequential languages, such as on CM5 calling CMfortran, C* or CMMD message passing primitives from Fortran77 or C.

In order to exploit the special high-performance hardware arithmetic accelerator (called vector unit) in each processing node(PN) of the CM5 system, current CM5 programming software provides some limited way to mix data-parallel paradigm in message-passing program. Explicit message passing CMMD programs can use certain language constructs from CMFortran or C*, in which case the programs use both the microprocessor and the vector units of a PN, with serial arrays stored in the microprocessor and parallel arrays in the VU memory. It can thus take advantage of the VUs for high-speed floating-point computations. Parallel arrays can be passed as arguments to CMMD message passing primitives in much the same way as serial arrays. Within this integration model, data parallel operations are confined in a "node-level" while the overall program is in explicit message-passing paradigm.

If adopting this approach on a CM5, an ideal integrated program would be as follows: Executed as a single process on the control processor(CP) of the CM5, the main program in Fortran77 (or C) would first act as a host sequential program in the CMMD host/node programming model and starts a CMMD node program by enabling and broadcasting some data to the node program which is written in Fortran77, C or even CMfortran(node-level data-parallel), then this host program invokes another subroutine written in CMFortran (or C*). After the control is back to the main program from the CMFortran subroutine, it collects the results sent from the node CMMD program and finally ends. Before and after enabling the CMMD node program or invoking the CMFortran subroutine, there can be any other non-parallel computing subroutines in the sequential host program. Technically, however, this approach would pose significant difficulties on the complier(linker) as it is now the compiler's responsibility, as opposed to that of a third party daemon or the OS in the process-based approach described below, to manage the time-sharing of CM5's PNs and internal networks among the CMFortran subroutine and the CMMD node program.

3. Loosely coupled integration — process-based integration. Different individual program modules written in same or different languages are complied and linked separately, while their executables run concurrently as multiple processes time-sharing a CPU. Message passing among program processes is normally via socket-based interprocess communication(IPC) though if two modules share a host it is common to use shared memory to optimize the communication. Runtime process control is coordinated by a supervising manager(or called daemon) or the operating system.

Currently most windowing systems and networking applications, for instance the client-server model, follow this approach.

On the host machine of a parallel system, IPC-based systems provide an open environment for the parallel programming software to be naturally migrated into the conventional software environment in which software such as concurrent operating system, networking, graphics, I/O, and sequential programming languages have been well-established for many years. This process-based approach also opens great opportunities to construct a "metacomputer" via a networked environment[18]. Depending on the way (protocol) to facilitate the IPC, such as data exchange, management of buffers and sockets, threading, process control and synchronization, an integration system of this kind can be quite different. For example, PVM[20], a process-based general message-passing system, provides a set of user interface primitives that may be incorporated into existing procedural languages. In our work, we use another process-based system, AVS[24], which was primarily used for visualization applications. We will demonstrate in later sections the advantages of the AVS's dataflow IPC model in facilitating data-parallel and message-passing programming paradigms on CM5.

4. Not-at-all integration — non-interactive style. Data passing and sharing among different program modules, usually stand-alone executables, are carried out at (UNIX) operating system level through shell scripts, pipes, intermediate data files, and other pre- or post- processing techniques. This is the usual way for an application to collectively use the (parallel) computing services when it does not require their programming tasks to run in an interactive way. High system overhead is incurred in this approach.

2.3. Parallel Programming on Connection Machine CM5

Detailed description about CM5 architecture and software system can be found in [22] and related CM5 documents from TMC. In this section, we briefly outline existing parallel programming environment on CM5, for purpose of our discussion in later sections.

The CM5 system combines the features of SIMD and MIMD designs, integrating them into a single parallel architecture. The unique feature of CM5 provides us the alternatives to exploit both paradigms in a parallel programming environment.

The CM5 architecture is designed to support especially well the data parallel which implies a relatively synchronous parallel programming featuring a single conceptual thread of control and a global approach to data layout and I/O, in which arrays of data are laid out across all processors, and all elements are computed upon simultaneously, by their respective processors. Although the conceptual model itself is completely synchronous, in practice processors may execute asynchronously any operations that are independent of other processors, such as conditional operations on locally stored data. The CM5 data parallel compilers take full responsibility and advantage of this freedom to exploit the MIMD hardware capabilities of the CM5. Currently, CM5 supports data parallel Fortran(CMfortran), data parallel C(C*) and data parallel Lisp(*Lisp) programming languages.

Explicit message passing model is extended from the data parallel model on CM5, by using a package of macros and runtime primitives called CMMD supporting MIMD style low level communications operations. Programs that use CMMD typically operate in a

Figure 3. *Components of a CM5 system*

SPMD(single program multiple data) style in that each processing node of the CM5 runs an independent copy of a single program and manages its own computations and data layout, and communication between nodes is handled by calls to CMMD message passing primitives. The message passing programs can be written in Fortran 77, CMFortran, C, C++, C* and an assembly language DPEAC.

As shown in Figure 3, each processing node(PN) is a general-purpose computer that can fetch and interpret its own instruction stream, execute arithmetic and logical instructions, calculate memory address, and perform interprocessor communication. All PNs can perform independent tasks or collaborate on a single problem. A control processor(CP) is similar to a standard high-end workstation acting as a partition manager to communicate with the rest of CM5 system through the Control Network and Data Network. CMOST, a parallel timesharing operating system enhanced from the UNIX, runs on the CP to make all CM5 resource allocation and swapping decisions, as well as most system calls for process execution, memory management, I/O and access to local networks. The Control Network provides tightly coupled communication services. Optimized for fast response and low latency, its functions include synchronizing the PNs, broadcasting data to every node, combining a value from every node to produce a single result, and computing certain parallel prefix operation. The Data Network provides loosely coupled communication services. Optimized for high bandwidth, it provide point-to-point data delivery. A data parallel or CMMD message passing process running on the CP plus the nodes is a single process time-sharing the CM5 system with other processes.

3. AVS - A Dataflow Based Integration Tool for Multi-paradigm Programming on CM5

3.1. The Application Visualization System

AVS[1] is a software development toolkit based on a dataflow model for scientific data visualization and process control. It incorporates visualization, graphics, visual programming, process management and networking into a single comprehensive software development environment. We are most interested in its software integration capability in this article.

The AVS dataflow is a model of parallel computation in which a flow network of autonomous processes compute by passing object data along arcs interconnected by input/output ports. Each module, represented as an individual UNIX process, fires autonomously as soon as new input object it requires arrives on its input ports. It then produces output object that flows on its output port thus triggers further computations. A set of base communication objects is pre-defined such as numbers and fields (n-dimensional

Figure 4. *A general AVS/CM5 integration model for multi-paradigm programming*

arrays) and a collection of computational modules is provided that transform sets of input objects into sets of output objects. AVS itself offers an extensive library of ready to use modules mostly for graphics programming and filtering, as well as it specifies the protocol for developing and including user modules into the system. Specific applications are assembled from reusable modules with the aid of visual network editing tools. Modules are visualized as nodes, ports as terminals, and communication objects as links of an interactively constructed dataflow network.

Both process control and data transfer among processes in AVS are in a modular fashion and is completely transparent to the programmer. AVS provides a data-channel abstraction that transparently handles module connectivity and ports type-checking. The module programmer needs only to define the input and output ports in AVS predefined data types and using a set of AVS routines and micros. Message passing occurs at a high level of data abstraction and only through the input and output ports. Each AVS application includes AVS kernel and a set of modules, specified by a dataflow network. AVS kernel executes network specification and instantiates module processes as well as the lower level inter-modular communication links, based on the RPC/XDR UNIX network protocol. Another important feature of the AVS execution model is that it supports remote module execution in which various nodes of the computational graph can be executed on distributed machines on a local or wide area network.

3.2. A General Parallel Programming Environment on the CM5 with AVS

Although AVS does not provide support for parallel computing within a module, a specific module may use machine/language specific features to achieve parallelisms such as data parallel or explicit message passing. Using the modular process management in AVS and the time-sharing CMOST on the control processor together with CM5's parallel programming software, we can naturally combine programs in data-parallel, message passing and sequential programming languages available on CM5 into a single process-based software system. As shown in Figure 4, an AVS/CM5 integration framework has two distinct levels of programming interfaces. At the higher level, AVS dataflow model is used to express and implement data and control flows among various modules which represent decomposed subtasks of a specific application. The inter-modular communication is handled by AVS IPC protocol. Process control (e.g. synchronization and flow control), scheduling and realization of interactive visualization at module level are also handled at this interface and are supervised by the AVS kernel. All the sequential process threads at this level are executed on the same control processor of CM5. The AVS kernel can be on the same CP or on a different machine networked to the CP. At the lower level, a specific CM5's parallel programming model, for example, a CMFortran subroutine or a CMMD program, is used to implement a highly parallelized algorithm in fine-grain parallelism, best suited to solve the subtask represented. All the parallel process threads at this level are executed on the CM5's processor nodes and CM5's specialized data network and control network (i.e. the fat-tree topology) are used for communications among the processor nodes within a module. Data transfer and thread control between the two levels in a process is handled by the host-node protocol in CMOST. An AVS/CM process can be a module or a set of modules written in the same language/paradigm. A typical integrated parallel module will have two distinct threads at the two different levels to be executed on different physical processors of the CM5 system (i.e., CP or PNs). They are represented as two different portions in the program code of a module:

1. *AVS/CM interface part* – Sequential code in Fortran77, C or C++ to coordinate data flow between AVS modules and control flow between AVS kernel and the CM parallel system. It contains code declaring the AVS module's input/output ports and (graphical) parameters. For a data-parallel module, it has program segments for packing input data, received either from other AVS modules or by sequential subroutines in the same module, into parallel data. e.g., transfer serial arrays into data parallel arrays or decompose/load block data onto each processing node, or vice visa. For an explicit message-passing module, it is a host program for the CMMD host/node programming model.

2. *CM computation part* – Data parallel subroutines in CMFortran or C*, or CMMD node programs in Fortran77, C or C++ to perform main calculations requiring intensive computation and communication. This part of the module is almost identical to a standalone parallel program implementing core parallel algorithms for a subtask, if not integrated under the AVS framework, except that it becomes a parallel subroutine invokable from the *AVS/CM interface part*.

After both parts of a module are compiled separately, possibly with different compilers, they are linked into a single executable. Multiple modules can either be compiled and linked individually into separate executables(processes) or they can be linked into a single executable. Modules in different processes must communicate data through the CP under the supervise of the AVS kernel. If the modules are written in different paradigms (e.g. CMFortran or CMMD), they are in different processes. If both modules are in the same process on the CM5, only a pointer needs to be passed, therefore, modules written in the same paradigm can be grouped in a single process and data transfer among them has no data-movement cost.

This integration framework provides a sufficiently high level way to couple modules written in multiple programming paradigms into a single environment on CM5, so that the programmers have the freedom to choose either parallel programming fashion to solve their problem, independently or in a mixed way. This is a general parallel software engineering environment which allows more than one programming paradigms to be used, including sequential and parallel models, data-parallel and explicit message passing programming, concurrent and pipeline execution, task and functional parallelism. Running in an interactive environment, it gives the user maximum control over the optimization of heterogeneous resources and capabilities of a high performance computing system. Other advantages using this integration system include:

1. Modularity and template-oriented — Programs are constructed by using explicitly declared communication channels to plug together program processes. A process can encapsulate common data decomposition, manipulation and internal communication, and the programming paradigm inherent in the code. The modular approach of AVS also lends itself well to software sharing, module reuse, extensibility and flexibility.

2. Module reusability — CM5 data parallel or message passing codes and algorithms are modularly developed and compiled once, and can then be reused indefinitely in a plug-and-play mode in a variety of diverse scientific applications. Because the interfaces between modules are well defined, modules developed at different computer sites, by different developers and in different paradigms may be freely shared. Modules created for one application can be readily be used to another similar application with little or no change.

3. Flexibility and extensibility for rapid prototyping — At first stage of an application development, programmers can concentrate on task decomposition, parallel algorithm design and performance improvement, totally ignore system integration issues. They may write their own modules in their favorite programming paradigm and language binding. The programs are then migrated to the AVS environment by simply attaching and linking the AVS interface code, while the main parallel algorithms can be untouched.

4. Hierarchical parallelisms — At top of the task level decomposition of problem domain, modules that are connected in the dataflow programming paradigm have the potential to run concurrently or in a pipelined way. At this coarse grained process level, functional parallelism can also be achieved such that graphical interface, sequential and parallel I/O, remote networking access and the decomposed task computation can be carried out concurrently by different components of the CM5 supercomputer. The second level of the hierarchy is the hybrid data parallelism in data parallel modules and control parallelism in explicit message passing modules. At the bottom level parallelism can be thought as that inherented in the piplelined processing hardware such as the vector units in the CM5.

5. Safety — Operations on communication channels are restricted so as to guarantee deterministic execution. Channels are strongly typed, so a compiler can check for correct usage.

3.3. Limitations in the AVS Integration Model

The process-based integration system offers many advantages for a parallel software engineering environment in terms of multi-paradigm programming, software re-use and modularity. However, towards an efficient and fully functional multi-paradigm system, the current AVS framework suffers a few significant performance problems and operational restrictions.

1. Inter-modular Communication — The major performance problem is a centralized AVS kernel versus a decentralized parallel system. Restricted by the kernel and its IPC data-flow protocol, any communication between two parallel modules has to go through the sequential host-node channel and the module-module pipeline on the control processor. This is illustrated in Figure 5 in which two modules (m_1 and m_2), each having two separate but linear threads (h_1,h_2 on the CP and p_1,p_2 on PNs). When m_1 needs to communicate to m_2, it has to follow the serialized data transfer path $p_1 \Rightarrow h_1 \Rightarrow h_2 \Rightarrow p_2$, while the most efficient path (or parallel channel) is directly $p_1 \Rightarrow p_2$ on PNs using CM5's high-bandwidth data and control networks. In this sense, current AVS kernel serializes parallel channels. This may not be a performance bottleneck between a data-parallel module and a message-passing module as in most cases scalar data are expected to flow between modules in different parallel paradigms. This is not true for modules in the same paradigm (e.g., CMFortran modules) between which inevitably very large scientific data sets must be transferred. In addition, because each module is buffering its entire data on the input and output ports and all the intermediate data have to reside in memory of the CP, total memory requirement on the CP can be very large. Continuous memory-disk swapping of processes by the operating system can make this situation even worse.

2. Dynamic Flow-network Reconfiguration and Load Balance — Current AVS system provides very limited support to dynamic modification of flow-network, e.g., processes and their communication requirement may change with time - processes can be created or destroyed, communication pattern moves at run-time [19]. It is also difficult in AVS to perform fine-grain dataflow paradigms such as round-way dataflow, i.e., an output port of a down-stream module in reverse is connected to an input port of a upstream module.

   In certain "real world" multitasking applications, the need arises to create or (re)initiate individual modules in a dynamic fashion. Consider for example a signal processing component of a command and control system. The input dataset comes typically from several sensors, is processed by a set of coupled filters and then analyzed by a set of higher reasoning modules. A particular specialized down-stream filter might be requested and activated in runtime by the up-stream reasoning module. Such transient specialized module could not be visually pre-edit in the static AVS flow-network since it is invoked dynamically in a data driven fashion.

   Dynamic reconfiguration of flow-network is also required by load balance issues. The AVS/CM5 system assigns modules to processor nodes in a static fashion and offers no support or visual metaphor for task mobility. A good example is provided by an adaptive multigrid PDE algorithm, operating on a highly inhomogeneous manifold. Such algorithm would start as a single data-parallel task and would often

Figure 5. Serial AVS I/O versus parallel node-node channels for inter-module communication

spawn additional tasks in runtime. For instance, a complex local geometry might be detected which requires a high density local mesh refinement. Local inhomogeneities of this type will be naturally load balanced by dynamically distributing such spots, thereby spawning new concurrent data-parallel PDE tasks.

3. Synchronization Control and Scheduling of Module Processes — The only synchronization mechanism in the AVS model is passive causality provided by the data-flow paradigm. High performance interactive applications often require the near real-time operating system support and tight synchronization of various media streams. An extension of the dataflow model is required to handle such requirements. This includes both the underlying low level message passing system and the high level, scriptable and visual network editing tools. In the current model, the only visual metaphor is that of a graph connectivity, implemented in terms of nodes-modules and links-communications. Suitable objects and visual tokens need to be included to encompress the synchronization metaphor and to provide graphical tools for representing priorities, threads and synchronization objects such as semaphores, mutexes, stacks, FIFOs or message queues.

Performance of these systems on massively parallel machines will in large part depend upon the systems' capability to support high bandwidth and low latency for data transfer between a host machine and all the node processors, and the way how a problem's tasks are decomposed. The task granularity must be restricted to a coarse-grained domain and the task partitioning should be carefully evaluated to consider the balance of each module's computation time and process communication time. System software is needed to support high-bandwidth parallel I/O and shared memory segments between processes. The CM/AVS system developed at Thinking Machines is expected to greatly reduce the process-level data-movement overhead[16,14]. In CM/AVS, CM5 shared-memory regions are used to exchange data, if they are in the same CM5 partition, or they exchange data directly over the routine network via CM-domain socket, if they are in different partition. They

Figure 6. System configuration of the integration system

may communicate with serial modules on a workstation over the fastest available network connecting the various components, using parallel sockets.

4. A Case Study — Comparison of Numerical Advection Models

In previous work, we have demonstrated the use of AVS in two real-world applications, a stock option pricing modeling[3] and an electromagnetic scattering simulation[4], both in the context of scientific visualization and also, more broadly, as an attractive environment for software integration in high performance distributed computing. Those work motivated us to explore further the AVS's capability of system integration for HPCC applications such as GIS[6], four-dimensional data assimilation and environmental modeling, in particular, the methodology to support multi-paradigm parallel programming as described in this paper. In this section, we use a case study to demonstrate the feasibility of integrating CMFortran and CMMD modules on a CM5 in the proposed environment.

In fluid dynamics, the simple one-dimensional advection equation (1) with constant positive velocity serves as a basic model for the shape-conserving movement of an initial distribution of fluid volume toward positive x. Since the analytic solution is known in this simple case, numerical approximations to the advective process can be critically evaluated on fundamental properties such as accuracies, stabilities, coordinate systems and computer

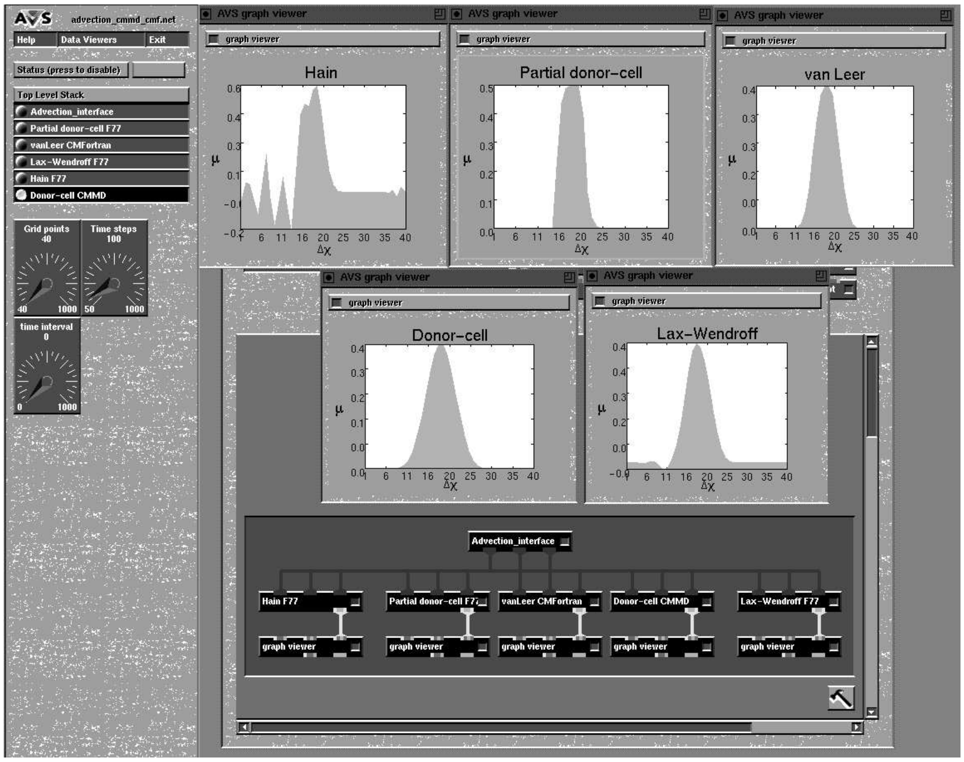

Figure 7. A graphical user interface of the integrated system

time and memory requirements.

$$\frac{\partial \mu}{\partial t} + u\frac{\partial \mu}{\partial x} = 0 \tag{1}$$

where μ is the fluid velocity and u is the mixing ratio $\mu \equiv C/\rho$ in which C is the constituent density and ρ the density of the fluid.

Our case study is to create a prototype interactive computing/visualization environment in which a couple of 1D advection models are purposely implemented in different programming paradigms on CM5 and the results can be graphically compared with through a graphical user interface.

We choose from [17] five numerical advection algorithms with different diffusion, dispersion and monotonicity properties. They are *'Donor-cell'*, *'Partial donor-cell'*, *'van Leer'*, *'Hain'* and *'Lax-Wendroff'*, named after the authors of the algorithms. *'Donor-cell'* is then implemented in CMFortran and *'van Leer'* in CMMD, while the rest models are in Fortran77. All the "*AVS/CM interface part*" codes are written in C.

To demonstrate the system's integration capability on a heterogeneous architectures, we purposely run the AVS kernel on a IBM RS/6000, the *'Hain'* module on a DEC5000 and the *'Lax-Wendroff'* module on a SUN4, while the data-parallel *'Donor-cell'* module, the message-passing *'van Leer'* module and the sequential module *'Partial donor-cell'* are running on a 32-node CM5. The system configuration is shown in Figure 6.

Figure 7 is a screen dumped picture showing the system parameter control panel(left), model output windows(top) and the flow network(bottom). The flow-chart-like diagram is the process configuration built by the AVS visual programming interface Network Editor, in which *'Advection Interface'* is a system control module for model input parameters steering and the *'graph viewer'* is an AVS system module for model graphical output and both run on the same machine as the AVS kernel. The other modules correspond to the respective advection models.

This prototype environment has the following features:

1. User interactive control of the modeling. Before or during a modeling run, the user can graphically choose advection models, set model parameters including execution modes (single step, continuous, pause, abort, or stop), displaying time steps, the total number of grid points and time steps.

2. Graphical output of the model results. The user can decide the displaying pattern, such as line, area, bar, and other graphics parameters such as title, color, etc.

3. Easy and flexible to include other advection models. It can be easily extended to add in other old/new models with varying velocity fields, multidimensions and non-rectangular coordinated systems or with sequential, data parallel or message passing implementations. All the models can be transparently configurated to run in a distributed computing environment.

5. Conclusion

The purpose of this experiment in integrating multi-paradigm programming is to provide programmers with a rich and attractive programming environment with which choice of high and low level programming models can be made upon problem structures, as well as particular machine architectures, to facilitate a gradual and eventually efficient mapping from problems to parallel systems. In the case multiple programming languages are required, this environment supports a seamless and modular way to interface the data and control transfer among processes in different languages.

With its visual programming interface, modular program structure, dataflow based execution, interactive visualization functionality and its open system characteristics, a process-based integration software like AVS provides an excellent framework to facilitate integration of various system components required by a large, multidisciplinary applications, including integration of sophisticated interactive visualization, database management, heterogeneous networking, massively parallel processing and real-time decision making, as well as a useful tool for software development and project planning. We believe that with the adoption of HPCC technologies into industry applications, system integration will play an ever-growing important role in parallel software development and system design.

ACKNOWLEDGEMENTS

We are grateful to Ricky Rood of NASA/Goddard who made the original advection codes available to us.

REFERENCES

1. Advanced Visual Systems Inc. *AVS 4.0 Developer's Guide and User's Guide*, May 1992.
2. K. M. Chandy, C. Kesselman, 'Compositional C++: Compositional Parallel Programming', Technical Report, California Institute of Technology, 1992.
3. G. Cheng, K. Mills and G. Fox, 'An Interactive Visualization Environment for Financial Modeling on Heterogeneous Computing Systems', in Proc. of the 6th SIAM Conference on Parallel Processing for Scientific Computing, R. F. Sincovec, eds., SIAM, Norfolk, VA, March 1993.
4. G. Cheng, Y. Lu, G.C. Fox, K. Mills and T. Haupt, 'An Interactive Remote Visualization Environment for an Electromagnetic Scattering Simulation on a High Performance Computing System', in Proc. of Supercomputing '93, Portland, OR, November,1993.
5. G. Cheng, and Y. Zhang, 'A Functional + Logic Programming Language in Interpretation-Compilation Implementation', Lisp And Symbolic Computation: An International Journal, Vol. 5, No. 3, 1992, pp. 133-156.
6. G. Cheng, C. Faigle, G. C. Fox, W. Fumanski, B. Li, and K. Mills, 'Exploring AVS for HPDC Software Integration: Case Studies Towards Parallel Support for GIS', Proc. of the 2nd AVS Conference AVS'93, Lake Buena Vista, FL, May 1993.
7. *MPI: A Message-Passing Interface Standard*, May 5, 1994. http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html.
8. I. Foster and K. M. Chandy, 'Fortran M: A Language for Modular Parallel Programming', Preprint MCS-P237-0992, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992.
9. I. Foster, 'Fortran M as a language for building earth system models', Preprint MCS-P345-0193, Argonne National Laboratory, and Proc. 5th ECMWF Workshop on Parallel Processing in Meteorology, ECMWF, Reading, U.K., 1992.
10. G. C. Fox, 'Parallel Computers and Complex Systems', Complex Systems '92: From Biology to Computation, Inaugural Australian National Conference on Complex Systems, December 1992. Editors: Bossomaier, David G. Green. CRPC-TR92266
11. G. C. Fox, S. Hiranadani, K. Kennedy, C. Koelbel, U. Kremer, C-W Tseng, and M-Y Wu, 'Fortran D Language Specification', Syracuse Center for Computational Science-42c, Rice COMP TR90-141, 37 pps, 1991.
12. B. Hailpern, 'Multi-Paradigm Languages', IEEE Software, Vol. 3, No. 1, 1986, pp. 54-66.
13. *High Performance Fortran Language Specification*, High Performance Fortran Forum, May, 1993, Version 1.0, 184 pp., Rice University, Houston, Texas.
14. M. F. Krogh and C. D. Hansen, 'Visualization on Massively Parallel Computers using CM/AVS', Proc. of the 2nd AVS Conference AVS'93, Lake Buena Vista, FL, May 1993.
15. K. Mills and G. C. Fox, 'HPCC Application Development and Technology Transfer to Industry', in the Postproceedings of the New Frontier: A Workshop on Future Directions of Massively Parallel Processing, IEEE Computer Society Press, Los Alamitos, CA, July 1993.
16. G. Oberbrunner, 'Parallel Networking and Visualization on the Connection Machine CM-5', the Symposium on High Performance Distributed Computing HPDC-1,September, 1992, pp. 78-84, Syracuse, NY.
17. R. B. Rood, 'Numerical Advection Algorithms and Their Role in Atmospheric Transport and Chemistry Models', Review of Geophysics, Vol. 25, No. 1, pp. 71-100, February, 1987.
18. L. L. Smarr and C. E. Catlett, 'Metacomputing', Communication of the ACM, Vol. 35, No. 6, June, 1992, pp. 45-52.
19. P. A. Suhler, J. Biswas, K. M. Korner and J. Browne, 'TDFL: A Task-Level Dataflow Language', Journal of Parallel and Distributed Computing, No. 9, 1990, pp. 103-115.
20. V. Sunderam, 'PVM: A Framework for Parallel Distributed Computing', Concurrency: practice and experience, 2(4), Dec. 1990.
21. *System Software and Tools for High Performance Computing Environments*, Final report of the Workshop on System Software and Tools for High Performance Computing Environments, Pasadena, California, April 14-16, 1992.
22. Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, Technical Report, Cambridge, MA, October 1991.
23. Thinking Machines Corporation, *CM Fortran Utility Library Reference Manual*, Technical

Report, Cambridge, MA, January 1993.
24. C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz and A. van Dam, 'The Application Visualization System: A Computational Environment for Scientific Visualization', IEEE Computer Graphics and Applications, July, 1989.