GWD-I (Informational)
Advanced Programming Models Research Group
http://www.eece.unm.edu/~apm/docs/APM_Primer_0801.pdf

C. Lee, The Aerospace Corp.
S. Matsuoka, Tokyo Institute of Technology
D. Talia, Istituto per la Sistemistica e l'Informatica/CNR
A. Sussman, University of Maryland
M. Mueller, High Performance Computing Center, Stuttgart
G. Allen, Max-Planck-Institut für Gravitationsphysik
J. Saltz, University of Maryland
August 2001

A Grid Programming Primer

1. Abstract

A grid computing environment is inherently parallel, distributed, heterogeneous and dynamic, both in terms of the resources involved and their performance. Furthermore, grid applications will want to dynamically and flexibly compose resources and services across that dynamic environment. While it may be possible to build grid applications using established programming tools, they are not particularly well-suited to effectively manage flexible composition or deal with heterogeneous hierarchies of machines, data and networks with heterogeneous performance. Hence, this paper investigates what properties and capabilities grid programming tools should possess to support not only *efficient* grid codes, but also their *effective development*. The required properties and capabilities are systematically considered and then current programming paradigms and tools are surveyed, examining their suitability for grid programming. Clearly no one tool will address all requirements in all situations. However, paradigms and tools that can incorporate and provide the widest possible support for grid programming will come to dominant. Across all identified grid programming issues, suggestions are made for focus areas in which further work is most likely to yield useful results.

2. Introduction

Grid programming will require capabilities and properties beyond that of simple sequential programming or even parallel and distributed programming. Besides orchestrating simple operations over private data structures, or orchestrating multiple operations over shared or distributed data structures, a grid programmer will have to manage a computation in an environment that is typically heterogeneous and dynamic in composition with a deepening memory and bandwidth/latency hierarchy. Besides simply operating over data structures, a grid programmer could also have to orchestrate the interaction between remote services, data sources and hardware resources. While it may be possible to build grid applications with current programming tools, there is a growing consensus that current tools and languages are insufficient to support the effective development of efficient grid codes.

Grid applications will tend to be heterogeneous and dynamic, i.e., they will run on different types of resources whose configuration may change during run-time. These dynamic configurations could be motivated by changes in the environment, e.g., performance changes or hardware failures, or by the need to flexible compose *virtual organizations* [34] from any available grid resources. Regardless of their cause, can a programming model or tool give those heterogeneous resources a common "look-and-feel" to the programmer; hiding their differences while allowing the programmer explicit control over each resource type if necessary? If the proper abstraction is used, can such transparency be provided by the run-time system?

Grids will also be used for large-scale, high-performance computing. Obtaining high-performance requires a balance of computation and communication among all resources involved. Currently this is done by managing computation, communication and data locality using message-passing or remote method invocation since they require the programmer to be aware of the marshalling of arguments and their transfer from source to destination. To achieve petaflop rates on tightly or loosely coupled grid clusters of gigaflop processors, however, applications will have to allow extremely large granularity or produce over $\approx 10^8$-way parallelism such that high latencies can be tolerated. In some cases, this type of parallelism, and the performance delivered by it in a heterogenous environment, will be manageable by hand-coded applications. In general, however, it will not be. Hence, what programming models, abstractions, tools, or methodologies can be used to reduce the burden (or even enable the management of) massive amounts of parallelism and maintaining performance in a dynamic, heterogeneous environment?

In light of these issues, we must clearly identify where current programming technology is lacking, what new capabilities are required and whether they are best implemented at the language level, at the tool level, or in the run-time system. Hence, this paper endeavors to *identify and investigate programming methodologies that support the effective development of algorithms and codes that perform well in grid environments.* The term "programming methodology" is used here since we are not just considering programming languages. A programming methodology or model can be present in many different forms, e.g., a language, a library API, or a tool with extensible functionality. "Effective development" means that any such methodology should facilitate the entire software lifecycle: design, implementation, debugging, operation, maintainence, etc. Hence, successful programming methodologies should facilitate the effective use of all manner of tools, e.g., compilers, debuggers, performance monitors, etc. "Perform well" is intended to have the broadest possible interpretation. In a grid, "perform well" can mean not only the effective use (high utilization) of high-performance resources, but also the flexible composition and management of resources.

First, we list desirable properties for grid programming. Next, we review programming models fundamentals for parallel and distributed computing prior to discussing issues for grid programming. With these issues in mind, we then survey and evaluate existing programming tools in Section 4 and discuss focus areas for further research and development in Section 5. Conclusions are given in Section 6.

3. Properties for Grid Programming Models

There are several general properties that are desirable for all programming models. Properties for parallel programming models have also been discussed [64]. Grid programming models inherit all of these. The grid environment, however, will shift the emphasis on these properties dramatically to a degree not seen before.

- *Useability.* Grid programming tools should support a wide range of programming concepts and paradigms, e.g., simple desktop computing to large-scale, data-intensive computing, distributed client-server applications and "systems" code. There should be a low *barrier to acceptance*, i.e., new tools should not require

a drastically different approach to code building yet should offer a "development path" that will build more sophisticated concepts on previous successes.

- *Dynamic, Heterogeneous Configurations.* Grid applications will typically run in a heterogeneous environment that is changing, either due to competing resource demands or resource failures. A grid programmer should be able to ignore architectural/configuration details when possible yet control those same details when necessary. A grid program may want to change configuration based on available resources. This could entail process or data migration.

- *Portability.* In much the same way that current high-level languages allowed codes to be processor independent, grid programming models should allow grid codes greater software portability. Portability, or *architecture independence*, is a necessary prerequisite for coping with dynamic, heterogeneous configurations.

- *Interoperability.* The notion of an *open and extensible grid architecture* implies a distributed environment that may support protocols, services, application programming interface, and software development kits [34]. Any of these protocols, services, interfaces, or kits, may present a programming model that should be interoperable where appropriate.

- *Reliable performance.* Most grid users will want their applications to exhibit reliable performance behavior. Besides quality of service issues, a user should be able to know the "cost" of a programming construct on a given resource configuration. While some users may require an actual deterministic performance model, it may be more reasonable to expect reliable performance at least within some statistical bound. Besides providing reliable performance, grid programming tools should also promote *performance portability* as much as possible.

- *Reliability and Fault-tolerance.* Grid user must be able to check run-time faults of communication and/or computing resources and provide, at the program level, actions to recover or react to faults. At the same time, tools could assure a minimum level of reliable computation in the presence of faults implementing run-time mechanisms that add some form of reliability of operations.

- *Security and privacy.* Grid codes will commonly run across multiple adminstrative domains using shared resources such as networks. Hence, it is imperative that security and privacy be integral to grid programming models.

4. Programming Model Fundamentals

These desireable properties will be tempered by both current programming practices and the grid environment. The last twenty years of research and development in the areas of parallel and distributed programming and distributed system design has produced a body of knowledge that was driven by both the most feasible and effective hardware architectures and by the desire to be able to build systems that are more "well-behaved" with properties such as improved maintainability and reusability. Here we provide a brief survey of the fundamentals of programming models as they apply to parallel and distributed programming.

4.1 Access to State

Programming requires the algorithmic or systematic management of data which we are calling *access to state*. On a uniprocessor, it is typical and convenient to think of the data as being "in the same place" as the code such that an algorithm's or system's performance only depends on the structure of the algorithm or system, e.g., QuickSort having a complexity of *O(nlogn)*. Of course, multilevel caches and memory layout can have profound effects on performance, especially in scientific applications with very large data sets on vector machines.

In parallel and distributed grid environments, the issue of accessing state is even more complicated and affects an application's behavior to an even greater extent. Access to state is managed in two basic ways: *shared data abstractions* and *shared-nothing environments*. We also discuss *scoping* as the mechanism for managing the visibility of state.

- *Shared data space abstractions.* The most common shared data space abstraction is that of *shared-memory* actually supported in hardware. Of course, shared data spaces can be provided as an abstraction on top of distributed memory. A globally shared address space can be provided by *virtual* or *distributed shared memory (DSM)* systems. A globally shared name space can provided by languages such as High-Performance Fortran (HPF).

- *Shared Nothing.* Parallel computing can also be structured by assuming that nothing is shared but that a well-defined method for *copying* data between two or more disjoint processes exists. This is called *message-passing*. Typically there is a matched pair of processes called the *sender* and the *receiver* that exchange messages either synchronously or asynchronously, i.e., the sender or receiver blocks until a particular message is received or sent, respectively. The Message-Passing

Interface (MPI) is the most commonly used message-passing tool. One-sided message-passing is also possible where a message send operation does not have to be matched with an explicit receive operation, i.e., the receiver is always "listening" for incoming messages of any kind. Examples of this include *remote procedure calls (RPC), remote method invocations (RMI), remote service requests (RSR), and active messages (AM).*

- *Scope.* For both shared data and shared nothing abstractions, the concept of *scope* must be addressed. For a shared data abstraction, who can read or write which data? For a shared nothing environment, which receivers can a sender address messages to? Hence, scope is an important tool for structuring and controlling grid computations and services. Scope is typically associated with a *name space* that can be *local* (known to a few) or *global* (known to all).

4.2 Dependency Management

The *use* of data in a parallel environment must also be managed to enforce *dependencies*, i.e., when the data are valid and available, or when the storage space for the data is available. Enforcing dependencies requires some form of *synchronization* of which there are two major categories.

- *Control-oriented synchronization* is based on operations that explicitly manage the flow of control in a code, e.g., barriers, mutexs, condition variables, parblock, parfor, etc. All of these mechanisms have little to do with the application problem itself but are added to observe and enforce one or more, and typically a block of many, data dependencies. For example, one barrier may be used to enforce many loop-carried dependencies. We note that blocking message sends/receives is a form of control synchronization.

- *Data-oriented synchronization* is associated with actual data dependencies and typically results in finer grained synchronization with less unnecessary blocking. Data-oriented synchronization can be implemented as single-assignment variables or full/empty variables. We note that blocking on data to be received in a message is a form of data synchronization. Data-oriented synchronization is, of course, used in functional languages where scoping rules can eliminate side-effects and race conditions at the cost of high dynamic allocation and garbage collection overheads, high dynamic scheduling overheads, and poor data locality.

4.3 Process Management

In addition to managing access to data and the dependencies among them, the creation and termination of *processes* or *tasks* must also be managed. On a uniprocessor, this is done by the operating system and can be managed from a shell or under program control. This is also the case on a homogeneous parallel machine but richer control mechanisms must be available to create and terminate sets of related processes on parts or all of the machine. In a distributed environment, such as a grid, the issue is further complicated by heterogeneity. As with dependency management, there are two major categories for process management.

- *Task parallel – explicit management of parallelism.* Like control-oriented synchronization, task parallelism is based on explicit operations for managing the creation and termination of parallel threads of control, e.g., fork/join, parblock, RPC, etc. These threads of control can be implemented as heavyweight processes, lightweight threads, or even hardware threads.

- *Data parallel – implicit management of parallelism.* Data parallelism is associated with the application of a function to structured data, e.g, array operators, parallel objects, etc. We note that these *implicitly* manage the creation and termination of parallel threads of control.

4.4 Resource Management

The scope of resource management will be drastically different in a grid than any other single machine. We note that in a single machine, a process can be considered a resource, in addition to memory, disk, and other immediate hardware devices. In a grid, however, the scope of resources that are manageable from a single code explodes into an essentially open-ended environment. There are more machines, processes, storage devices, networks, services, and specialized instruments that are potentially discoverable and available. These resources will be heterogeneous and distributed over a wide area and will typically be used as a shared infrastructure.

Hence, grid programming models and tools will have to support not only the "traditional" operations of data access, synchronization, and process control, but also the *flexible composition of grid resources*. As we shall see, established programming tools are the least developed in this area. Not only must flexible composition be supported, but data access, synchronization and process control have to be supported across those compositions. While current programming tools, such as C and sockets, may be sufficient, in some sense, for building a grid infrastructure, it can hardly

be argued that these are the best tools for building grid applications.

5. Current Tools, Languages, and Environments

Having briefly surveyed the fundamentals of programming models, we now examine the many specific tools, languages and environments that have been built using these concepts. Many, if not most, of these systems have their roots in "ordinary" parallel or distributed computing and are being applied in grid environments, with varying degrees of success, because they are established programming methodologies. A few of these systems were actually developed concomitantly with the notion of grid computing and, in fact, may have helped define that notion. In this section, we focus on those programming tools that are actually available and in use today. We do, however, mention a few tools and languages that are not actually in use but nonetheless represent an important set of capabilities.

5.1 Shared Data Abstractions

5.1.1 Global Shared Address Space

Programming languages and tools that fall under this class provide a global shared memory model as a useful abstraction for state access, although on distributed computing architectures its implementation is distributed. This approach is called *virtual shared memory* or *distributed shared memory* (DSM). These programming languages present a view of memory as if it is shared, but the implementation may or may not be. The goal of such approaches is to emulate shared memory well enough that the same number of messages travel around the system when a program executes as would have traveled if the program had been written to pass messages explicitly. In other words, the emulation of shared memory imposes no extra message traffic.

Significant examples of languages and tools in this class are TreadMarks, Rthreads, IVY, Munin, and Linda. Some of them share similar features, others use different approaches in the implementation of global shared address space. Treadmarks [5] supports parallel computing on networks of computers by providing a global shared space across the different machines on a cluster. TreadMarks provides shared memory as a linear array of bytes via a relaxed memory model called *release consistency*. The implementation uses the virtual memory hardware to detect accesses, but it uses a multiple-writer protocol to alleviate problems caused by mismatches between page size and application granularity. Treadmarks provides facilities for process creation and destruction, synchronization, and shared memory allocation (for instance, by *Tmk_malloc()* is allocated shared memory). Systems such as IVY and Munin, use the same programming approach of TreadMarks, but they implement more strict consistency models.

Rather than providing the programmer with a shared memory space organized as a linear array of bytes, structured DSM systems offer a shared space of objects or tuples accessed by properly synchronized methods. Linda [18] is a well-known language that uses this approach. It is based on an associative memory abstraction called *tuple space*. Linda threads communicate with each other only by placing tuples in and removing tuples from this shared associative memory. In Linda, tuple space is accessed by four actions: one to place a tuple in tuple space, *out(T)*, two to remove a tuple from tuple space, *in(T)* and *rd(T)*, one by copying and the other destructively, and one which evaluates its components before storing the results in tuple space (allowing the creation of new processes), *eval(T)*. When invoked, the *eval* operation creates a process to evaluate the components of the tuple that it takes as argument. The efficient implementation of tuple space depends on distinguishing tuples by size and component types at compile time, and compiling them to message passing whenever the source and destination can be uniquely identified, and to hash tables when they cannot. In distributed-memory implementations, the use of two messages per tuple space access is claimed, which is an acceptable overhead.

Other models that are based on global shared address space are based on threads, such as Pthreads (POSIX threads) [57] that define low-level primitives to control access to shared address space or Java threads, implemented by the *Thread* class. An implementation of Pthreads on shared distributed memory systems is Remote threads (*Rthreads*) [24]. Rthreads supports sharing of global variables according to the Pthreads semantics on distributed memory architectures, also composed of heterogeneous nodes.

The use of global shared address space models is simple and direct because they offer a global view of grid memory resources. Thus their usage may result in significant programming benefits. The real question in programming grid applications with these models is performance, that is how efficient will be the implementation of DSM in a grid framework. This strictly depends on the implementation of run-time systems for programming languages and tools in this heterogeneous setting.

5.1.2 Global Data Space

Programming languages of this category allow users to define variables that are automatically mapped into the memory of processing elements that at higher level are seen as a global memory space. When data are mapped to their processing elements, program constructs can be used to express

parallel operations. Data that must be processed in parallel are defined using specific keywords. Then the compiler will partition the data and map them onto the different processors of the parallel computer so instructions that operate on these data will be executed in parallel on different processors that execute the same operation on different elements.

Languages and tool kits that implement a global data space are HPF, OpenMP, and C*. High Performance Fortran or HPF is a language for programming computationally intensive scientific applications [53]. A programmer writes the program in HPF using the SPMD style and provides information about desired data locality or distribution by annotating the code with data-mapping directives. An HPF program is compiled by an architecture-specific compiler. The compiler generates the appropriate code optimized for the selected architecture. According to this approach, HPF could be used also on shared-memory parallel computers. HPF must be considered as a high level parallel language because the programmer does not need to explicitly specify parallelism and process-to process communication. The HPF compiler must be able to identify code that can be executed in parallel and it implements inter-process communication. So HPF offers a higher programming level with respect several tool kits. On the other hand, HPF does not allow the exploitation of control parallelism and in some cases (e.g., irregular computations) the compiler is not able to identify all the parallelism that can be exploited in a parallel program, and thus it does not generate efficient code for parallel architectures.

OpenMP [58] is a library (application program interface or API) that supports parallel programming in shared memory parallel computers using the global data space approach. OpenMP has been developed by a consortium of vendors of parallel computers (DEC, HP, SGI, Sun, Intel, etc.) with the aim to have a standard programming interface for parallel shared-memory machines (like PVM and MPI for distributed memory machines). The OpenMP functions can be used inside Fortran, C and C++ programs. They allow for the parallel execution of code (*parallel DO loop*), the definition of shared data (*SHARED*), and synchronization of processes.

A standard OpenMP program begins its execution as a single task, when a *PARALLEL* construct is encountered, a set of processes are spawned to execute the corresponding parallel region of code. Each process is assigned with an iteration. When the execution of a parallel region ends, the results are used to update the data of the original process, which then resume its execution. From this operational way, could be deduced that support for general task parallelism is not included in the OpenMP specification. Moreover, constructs or directives for data distribution control are absent from the current releases of OpenMP.

The implementation of these programming models on grids is very hard for several reasons mainly related to the different computational model between global data space computing and grid. Here we list three of these reasons: (1) the programming models in this class abstract from several implementation issues that arise in a grid computing architecture, (2) generally they impose a tight synchronization model among parallel activities that can not be efficiently implemented on remote heterogeneous machines, and, (3) most of the applications developed with these models are based on regular patterns of computation and communication that do not match well the irregularities of grids.

5.1.3 Distributed Data Space

In this approach data shared among processes that compose a parallel or distributed application can be distributed among the memories of processors, but there is no a single global scope. Languages and tools in this class provide mechanisms for sharing data that are physically distributed on different processing elements by abstract shared spaces that can be accessed by groups of processes. Extensions to Linda, such as ISETL-Linda, SDL and Ease [76], are based on distributed data space abstraction for solving problems of a global single memory space. The first problem is that a single, shared, associative memory does not provide any way to structure the processes that use it, so that Linda programs have no natural higher-level structure. The second issues is that, as programs get larger, the lack of scoping in tuple space makes the optimizations of tuple space access described above less and less efficient. For example, two sets of communications in different parts of a program may, by coincidence, use tuples with the same type signature. They will tend to be implemented in the same hash table and their accesses will interfere. Thus, when a Linda program get larger or it is composed of several processes the management of a large tuple space and of the parallel processes becomes difficult.

Other languages based on the distributed data space abstraction are Opus, Orca and Emerald. In particular, Orca and Emerald are two object-based languages that implement data sharing by means of data managers and data replication mechanisms. For example, Orca defines a set of constructs for the sharing of data among processes on different processors [61]. Data are encapsulated in data-objects that are instances of user-defined data types. The Orca language is based on a hierarchically structured set of abstractions. At the lowest level, *reliable broadcast* is the basic primitive so that writes to a replicated structure can rapidly take effect throughout a system. At the next level of abstraction, shared data are encapsulated in passive objects that are replicated throughout the system. Orca itself provides an object-based language to create and manage objects. Rather than a strict coherence, Orca provides serializability: if several opera-

tions execute concurrently on an object, they affect the object as if they were executed serially in some order.

Systems with a single global space are more appropriate for grid environments with regards to models and languages discussed in Section 4.1.1. These models can be used to program grid applications as composed of groups or clusters of processes that share bunches of data each one composing a different data scope. At the same time, these models can support shared data replications and hierarchies that can match computing and memory hierarchies that exist in a grid.

5.1.4 Evaluation

- *Usability:* Programming languages and tools based on shared data abstractions can be effectively used for a wide range of grid applications and they offer programming approaches that do not drastically differ from well know programming models. However, it is worth to mention that these approaches are not particularly suited for massively parallel applications on grids, especially when irregular communication and computation structures are used.

- *Dynamic, Heterogeneous Configuration:* Among the shared data abstraction models, global shared address space models and distributed data space models can deal efficiently with dynamic and heterogeneous settings. These models offer the programmer abstractions that allow she/him to ignore architecture details. On the other hand, global data space tools, like HPF and OpenMP, may suffer from configuration changes and heterogeneous resources usage.

- *Portability:* The languages and tools based on shared data offer a programming approach that is architecture independent hence can offer good portability of code, whereas performance portability is not assured because of their high level of abstraction.

- *Interoperability:* Interoperability is an open question in this framework. It needs to be provided but at the moment no high-level protocols are provided for this class of languages. This issue can be effectively addressed if implementations of these languages will be based on protocols used for computational grids.

- *Reliable Performance:* As mentioned above, performance portability is not assured for shared data abstraction models. The programming models they implement is high level, so complex measures must be provided to compute performance costs on different architectures. Significant efforts in this area must be performed to define cost models in grid environments.

- *Fault Tolerance:* Some models in this class provide mechanisms to detect faults and to recover. Other models offer a programmer techniques that can be used to implement fault tolerance of grid applications via replication of data and processes or via reliable communication mechanisms such as Orca.

- *Security and Privacy:* Up to now, this issue has not explicitly addressed in parallel and concurrent shared data languages. Thus it must be investigated and new mechanisms must be added to the language constructs and operations to provide secure data sharing, cooperation and computations as other programming tool kits, such as MPICH-G2, do.

5.2 Shared Nothing – Message Passing

5.2.1 Two-Sided Communication

As previously stated, one method to manage access to state is the *shared nothing environment*. In this model processes run in disjoint address spaces and information is exchanged using message passing of one form or another. The Message Passing Interface (MPI) [54] standard defines a two-sided message passing library (matched sends and receives) that is well-suited for *shared nothing environments*. While the explicit parallelization with message passing is cumbersome it gives the user full control and is thus applicable to problems where more convenient semi-automatic programming models may fail. It also tells the programmer exactly where a potential expensive communication takes place. These two points do not hold only for single parallel machines, but even more for grid computing.

Some MPI applications are better-suited to run in a grid environment than others. One important class of problems is those that are *distributed by nature*, that is, problems whose solutions are inherently distributed. One example are remote visualization applications in which computationally intensive work producing visualization output is performed at one location, perhaps as an MPI application running on some massively parallel processor (MPP), and the images are displayed on a remote high-end (e.g., IDesk, CAVE) device. For such problems, all implementations cited below allow you to use MPI as your programming model.

A second class of problems is those that are *distributed by design*. Here you may have access to multiple computers and a problem that is too big for any one of those computers. These computers, perhaps at multiple sites connected across a WAN, may be coupled to form a computational grid.

There are many implementations of the MPI standard including vendor-supplied implementations optimized for specific platforms. If you want to run your applications distributed on different platforms an MPI implementation is

required that takes the heterogeneity of the network into account. A *grid-enabled* library should also be integrated into a *grid computing environments* to take care of launching the application. Problems to address are authentication, authorization and resource reservation across multiple computers on possibly different administrative domains. In the last couple of years several groups have been working on these *grid-enabled* MPI implementations [28, 32, 35, 42, 48].

MPICH-G2 [32] alleviates the user from the cumbersome (and often undesirable) task of learning and explicitly following site-specific details by enabling the user to launch a multimachine application with the use of a single command, *mpirun*. MPICH-G2 requires, however, that Globus services be available on all participating computers to contact each remote machine, authenticate the user on each, and initiate execution (e.g., fork, place into queues, etc.).

PACX-MPI [35] is an implementation that covers the complete MPI-1 standard with some support of MPI-2 and the MPI-2 Journal of Development. It contains a wide range of optimizations for grid environments, e.g. for efficient collective communications or derived datatypes. It supports TCP, ATM and an SSL based protocol for intermachine messaging.

Stampi [42] implements most of MPI-1. It includes support for MPI-IO and dynamic process management of MPI-2. It also offers the possibility to have a flexible number of router processes for inter machine messaging.

MPI_Connect [28] is designed to support applications running on vendor MPI implementations. By adding MPI intercommunicators it allows to link applications running on different machines. Unlike other grid-enabled MPI libraries each system maintains its own `MPI_COMM_WORLD` making it suitable for the *distributed by nature* applications described above.

MagPIe [48] is not a stand-alone library, but a tool to optimize an existing MPI library for grid environments. It uses the MPI profiling interface to replace the collective communication calls with optimized routines. Message delivery is done with the existing grid-enabled MPI.

LAM is an MPI implementation that has become grid aware with the support of IMPI [65].

Once the application is running, MPI automatically converts data in messages sent between machines of different architectures and supports multiprotocol communication by automatically selecting TCP for intermachine messaging and (where available) vendor-supplied MPI for intramachine messaging.

While MPI addresses some of the challenges in grid computing, it has not addressed them all. Some issues (e.g., algorithm design, communication patterns) can only be addressed by the *MPI application developer*. Local- and wide-area networks inject significantly higher latencies and lower bandwidths, and therefore MPI applications that expect to run efficiently in grid environments must be written with respect to this disparity in communication paths. MPI has the advantage that it offers an incremental approach to grid programming. It is possible to first gain some experiences and adopt and improve an existing solution to the grid. It also precludes the user from learning a new method for interoperating.

To allow an dynamic adoption to grid environments the applications might need more informations than provided by MPI. One example is the exact distribution of MPI processes on multiple machines. This information can be retrieved by the so called *cluster attributes* that are contained in the MPI Journal of Development [56] and available in some implementations [35].

5.2.2 One-sided Communication

While having matched send/receive pairs is a natural concept, *one-sided communication* is also possible and included in MPI-2 [55]. In this case, a *send* operation does not have to have an explicit *receive* operation. Not having to match sends and receives means that irregular and asynchronous communication patterns can be easily accommodated. To implement one-sided communication, however, means that there is usually an *implicit* outstanding receive operation that *listens* for any incoming messages, since there are no remote memory operations between multiple computers. However, the one-sided communication semantics as defined by MPI-2 can be implemented on top of two-sided communications [13].

A number of one-sided communication tools exist. One that supports multi-protocol communication suitable for grid environments is Nexus [31]. In Nexus terminology, a *remote service request (RSR)* is passed between *contexts*. Nexus has been used to built run-time support for languages to support parallel and distributed programming, such as Compositional C++ [21], and also MPI. We note that *remote procedure call (RPC)* or *remote method invocation (RMI)* are forms of one-sided communication. These are discussed later.

5.2.3 Evaluation

- *Usability:* Message-passing is inherently easy to understand and, hence, to use. However, it is constrained by this very simplicity to exchanging buffers of data that the application must explicitly interpret and the

method of interaction must be explicitly managed at the send-receive level.

- *Dynamic, Heterogeneous Configuration:* Most message-passing systems do not support dynamic, heterogeneous configurations. As in the case of MPI-1, the communication "universe" is finite and fixed at start-time. MPI-2 adds dynamic process creation, but this is only supported to a limited extend by the grid-enabled implementations.

- *Portability:* Since message-passing models hide processor and OS details, portability is typically quite good. However, the MPI standard allows a number of implementation defined behaviors which limits the portability if used carelessly.

- *Interoperability:* Interoperability in message-passing models is typically not supported since most models assume a "closed" communication world that observes a single communication mechanism. Interoperability should be possible, however, assuming that a common "wire protocol" is used. IMPI [22] is such a protocol, but it does neither cover the complete MPI-1 standard nor address MPI-2.

- *Reliable Performance:* The simplicity of the message-passing models means that the performance on the end-hosts is usually quite reliable. Congestion and bottlenecks in the communication medium, however, can drastically alter performance. Therefore QoS mechanisms have been integrated into some MPI libraries [35, 59]. The major problem here is that there is no commonly deployed QoS standard on the network level.

- *Fault Tolerance:* Message-passing models were originally designed for single-chassis parallel machines with lower fault rates and not grid environments. Hence, fault tolerance in message-passing models typically means just checkpointing on the part of the application. Introducing Fault Tolerance into MPI is ongoing research [27]. Even if it is supported by the MPI library it will still be the responsibility of the application to recover from failure.

- *Security and Privacy:* Security and privacy usually depend on the use of authentication and encryption outside of the message-passing model.

5.3 Object-Oriented Tools

The object concept is fundamental and has been used extensively to structure codes and systems. Accordingly a number of object-oriented parallel programming languages and tools have been designed [75]. We will briefly discuss two primary approaches to the use of objects: (1) "global" objects that can be referenced from anywhere, and (2) distributed objects that encapsulate a distributed internal structure.

5.3.1 Global Objects

CORBA. The Common Object Request Broker Architecture (CORBA) [69] uses a meta-language interface to manage interoperability among objects. Object member access is defined using the Interface Definition Language (IDL). An Object Request Broker (ORB) is used to provide resource discovery among client objects.

While CORBA can be considered middleware, its primary goal has been to manage interfaces between objects. As such, the primary focus has been on client-server interactions within a relatively static resource environment. With the emphasis on flexibly managing interfaces, implements tend to require layers of software on every function call resulting in performance degradation.

To enhance performance for those applications that require it, there is work being done on High-Performance CORBA [44]. This endeavors to improve the performance of CORBA not only by improving ORB performance, but by enabling "aggregate" processing in clusters or parallel machines. There are also efforts to make CORBA services available to grid computations. This is being done in the CoG Kit project [74] to enable "Commodity Grids" through an interface layer that maps Globus services to a CORBA API.

Legion. Legion [52] provides objects with a globally unique (and opaque) identifier. Using such an identifier, an object, and its members, can be referenced from anywhere. Being able to generate and dereference globally unique identifiers requires a significant distributed infrastructure. We note here that all Legion development is now being done as part of the AVAKI Corporation [7].

5.3.2 Distributed Objects

POOMA/SMARTS. POOMA (Parallel Object-Oriented Methods and Applications) [50] is an extended C++ class library for data-parallel numerical applications. Besides supporting data-parallel array operators, array indices can be arbitrary user-defined types and polymorphic. That is to say, indexing (mapping the domain to the range) can be done differently depending on the type of array being indexed. This allows arrays with different representations (e.g., column-major, row-major, sparse) to be transparently handled.

SMARTS (Shared Memory Asynchronous RunTime Systems) [73] is a run-time system developed to support

POOMA in a novel way. SMARTS provides a macro-dataflow approach based on the dependencies among the data-parallel operations. The granularity of computation is based on the dynamic decomposition of the data, rather than any fixed functional decomposition. SMARTS is implemented as a Dependency Graph Manager/Scheduler that controls the queues for multiple processors. While SMARTS was originally designed for shared-memory machines, work is being done to use it within a cluster of symmetric multiprocessors. The asynchronous scheduling in SMARTS will facilitate the overlapping of communication with computation and, hence, the tolerance of latency.

5.3.3 Evaluation

- *Usability:* The object-oriented paradigm is conceptually easy to use but the use of OO tools can become mired in the complexity of systems that must solve all problems. Nonetheless, objects provide a natural boundary to define their interfaces and behaviors.
- *Dynamic, Heterogeneous Configuration:* As exemplified by CORBA, the OO paradigm can be used to manage the interaction between brokers and applications such that dynamic configurations are possible.
- *Portability:* Object-oriented systems can provide excellent portability since encapsulation and the hiding of implementation details is fundamental to the OO paradigm.
- *Interoperability:* Interoperability must be addressed by common wire protocols such as IIOP in the CORBA world.
- *Reliable Performance:* In distributed OO systems, QoS mechanisms will eventually have to be integrated to provide reliable performance.
- *Fault Tolerance:* Fault tolerance must be achieved as in other environments by the use of techniques such as replication and transactions.
- *Security and Privacy:* Again, the OO paradigm provides a natural boundary on which to base security and privacy models. Authentication and encryption are integral to CORBA ORBs.

5.4 Middleware

While the tools developed in the last twenty years of parallel and distributed computing have become established, and will continue to play an important role in the development of grid software, the expanded resource richness of the grid means that *middleware* will emerge as an equally important class of tools. Middleware will typically be built on top of the established tools to provide additional useful abstractions to the grid programmer.

5.4.1 Network Enabled Servers and GridRPC

Overview of NES/GridRPC Systems

The Network-Enabled Server (NES) paradigm [20], which enables Grid-based RPC, or GridRPC for short, is a good candidate as a viable Grid middleware that offers a simple yet powerful programming paradigm for programming on the Grid. Several systems that facilitate whole or parts of the paradigm are already in existence, such as Neos [23], Netsolve [19], Nimrod/G [1], Ninf [62], and RCS [6], and we feel that pursuit of a common design in GridRPC, as had been done for MPI for message passing, will bring benefits of standardized programming model to the Grid world. This section will introduce the NES/Grid RPC features as an effective, simple-to use programming model and middleware for the Grid.

Compared to traditional RPC systems, such as CORBA, designed for applications that facilitate non-scientific applications, GridRPC systems offer features and capabilities that make it easy to program medium- to coarse-grained, task parallel applications that involve hundreds to thousands or more high-performance nodes, either concentrated as a tightly coupled cluster, or a set of them spread over a wide-area network. Such applications will often require handling of shipping megabytes of multi-dimensional array data in a user-transparent and efficient way, as well as requiring the support of RPC calls that range anywhere from 100s of milliseconds up to several days or even weeks. There are other necessary features of Grid RPC systems such as dynamic resource discovery, dynamic load balancing, fault tolerance, security (multi-site authentication, delegation of authentication, adapting to multiple security policies, etc.), easy-to-use client/server management, firewall and private address considerations, remote large file and I/O support etc. These features are essentially what is needed for the Grid RPC systems to execute well on the Grid—features either missing or incomplete in traditional 'closed world' RPC systems—and in fact are what are provided by lower level Grid substrates such as Condor, Globus, and Legion. As such GridRPC systems either provide these features themselves, or builds upon the features provided by such substrates.

In a sense, NES/GridRPC systems abstract away much of the Grid infrastructure and the associated complexities, allowing the users to program in a style he is accustomed to in order to exploit task-parallelism, i.e., asynchronous parallel procedure invocation where arguments and return values are passed by value or reference depending on his preference. Our studies, as well as user experiences, have shown that this paradigm is amenable to many large-scale

applications and especially to scientific simulations. The difference here is that 1) because of the 'open world' Grid assumptions the underlying GridRPC system must be much more robust, and 2) the scalability of the applications, in terms of the execution time, the parallelism, and the amount of data involved, must scale from just one node with a simple LAN RPC to thousand-node task parallel execution involving weeks and Terabytes of data.

Brief Sketch of Programming in NES/GridRPC

In this section, we take Ninf and Netsolve as an example NES/GridRPC system. Other systems such as Nimrod may have slightly different programming models and system features and will be covered in other sections of this document.

Both Ninf and Netsolve offer network-based numerical library functionality via the use of RPC technology. Parts of applications making procedure calls can be replaced with high-performance, remote equivalents in a substantially transparent manner, usually only a small modification to the call itself, without any RPC data types, prologue/epilogues, IDL management, etc.

Compared to simple, well-known RPC techniques such as CORBA, this is not simple as it seems: for example, for local procedure calls most numerical data structures such as arrays are passed by reference, so the numerical RPC mechanism must provide some shared memory view of the arguments, despite being a remote call. Furthermore, no information regarding its physical size, or which portions are to be used are maintained by the underlying language. Rather, such information are passed as parameters, which must be appropriately given by some calling convention which must be obeyed by the application. Also, the underlying language might not provide sufficient type information to the RPC system, as the types in the numerical RPC are somewhat 'finer grained', in that it must contain info such as the leading dimension of the array/matrix usage. Moreover, both Ninf and Netsolve locate appropriate computing resources on the network without explicit specification, achieving load balancing and fault tolerancy. Various information is contained in their specialized IDL language.

For example, in Ninf, if the original program, this case in C, involved calling a matrix multiply routine:

```
#define N 1000
double A[N][N], B[N][N], C[N][N];

matrix_mult(N, A, B, C); // C = A * B
```

The procedure call in the last line is merely replaced with:

```
Ninf_call("matrix_mult", N, A, B, C);
```

Here, notice that:

1. Only one generic, polymorphic call (`Ninf_call()`) given as the basic API for RPC. This not only allows simple one-line substitution of call sites to make the call be remote, but also allows the clients *not* to maintain stub libraries or IDLs (see below).

2. the size of the matrix is given by `N`, but C (nor Fortran) has no inherent ways to determine the size of the data structure, let alone which parts of the matrix is used for more intricate algorithms.

3. Both C and Fortran assumes that array arguments `A`, `B`, and `C` are passed by reference. Moreover, there is no way to tell from the program that `C` is being used as an output, while `A` and `B` are inputs to the procedure.

4. The first argument of the call is the signature of the library or an application being called. For Ninf, unless a URL is specified to target a particular library on a host, the metaserver infrastructure selects an appropriate server from a set of servers under its maintenance, depending on monitored network throughput, server performance and load, and the availability of the library.

In order to facilitate such flexibility for program globalization and ease-of-maintenance on the client side, both Ninf and Netsolve provide specialized IDLs that embody sufficient information to implement the features described above. For example, for Ninf, the IDL for the matrix_mult would look like the following:

```
Define dmmul( long mode_in int n,
              mode_in double A[n][n],
              mode_in double B[n][n],
              mode_out double C[n][n]
            )  /* Ninf interface */
"description"
Required "libXXX.o"   /* link libs */
/* lang. and call seq. */
Calls "C" dmmul(n,A,B,C);
...
```

Here, the IDL embodies the necessary information to describe the in/out values of the call, just as is with the CORBA IDL. Some additional information are present, such as the computational complexity of the call with respect to its arguments. Moreover, the IDL compiler automatically deduces the dependencies of scalar parameters versus the array indices. The example IDL describes only the simple situation of shipping the entire matrix based on n; more complex descriptions such as leading dimension, stride, dependencies on linear combinations of multiple indices, etc., are supported.

In both Ninf and Netsolve, the client does not maintain *any* form of IDLs; rather, the client only contains a small

IDL interpreter. When a call is made, the metaserver (Agent in the case of Netsolve) locates an appropriate server, and lets it connect to the client. Given the signature of the call, the server sends the compiled IDL to the client; the client IDL interpreter in turn uses it to marshal and demarshal the arguments in order to make the call. This is somewhat similar to the Dynamic Invocation Interface (DII) of CORBA, but is completely transparent on the client side, unlike DII.

In summary Ninf and Netsolve are *asymmetrical* systems in that the client and server side software packages being different. Not only this is the matter of size, but also for functionality, as well as the necessary of maintenance; the client side need not be updated to a limited degree even if there is some protocol change, thanks to the IDL info being uploaded dynamically. By contrast, CORBA and other typical RPC systems are *symmetrical*, in that the same software packages are used for both client and the server. While this provides better flexibility for example clients can become servers and vice versa, it could put the complexity of the management not only on the server but also on the client side.

5.4.2 Frameworks

Cactus. The Cactus Code and Computational Toolkit [17] is the cumulation of over 10 years of development by many computer scientists and physicists to provide computational physicists with a flexible, modular, portable and importantly easy-to-use, programming environment for large-scale simulations.

One of the design requirements for Cactus was to provide application programmers with a high level set of APIs which hide features such as the underlying communication and data layers. These layers are implemented in modules (in Cactus terminology *thorns*), which can be chosen at runtime, using the best available technology for a given resource, e.g., MPI, PVM, pThreads, SHMEM, OpenMP for communication, or HDF5, IEEEIO, PandaIO for parallel data I/O. For example, to output a grid variable from a Cactus application, programmers use the API call `CCTK_OutputVar(grid_identifier, grid_variable)`, depending on the modules available for I/O, the grid variable could be written to a local file-system, to a remote file-system using DPSS, or even written to a virtual file which can then be streamed across the network to any other resource.

Much of design of the Cactus architecture is influenced by the vast computing requirements of some of the main applications using the framework, including numerical relativity and astrophysics. These applications, which are being developed and run by large international collaborations, require Terabyte and Teraflop resources, and will provide an ideal test-case for developing Grid computing technologies for simulation applications.

Features of Cactus which are relevant for Grid programming [4] include:

(i) automatic and configurable compilation system for most machine architectures,

(ii) core code and toolkits written in ANSI C for portability,

(iii) parallel I/O capabilities compatible with distributed simulations,

(iv) parallel checkpointing and recovery of simulations, including distributed simulations,

(v) steering interface for dynamically changing the values of parameters during a simulation,

(vi) existing applications already trivially Grid-enabled using the Globus MPI implementation MPICH-G2, and,

(vii) existing modules to implement remote visualization (streaming data with HDF5), remote monitoring and steering of simulations (*e.g.* using a module which provides a simulation with its own web server), parallel I/O, etc.

There are several ongoing Grid related projects associated with Cactus, either to develop Cactus Grid-infrastructure and tools, or to build more generic Grid tools using Cactus as an application. These projects include the development of a web-based portal for remotely compilation, staging and control, developing the communication layer for more efficient distributed simulations, enhancing parallel and distributed I/O capabilities and the construction of a *Grid Application Toolkit* to allow application and infrastructure to easily incorporate grid programming capabilities in a generic manner.

Meta-Chaos. The ability to compose multiple separately developed parallel applications is becoming increasingly important in many applications, in areas as diverse as multidisciplinary complex physical simulations and medical image database applications. Meta-Chaos is a prototype "meta-library" developed at the University of Maryland that makes it possible to integrate multiple data parallel programs (perhaps written using different parallel programming paradigms) within a single application [25, 60]. Meta-Chaos also supports the integration of multiple data parallel libraries within a single program. In effect, Meta-Chaos provides a Unix-style pipe for parallel programs. Applications that have been developed with Meta-Chaos include coupling multiple scientific simulations, potentially running at different sites across a wide-area network, and integrating results from multiple remote sensor or medical image databases.

In collaborative projects with groups at several universities, Meta-Chaos has been used to exchange data between data parallel programs written using High Performance Fortran (HPF), the Maryland CHAOS and Multiblock Parti libraries, and the UC San Diego KeLP library [29]. These libraries are targeted at parallelization of applications that work on unstructured (CHAOS) and multiple structured grids (Multiblock Parti, KeLP). In addition several high-end sensor and scientific database applications implemented with the Maryland Active Data Repository (ADR) object-oriented framework have used Meta-Chaos to exchange data between a parallel ADR database and a parallel client application.

Meta-Chaos implements a framework-based solution to the interoperability problem, requiring that each data parallel library export a set of interface functions, and uses those functions to allow all the libraries to interoperate. This approach gives the task of providing the required interface functions to the data parallel library developer (or a third party application developer that wants to be able to exchange data with the library). The interface functions provide information that allows the meta-library to inquire about the location (processor and local address) of data distributed by a given data parallel library.

DataCutter. DataCutter [12, 11] is an application framework, under development at University of Maryland, that provides support for developing data-intensive applications that make use of scientific datasets in remote/archival storage systems across a wide-area network. To make efficient use of distributed shared resources, the application processing structure is implemented as a set of distributed processes, called *filters*, that adhere to the *filter-stream programming* model. DataCutter uses these distributed processes to carry out a rich set of queries and application specific data transformations. Filters can execute anywhere (e.g., on computational farms), but are intended to be run on machines where the location provides an efficiency advantage. For example, given a filter that greatly reduces the size of the data it receives before sending it to the next filter, an efficient location would be close to the archival storage server. DataCutter also provides support for subsetting very large datasets through multi-dimensional range queries. It uses a multi-level hierarchical indexing scheme, based on R-tree indexing methods, to ensure scalability to very large datasets.

The basic ideas underlying the filter-stream model [12] are to (1) constrain application components to allow for location independence, which is necessary for execution in a distributed environment, and (2) expose application communication patterns and resource requirements, allowing a runtime system to aid in efficient execution. The programming model is loosely based on the *stream-based programming model*, originally developed for Active Disks [2].

In the filter-stream programming model, part of an application is represented by a collection of *filters*. A filter is a portion of the full application that performs some discrete function. Filters can pre-disclose dynamic memory and scratch space needs so that the required space can be allocated by the underlying runtime system on behalf of the filter. Communication with other filters is solely through the use of *streams*. A stream is a communication abstraction that allows fixed sized untyped data buffers to be transported from one filter to another. A simple example of this model is Unix system pipes, where the standard output of a process is used as standard input for another process. Unix pipes represent a linear chain of filters, each of which have a single input stream and a single output stream. The filter-stream model allows for arbitrary graphs of filters with any number of input and output streams.

The process of manually restructuring an application using this model is referred to as *decomposing* the application. The main goal in choosing the appropriate decomposition is to achieve efficient use of limited resources in a distributed and heterogeneous environment. A particular granularity of the application decomposition into filters is not mandated by the model. Given a set of filters, the runtime mapping of filters onto various hosts in a wide-area grid environment is referred to as *placement*. The choice of placement represents the main degree of freedom in affecting application performance by, for instance, placing filters with affinity to data sources near the sources, minimizing communication volume on slow links, placing computationally intensive filters on less loaded hosts, etc. Note that a placement decision is not assumed to be static, and the programming model supports the notion of stopping a set of filters and replacing them with possibly a new set of filters with a different placement. A runtime system infrastructure, called the DataCutter Filtering Service, provides support for the execution of applications that are structured in the filter-stream programming model.

The lifetime of a filter is defined by the amount of work required by the application. Within this lifetime, a filter can process multiple logically distinct portions of the total workload. This is referred to as a *unit-of-work*, and provides an explicit time when adaptation decisions may be made while an application is running. A unit-of-work starts with the submission of a work description to a running set of filters, and ends when the last filter finishes processing the work. A collection of running filters that operate collectively to process a unit-of-work is referred to as a *filter instance*. An application may have multiple concurrent filter instances.

5.4.3 Component Architectures

Components extend the object-oriented paradigm by enabling objects to manage the interfaces they present and discover those presented by others [66]. This also allows implementation to be completely separated from definition and version. Components are required to have a set of *well-known ports* that includes an *inspection* port. This allows one component to query another and discover what interfaces are supported and their exact specifications. This capability means that a component must be able to provide metadata about its interfaces and also perhaps about its functional and performance properties. This capability also supports software reuse and composibility.

A number of component and component-like systems have been defined. These include COM/DCOM [63], the CORBA 3 Component Model [69], Enterprise Java Beans and Jini [26, 68], and the Common Component Architecture [36]. Of these, the Common Component Architecture includes specific features for high-performance computing, such as *collective ports* and *direct connections*.

5.4.4 Evaluation

- *Usability:* By design, middleware systems provide a simple programming model to allow the user to migrate existing code to the Grid. For example, experiences in both Ninf and Netsolve have shown that, for simple parallelizations such as parameter sweep, users can parallelize their code and effectively "gridify" it in matters of hours. Some systems restrict the scope of parallelization to task-parallel programming models. Mixed programming models, such as OpenMP and MPI, are also not supported very well.

- *Dynamic Heterogeneous Configuration:* Middleware systems offer varying degrees of resource discovery, monitoring, naming(directory), maintenance, and scheduling features, relieving the users of at least some of the burden of explicit resource specifications. Since middleware systems are intended to hide low-level details, they almost always support heterogeneous resources.

- *Portability:* Again, since middleware systems are intended to hide details, they are independent of specific processors or OSs, and, hence, are portable. Moreover, most systems offer client and server bindings for various programming languages, such as C/C++, Fortran, Java, Lisp, as well as interfaces to tools and components such as Matlab, Mathematica, and COM components.

- *Interoperability:* Interoperability of middleware systems is a problematic issue. Assuming for the moment that two middleware systems understand the same semantics, then interoperability would only require a commonly understood communication protocol. However, since most middleware systems are intended to support complex, application-domain-specific semantics, interoperability may be meaningless. Interoperability might be reasonable, though, for simple, common operations (such as passing an array of floats) or where the semantics of two systems overlaps significantly (e.g., between Ninf and NetSolve).

- *Reliable Performance:* General performance prediction or maintenance of middleware system performance is still difficult due to the dynamic nature of the Grid. Since QoS support is not commonly available yet, the best that can be done is to estimate performance. Some NES/GridRPC systems, for example, offer entries in their IDLs to specify the complexity of the computation being remotely invoked to assist the scheduler in making more effective scheduling decisions. Various work is underway to determine the performance characteristics of the Grid. For NES/GridRPC systems, see [3] for example.

- *Fault Tolerance:* At the middleware level, fault tolerance may depend on both implementation and inherent design properties. NES/GridRPC systems can support transactional behavior that provides better fault recovery than simply skipping servers that are down and calling another.

- *Security and Privacy:* Middleware systems can support a variety of security mechanisms. This can be as simple as no authentication, the use of passwords and secure shells, or as complicated as using X.509 certificates. Of course, an advantage of middleware is that they can hide the details of a security mechanism beneath their APIs.

5.5 Grid Computing Environments

5.5.1 Problem Solving Environments

Problem Solving Environments (PSEs) form another class of higher-level computing environments. The basic notion is that a PSE is comprised of a number of modular functions that can be composed into a more complex, composite application. Each of these modules or functions provides a service and hides low-level details involved in grid use. While some of these modules provide basic services, such as a communication "channel" or "pipe", others can be tailored to a particular style of programming or an application domain.

Since this definition of PSE is somewhat loose and high-level itself and, hence, can cover many different types of

systems. The terms PSE, workbench and framework are often used interchangeably. Some systems that can be considered PSEs have already been introduced. This includes Ninf/NetSolve, Cactus, and DataCutter.

The prime distinction, however, should be that such systems provide the abstractions and interfaces that serve a particular application problem domain. Jaco3, for example, provides an environment for coupling multiple physics simulation codes to address multi-physics problems [43]. The existing codes are given either a generic CORBA wrapper or an IDL interface. Parallel interfaces are possible. VisualORB provides a graphical interface for composing the modules. Another example is Nimrod/G [15]. It provides automated support for modeling and execution of parameter sweep applications. It uses a declarative parametric modeling language that enables the user to express a set of parametric computations which are then automatically scheduled and managed. Other PSEs include SCIRun for visualization [45] and VDCE for virtual computing [40].

A more detailed, template comparison of these and other PSEs is available as a white paper from the APM web site [16].

5.5.2 Portals

Portals can be viewed as providing a web-based interface to a distributed system. More precisely, though, portals entail a *three tier architecture* that consists of (1) a first tier of clients, (2) a middle tier brokers or servers, and (3) a third tier of object repositories, compute servers, databases, or any other resource or service needed by the portal. Clients and middle-tier servers typically communicate via HTTP allowing any web browser to be used. Middle-tier servers can simply access local files to serve pages but also can dynamically generate web page content by running CGI scripts, and by directly or indirectly interacting with the back-end resources. The interaction with third-tier resources can be accomplished in any protocol or manner appropriate.

Using this general architecture, portals can be built that support a wide variety of application domains, e.g., science portals, compute portals, shopping portals, education portals, etc. To do this effectively, however, requires a set of portal building tools that can be customized for each application area. An example of this is (surprise, surprise) the Grid Portal Toolkit, aka GridPort [71]. The GridPort Toolkit is partitioned into two parts: (1) the client interface tools, and (2) the web portal services module. The client interface tools enable customized portal interface development and does not require users to have any specialized knowledge of the underlying portal technology. The web portal services module runs on commercial web servers and provides authenticated use of grid resources. Note that the portal interface and the portal services can be run on different servers which provides isolation between the front-end and back-end functions, facilitates distributed execution, and enhances security.

HotPage [72] is an example of a system built using GridPort. HotPage provides users with a view of distributed computing resources and allows individual machines to be examined as to status (up or down), load, etc. Besides examining machines, users can access files and perform routine computational tasks. Sessions are authenticated and encrypted to provide security.

Another very important project is the Indiana/NCSA Science Portal [37]. In this effort, portals are designed using a *notebook* of typical web pages, input forms, and execution scripts. Notebooks have an interactive script/forms editor based on JPython that allows access to other tool kits such as CoG Kit and the Common Component Architecture Toolkit (CCAT). The ability to manipulate components in the context of a portal interface and architecture is extremely important. As discussed early, components must be able to provide metadata about their interfaces and perhaps about their properties. This means that notebook scripts can dynamically compose and manage components in a grid environment. Proxy components can be used to encapsulate legacy applications and manage I/O staging. Notebook components have an integral event model enabling greater functionality and robustness. The coupling of portals and components will facilitate ease of use by the user and the dynamic composition of grid codes and services in a way that will provide the best of both worlds.

Other examples of grid computing portals are available from the Grid Computing Environment Working Group web site [39].

5.5.3 Evaluation

- *Usability:* PSEs and portals are easy to use by their design.
- *Dynamic, Heterogeneous Configuration:* Since PSEs and portals are intended to have integral “front-end” interfaces, the back-end systems can more easily manage dynamic configurations.
- *Portability:* For portals, portability can be considered separate for the front and back ends. The use of HTTP-based browsers on the front-end enables tremendous portability. On the back-end, portals and PSEs face the same portability issues. If the modules on which the back-ends are built hide processor and OS details, then portability is possible.
- *Interoperability:* A similar dichotomy exists for interoperability. The use of HTTP on the front-end leverages the tremendous interoperability of web browsers.

On the back-end, interoperability depends on the use of common wire protocols and common semantics.

- *Reliable Performance:* Aside from the issues of dealing with a dynamic grid environment, PSEs and portals need to provide a mechanism whereby the user or the client can determine when a computational request is going to exceed some reasonable bound. Their flexibility and ease of use mean that PSEs and portals can enable users to unwittingly "ask for too much" or ask for something that current conditions cannot support.

- *Fault Tolerance:* Fault tolerance needs to be addressed in PSEs and portals.

- *Security and Privacy:* Security and privacy should be provided by the underlying infrastructure and be integral to the model presented to the user as is the case with GridPort.

6. Grid Programming Issues

The systems surveyed in the previous section vary widely in scope and the capabilities they are able to provide in a grid environment. We now discuss the significant issues that set grid programming apart using a simple categorization. While this captures the major classes of issues, there are some topics that nevertheless cut across multiple categories.

For each of these issues, where possible, we suggest Focus Areas representing technologies that could be used to address them. We also pose the questions relevant to framing the discussion about an issue and the further research that needs to be done to resolve it. Across all focus areas, an inherent question is *where to address it?* In the hardware, operating system, run-time, language, tool/library/environment, or "in the application"? In the terminology of the proposed Grid Protocol Architecture, should an issue be addressed in a protocol, a service, an API, or a SDK? In some cases, it may be clear but, in others, different alternatives may be possible.

6.1 Performance Management

For any significant application, performance is always an issue. Grids present challenging performance issues due to their heterogeneous nature and open architecture.

6.1.1 Heterogeneous Bandwidth/Latency Hierarchy

Grids will present a hierarchy of bandwidths and latencies that is getting deeper and more heterogeneous. Simply put, *highly synchronous operations will not be desirable* since they will be too inefficient. This situation will not improve, especially in a wide-area grid, since simple propagation delays are coming to dominant end-to-end latencies [51]. This also means that the consistency of replicated information across multiple sites will be increasingly difficult to maintain. (Distributed shared-memory systems, for instance, will be unattractive across a wide-area grid.)

Hence, the central question here is *What distribution of latencies can be accommodated?* A host of well-known latency tolerance techniques can be applied but to what *degree* will they be effective? Aside from arguments of shorter turn-around time, will large computations always be married to tightly coupled hardware? We note that any given code represents an abstract problem architecture which can determine which platforms are appropriate performance-wise. Can the abstract problem be restructured to allow more loosely coupled codes to be constructed? Can execution models enable looser coupling?

Focus Area: Execution Models. Hiding latency with throughput is a well-known technique. In the Tera MT-1 machine and the HTMT, this is used to hide memory latency. A processor can switch between hardware threads depending on which thread has a memory reference that is completing. This same technique could be applied in a grid computation *but on a vastly different scale.* Hence, in a grid computation, can enough parallelism be extracted with a large enough granularity to permit a data-driven execution model? If so, can asynchronous models or styles of programming be developed?

6.1.2 Data and Resource Topology

Large scientific codes using data decomposition could benefit from programming tools that understand the *data topology* and how it is mapped on to the resource topology. Some work has been done in this area with regard to clusters of symmetric multiprocessors, or "clumps". In this environment, communication through shared-memory must be balanced with slower network communication. SIMPLE provides a set of collective communication operations for clumps [9]. The KeLP system, however, uses the notion of *structural abstraction* and an associated *region calculus* to manage message-passing, thread scheduling and synchronization in clumps [29, 8]. MagPIe provides collective operations for MPI for wide-area, two-tier clusters, i.e., clusters of clusters. [47]. As discussed earlier, POOMA/SMARTS uses macro-dataflow scheduling to manage large-grain data-parallel operations [50, 73].

Focus Area: Heterogeneous Topologies. Can these techniques be gainfully applied in wide-area, heterogeneous environments? How do the bandwidths and latencies encountered in a general grid change their effectiveness? How

would a distributed object encapsulate the knowledge and understanding necessary to efficiently manage parallel operations over a dynamic bandwidth and latency hierarchy?

6.1.3 Performance Reliability

The heterogeneous nature of grids means that it will be difficult for many applications to provide reliable performance. For those applications where this is important, there will have to be methods whereby sources of variability can be controlled, or at least monitored. *Quality of Service* (QoS) is the concept whereby resources deliver a "contracted" level of service. The broad definition of a "resource" in grid computing, however, means that QoS can also be broadly applied. Besides network bandwidth and latency, this can include cpu scheduling, memory/disk space, and access to remote services. Besides *capacity reservations*, QoS can also address *temporal reservations*, i.e., advance reservations for some time in the future.

We note that, in general, hard performance guarantees can be expensive to deploy and enforce. The Integrated Services Model, for example, can provide a hard bandwidth but requires per-flow state at every hop for network QoS. The Differentiated Services model, by contrast, aggregates flows into different classes of service at each hop (thereby not requiring per-flow state) but only provides a statistical bandwidth guarantee.

Focus Area: Quality of Service. Grid QoS is clearly desirable. In an open architecture, however, it can be difficult to enforce performance guarantees. For any resource, QoS implies that some controller or agent (centralized or distributed) knows how much resource capacity exists and how much has been allocated. In order for meaningful reservations to be made, (1) a controller must have a hard enforcement mechanism, or (2) all users must engage the controller and observe policies. If this is not the case, then there is nothing to prevent a rogue (or simply unaware) user from consuming excessive resource capacity.

Aside from these hard issues, work has been done in this area. The Globus Advance Reservation Architecture (GARA), for instance, provides a uniform interface for dealing with both capacity and temporal reservations for several resource types, including network bandwidth and fractional cpu scheduling [33].

6.2 Configuration Management

In addition to performance issues, the open and heterogeneous nature of grids also presents configuration issues, that is to say, the grid programmer needs to know (1) what configuration the grid is currently in and (2) what configuration the application could be in. Furthermore, how does a grid programmer or a grid code know how to interact with other codes, services or resources?

6.2.1 Program Resource Metadata

To address these issues, grids must deploy *information services* to enable *resource discovery*. This allows tremendous flexibility in *when*, *where*, and *how* a grid program executes and also enables it to monitor and probe its environment during execution. Most established parallel and distributed programming tools are only aware of resources that are statically identified at start-time. How should dynamic resource-awareness be integrated? How should the universe of grid resources be represented, discovered, and used in a grid application?

Focus Area: Metadata Schema. Any representation of program resources must be open and extensible. Hence, a *metadata schema* must be used which can be developed using tools such as XML. Several related schema have been produced for grids, such as those by the Grid Information Services WG and the Grid Performance WG. The Globus MDS also defines an information schema [30]. These schema have not, however, been developed with the goal of enabling dynamic program configuration management.

6.2.2 Component Software.

Components could be used to encapsulate and manage the interfaces to many grid functions. For example, components can encapsulate communication as well as computation. In other words, components can also be connectors between other components. These could possibly encapsulate data management to hide latency, support a stream-based execution model, or possibly other, non-functional behaviors, such as fault-tolerance or security. These capabilities, in addition to interface discovery, have clear utility for grid programming that have not yet been completely investigated.

Focus Area: A Grid Component Architecture. How should a component architecture be defined for the emerging grid architecture? Some work has been done in this regards [36] but complete implementations and in-depth experience have yet to be realized.

6.2.3 Global Name Spaces and Persistence

A global name space and persistent objects have definite uses. A global name space has many issues in common with systems such as DNS.

Focus Area: Implementation. Resolving global names requires a deployment on the scope of DNS and will share many of the implementation issues. Should a global name

server for grid environments be piggy-backed on DNS or is different system design more suitable?

6.3 Programming/User Environments

After dealing with the basic mechanisms of managing performance and configuration, higher-level models can be defined to facilitate the *process* of building grid applications.

6.3.1 Remote Data Access

One of the goals of grid computing is to enable routine access to large amounts of data that are often associated with *data-intensive* applications. Extremely large data sets (currently in the terabytes) are difficult to move or copy. Hence, tools are being deployed to allow remote codes to access, subset, filter, and replicate large data sets. This will allow users to extract the minimum amount of data required, specify operations to be done "in-transit", such as decimations or corner-turns, and store copies "closer to home". Naturally this raises issues of access policies, replication policies, consistency requirements, etc.

Focus Area: Data Grid Issues. These are specifically the issues of data grids. A number of such projects are underway [41, 67, 70]. Data grids may typically involve systems like the Storage Resource Broker (SRB) [10] that provide a uniform middleware API to access heterogeneous, distributed storage resources. With regard to programming issues, tools such as DataCutter [49] have been integrated into the SRB to provide a *filter-stream programming model*. Filters are sub-classed from a C++ filter base class. Streams are named and optionally associated with an XML DTD. The allocation of filters and streams on to grid resources is an important issue for data-intensive applications, especially given the bandwidth and latencies issues noted above.

6.3.2 Frameworks, PSEs and Portals

Frameworks, problem solving environments, and portals share similar issues. While frameworks, PSEs, and portals are intended to be tailored to application-specific domains, what is the most appropriate software architecture that will support this flexibility and not excessively hamper performance? In such systems, there could be an emergent dominant practice for their construction.

Focus Area: Portals and Components. Components provide a tremendously flexible and dynamic way of composing functionality. This is a fundamental aspect of the programming task. Portals provide ease of use through a graphical interface that is the most widely used and familiar interface across the spectrum of computer users. The proper integration of these capabilities will have far-reaching benefits.

6.3.3 Languages, Compilers and Run-Time Systems

Compilers with an associated run-time system can implement a wide variety of interesting semantics. The open question here is what kind of *grid semantics* need to be offered at the language level? As an example, Compositional C++ included the notion of a *processor object* with data and function members that could be referenced through *global pointers* [46]. It seems unlikely, however, that compiler support for loop-level parallelism, e.g., loop unrolling and code reorganization, will be useful. More likely is the use of simple pragma to give useful hints to the compiler and run-time for improved program behavior under different grid conditions.

6.4 Properties

6.4.1 Portability

Most established tools also assume a homogeneous environment. While homogeneous sets of resources can be acquired in a grid environment, being able to use heterogeneous resources allows greater flexibility. How can heterogeneity be facilitated without having to simply pre-compile and pre-stage binaries, data, etc., or resorting to a single-language design such as Java?

6.4.2 Interoperability

Interoperability is a necessary prerequisite for grids to become established technology. If different implementations of the same services can interoperate, then there is no requirement to have exactly the same version of the grid infrastructure deployed everywhere. CORBA, for example, uses IIOP for this purpose. Protocols will likewise play an important role in grid architectures. What grid protocols are needed to support programming models? Is SOAP (Simple Object Access Protocol) sufficient [14]? Is it useful to consider interoperability at higher levels, e.g., *framework interoperability*?

Focus Area: Multi-method Communication. While tools like SOAP can provide interoperability, the cost of their flexibility can be lower performance [38]. Can multi-method communication packages (as in the spirit of Nexus [31]) be designed and built such that flexibility is provided for *negotiating* communication and beyond a certain break-even point, communication is done using a more efficient method?

6.4.3 Security

The wide-spread use and connectivity of shared compute resources in society at large, i.e., the Internet, has made security a serious issue. How does this need for security manifest itself in grid programming models and tools? The models and tools examined previously suggest the following short list of security requirements:

- Running an individual task or invoking an individual method on a remote host can require mutual authentication and authorization of the requestor and remote host, and also privacy and integrity checking of any data sent to and received from the remote host.
- Interacting with a remote service carries all the same requirements.
- Co-scheduling a set of remote tasks, methods, or services can require a set of mutual authentications, etc., that could be related or separate.
- A grid application may incrementally acquire resources or may interact with resources through proxies such that "chains" of trust must be established.

The typical grid security infrastructure is based on the concept of a grid user having a "grid identity" that is associated with something like an X.509 certificate. The management of certificates and Certificate Authorities (that issue and verify certificates) is beyond the scope of this document.

Focus Area: Delegation of Trust Chains. What is at issue here is how such mechanisms are used in programming models and tools in the above situations. An outstanding issue in security is the delegation of trust, i.e., allowing a remote host or proxy to act on your behalf *with your identity* such that chains of trust are established. This has clear implications for iterative and recursive computing topologies that have not been investigated.

6.4.4 Reliability and Fault Tolerance

Reliability and fault tolerance in grid programming models/tools are largely unexplored. Certain application domains are more amendable to fault tolerance than other, e.g., parameter sweep or Monte Carlo simulations that are composed of many independent cases where a case can simply be redone if it fails for any reason. The issue here, however, is how to make grid programming models and tools inherently more reliable and fault tolerant. Clearly a distinction exists between reliability and fault tolerance in the application versus in the programming model/tool versus in the grid infrastructure itself. An argument can be made that reliability and fault tolerance have to be available at all lower levels to be possible at the higher levels. A further distinction can be made between fault detection, fault notification and fault recovery. In a distributed grid environment, simply being able to detect when a fault has occurred is crucial. Propagating notification of that fault to relevant sites is also critical. Finally these relevant sites must be able to take action to recover from or limit the effects of the fault.

Focus Area: Event Models. These capabilities require that *event models* be integral to grid programming models and tools. Event models are required for many aspects of grid computing, such as a performance monitoring infrastructure. Hence, it is expected that a widely deployed grid event mechanism will become available. The use of such a mechanism will be a key element for reliable and fault tolerant programming models.

7. Conclusions

We have taken a broad look at the issues concerning the effective development of efficient grid codes. Grid programming models and tools will dramatically shift the emphasis of desireable or necessary properties to those of managing heterogeneous configurations, interoperability, and reliable performance. In reviewing current programming tools, languages and environments, we note that most are limited with regards to resource management, i.e., being able to dynamically orchestrate arbitrary grid resources. We also discuss a number of specific focus areas for future work in grid programming, from low-level performance issues to enhanced support for high-level tools.

8. Security

A brief discussion of general security issues appears in Section 6.4.3.

9. Author Contact Information

Craig A. Lee
The Aerospace Corporation, M1-102
2350 E. El Segundo Blvd.
El Segundo, CA 90245
lee@aero.org

Satoshi Matsuoka
Tokyo Institute of Technology
matsu@is.titech.ac.jp

Domenico Talia
Istituto per la Sistemistica e l'Informatica/CNR
talia@deis.unical.it

Alan Sussman
University of Maryland
als@cs.umd.edu

Matthias Mueller
High Performance Computing Center, Stuttgart
mueller@hlrs.de

Gabrielle Allen
Max-Planck-Institut für Gravitationsphysik
allen@aei-potsdam.mpg.de

Joel Saltz
University of Maryland
saltz@cs.umd.edu

References

[1] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 520–528, 2000.

[2] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *ASPLOS98*, pages 81–91. ACM Press, Oct. 1998. ACM SIGPLAN Notices, Vol. 33, No. 11.

[3] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, et al. Performance evaluation model for scheduling in global computing systems. *The International Journal of High Performance Computing Applications*, 14(3):268–279, 2000.

[4] G. Allen et al. Cactus tools for grid applications. *Cluster Computing*, 2001.

[5] C. Amza et al. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[6] P. Arbenz, W. Gander, and M. Oettli. The remote computational system. *Parallel Computing*, 23(10):1421–1428, 1997.

[7] AVAKI. The AVAKI corporation. *http://www.avaki.com*, 2001.

[8] S. Baden and S. Fink. The Data Mover: A machine-independent abstraction for managing customized data motion. *LCPC*, August 1999.

[9] D. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors. Technical report, University of Maryland, Department of Computer Science, and The University of Maryland Institute for Advanced Computer Studies, 1997. Tech report, CS-TR-3798, UMIACS-TR-97-48.

[10] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *CASCON '98*, 1998.

[11] M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Optimizing execution of component-based applications using group instances. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, May 2001. Brisbane, Australia.

[12] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *MASS2000*, pages 119–133. National Aeronautics and Space Administration, Mar. 2000. NASA/CP 2000-209888.

[13] S. Booth and E. Mourao. Single sided MPI implementations for SUN MPI. In *SC2000: High Performance Networking and Computing Conf.* ACM/IEEE, 2000.

[14] D. Box et al. Simple object access protocol (SOAP) 1.1. Technical report, W3C, 2000. W3C Note 08 May.

[15] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: an architecture for a resource management and scheduling system in a global computation grid. In *HPC Asia*, 2000.

[16] R. Buyya et al. Problem solving environment comparison white paper. *http://www.eece.unm.edu/ apm*, 2001.

[17] Cactus Webmeister. The cactus code website. *http://www.CactusCode.org*, 2000.

[18] N. Carriero and D. Gelernter. Application experience with Linda. In *Proc. ACM/SIGPLAN Symposium on Parallel Programming*, pages 173–187, 1988.

[19] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[20] H. Casanova, S. Matsuoka, and J. Dongarra. Network-enabled server systems and the computational grid. In *Proc. High Performance Computing Symposium (HPC'01)*, 2001. To appear.

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

[22] I. S. Committee. IMPI - interoperable message-passing interface. http://impi.nist.gov/.

[23] J. Czyzyk, M. Mesnier, and J. More. NEOS: The network-enabled optimization system. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1996. Technical Report MCS-P615-1096.

[24] B. Dreier, M. Zahn, and T. Ungerer. The rthreads distributed shared memory. In *Proc. Third Int. Conf. On Massively Parallel Computing Systems*, 1998.

[25] G. Edjlali, A. Sussman, and J. Saltz. Interoperability of data parallel runtime libraries. In *Proceedings of the Eleventh International Parallel Processing Symposium*. IEEE Computer Society Press, Apr. 1997.

[26] R. Englander. *Developing Java Beans*. O'Reilly, 1997.

[27] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353. Springer, 2000.

[28] G. E. Fagg, K. S. London, and J. J. Dongarra. MPI_Connect: managing heterogeneous MPI applications interoperation and process control. In V. Alexandrov and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 93–96. Springer, 1998. 5th European PVM/MPI Users' Group Meeting.

[29] S. Fink and S. Baden. Runtime support for multi-tier programming of block-structured applications on SMP clusters. *International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE '97)*, December 1997. Available at www-cse.ucsd.edu/groups/hpcl/scg/kelp/pubs.html.

[30] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375, 1997.

[31] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *J. Parallel and Distributed Computing*, 40:35–48, 1997.

[32] I. Foster and N. T. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of SC 98*. IEEE, Nov. 1999. http://www.supercomp.org/sc98.

[33] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance rservations and co-allocation. In *7th IEEE/IFIP International Workshop on Quality of Service*, 1999.

[34] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Intl. J. Supercomputer Applications*, 2001. To appear. Available from http://www.globus.org/research/papers/anatomy.pdf.

[35] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed computing in a heterogenous computing environment. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science. Springer, 1998.

[36] D. Gannon et al. CCAT: The common component architecture toolkit. *http://www.extreme.indiana.edu/ccat*, 2000.

[37] D. Gannon et al. Engineering and science portals: Building applications for the grid. *http://www.extreme.indiana.edu/g̃annon/paris_grid_portals.ppt*, 2001.

[38] M. Govindaraju et al. Requirements for and evaluation of rmi protocols for scientific computing. In *SC2000*, 2000.

[39] G. F. W. Group. Grid computing environments. *http://www.computingportals.org*, 2001.

[40] S. Hariri et al. *A Problem Solving Environment for Network Computing.* IEEE Computer Society, 1998.

[41] W. Hoschek et al. Data management in an international data grid project. In *First IEEE/ACM International Workshop on Grid Computing*, pages 77–90, 2000.

[42] T. Imamura, Y. Tsujita, H. Koide, and H. Takemiya. An architecture of Stampi: MPI library on a cluster of parallel computers. In J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, volume 1908 of *Lecture Notes In Computer Science*, pages 200–207. Springer, Sept. 2000. 7th European PVM/MPI Users' Group Meeting.

[43] JaCo3. Jaco3: Java and CORBA-based collaborative environments for coupled simulations. *http://www.arttic.com/projects/JACO3*, 2000.

[44] H. Jacobsen et al. High performance corba working group. *http://www.omg.org/realtime/working_groups/high_performance_corba.html*, 2001.

[45] C. Johnson, S. Parker, and D. Weinstein. Large-scale computational science applications using the scirun problem solving environment. In *Supercomputing 2000*, 2000.

[46] C. Kesselman. CC++. In *Parallel Programming Using C++*. MIT Press, 1996.

[47] T. Kielmann et al. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Symposium on Principles and Practice of Parallel Programming*, pages 131–140, May 1999. Atlanta, GA.

[48] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Ppopp'99)*, pages 131–140. ACM, May 1999.

[49] T. Kurc, M. Beynon, A. Sussman, and J. Saltz. Developing data-intensive applications in the grid. *http://www.eece.unm.edu/˜dbader/grid/WhitePapers/KurcV2.pdf*, October 2000.

[50] LANL. POOMA: Parallel object-oriented methods and applications. *http://www.acl.lanl.gov/PoomaFramework*, 2000.

[51] C. Lee and J. Stepanek. On future global grid communication performance. *10th IEEE Heterogeneous Computing Workshop*, May 2001.

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

[53] D. Loveman. High performance fortran. *IEEE Parallel & Distributed Technology*, pages 25–43, 1993.

[54] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. http://www.mpi-forum.org/.

[55] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. http://www.mpi-forum.org/.

[56] Message Passing Interface Forum. *MPI-2 Journal of Development*, July 1997. http://www.mpi-forum.org/.

[57] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads Programming.* O'Reilly Publishers, 1996.

[58] OpenMP Consortium. *OpenMP C and C++ Application Program Interface, Version 1.0*, 1997.

[59] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. MPICH-GQ: Quality-of-service for message passing programs. In *Proceedings of SC 2000*. IEEE, 2000. http://www.sc2000.org.

[60] J. Saltz, G. Agrawal, C. Chang, R. Das, G. Edjlali, P. Havlak, Y.-S. Hwang, B. Moon, R. Ponnusamy, S. Sharma, A. Sussman, and M. Uysal. Programming irregular applications: Runtime support, compilation and tools. In M. Zelkowitz, editor, *Advances in Computers*, volume 45, pages 105–153. Academic Press, 1997.

[61] B. Saniya, H. Bal, and J. Ceriel. A task- and data-parallel programming language based on shared objects. *ACM Trans. on Programmming Languages and Systems*, pages 1131–1170, 1998.

[62] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf: Network-based information library for globally high performance computing. In *Parallel Object-Oriented Methods and Applications (POOMA)*, pages 39–48, 1996. http://ninf.etl.go.jp.

[63] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.

[64] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2), June 1998.

[65] J. M. Squyres, A. Lumsdaine, W. L. George, J. G. Hagedorn, and J. E. Devaney. The interoperable message passing interface (IMPI) extensions to LAM/MPI. In *Proceedings, MPIDC'2000*, March 2000.

[66] C. Szyperski. *Component Software: Beyond Object-oriented Programming*. Addison-Wesley, 1999.

[67] The GriPhyN Project. Grid physics network. *http://www.griphyn.org*, 2000.

[68] The Jini Community. The community resource for jini technology. *http://www.jini.org*, 2000.

[69] The Object Management Group. CORBA 3 Release Information. *http://www.omg.org/technology/corba/corba3releaseinfo.htm*, 2000.

[70] The PPDG Project. Particle physics data grid. *http://www.cacr.caltech.edu/ppdg*, 2000.

[71] M. Thomas et al. The Grid Portal Toolkit. *http://gridport.npaci.edu*, 2001.

[72] M. Thomas et al. The NPACI HotPage. *http://hotpage.npaci.edu*, 2001.

[73] S. Vajracharya, S. Karmesin, P. Beckman, et al. SMARTS: Exploiting temporal locality and parallelism through vertical execution. *International Conference on Supercomputing*, 1999.

[74] S. Verma, J. Gawor, G. von Laszewski, and M. Parashar. A CORBA commodity grid kit. In *Grid 2001*, November 2001. To appear.

[75] G. Wilson and P. Lu. *Parallel Programming Using C++*. MIT Press, 1996.

[76] S. Zenith. Ease: The model and its implementation. In *Proc. of a Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors*, 1992. Appeared as SIGPLAN Notices 28, 1, 1993.