Grid Programming Models: Current Tools, Issues and Directions

Craig Lee

Computer Systems Research Department
The Aerospace Corporation, P.O. Box 92957
El Segundo, CA USA
lee@aero.org

Domenico Talia

DEIS
Universitá della Calabria
87036 Rende, CS Italy
talia@deis.unical.it

Abstract

Grid programming must manage computing environments that are 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 environments. 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. This chapter discusses issues, properties and capabilities of grid programming models and tools to support efficient grid programs and their effective development. The main issues are outlined 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. Advanced programming support techniques are analyzed discussing possibilities for their effective implementation on grid environments.

1 Introduction

The main goal of grid programming is the study of programming models, tools and methods that support the effective development of portable and high-performance algorithms and applications on grid environments. 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 open-ended, 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 design 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* [21] 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 some control over each resource type if necessary? If the proper abstraction is used, can such transparency be provided by the run-time system? Can *discovery* of those resources be assisted or hidden 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 upwards of $\approx 10^8$-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? What programming models, abstractions, tools, or methodologies will permit the required *scalability* for a given application?

In light of these issues, we must clearly identify where current programming models are 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. The term "programming model" is used here since we are not just considering programming languages. A programming model can be present in many different forms, e.g., a language, a library API, or a tool with extensible functionality. Hence, programming models are present in frameworks, portals, and problem-solving environments, even through this is typically not their main focus. The most successful programming models will enable both high-performance and the flexible composition and management of resources. Programming models also influence the entire software lifecycle: design, implementation, debugging, operation, maintainence, etc. Hence, successful programming models should also facilitate the effective use of all manner of development tools, e.g., compilers, debuggers, performance monitors, etc.

First, we begin with a discussion of the major issues facing grid programming. We then take a short survey of common programming models that are being used or proposed in the grid environment. We next discuss programming techniques and approaches that can be brought to bear on the major issues, perhaps using the existing tools.

2 Grid Programming Issues

There are several general properties that are desirable for all programming models. Properties for parallel programming models have also been discussed [60]. 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 and present several major challenges.

2.1 Portability, Interoperability, and Adaptivity

Current high-level languages allowed codes to be processor independent. Grid programming models should enable codes to have similar portability. This could mean *architecture independence* in the sense of an interpreted virtual machine, but it can also mean the ability to use different pre-staged codes or services at different locations that provide equivalent functionality. Such portability is a necessary prerequisite for coping with dynamic, heterogeneous configurations.

The notion of using different but equivalent codes and services implies *interoperability* of programming model implementations. 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 where this is possible [21]. Finally, portability and interoperability promote *adaptivity*. A grid program should be able to adapt itself to different configurations based on available resources. This could occur at start-time, or at run-time due to changing application requirements or fault recovery. Such adaptivity could involve simple restart somewhere else or actual process and data migration.

2.2 Discovery

Resource discovery is an integral part of grid computing. Grid codes will clearly need to discover suitable hosts on which to run. However, since grids will host many *persistent services*, they must be able to discover these services and the interfaces they support. The use of these services must be programmable and composible in a uniform way. Therefore, programming environments and tools must be aware of available discovery services and offer a user explicit or implicit mechanisms to exploit those services when developing and deploying grid applications.

2.3 Performance

Clearly for many grid applications, performance will be an issue. Grids present heterogeneous bandwidth and latency hierarchies that can make it difficult to achieve high performance and good utilization of co-scheduled resources. The communication-to-computation ratio that can be supported in the typical grid environment will make this especially difficult for tightly coupled applications.

For many applications, however, *reliable* performance will be an equally important issue. A dynamic, heterogeneous environment could produce widely varying performance results that may be unacceptable in certain situations. Hence, in a shared environment, *quality of service* will become increasingly necessary to achieve reliable performance for a given programming construct on a given resource configuration. While some users may require an actual deterministic performance model, it may be more reasonable to provide reliable performance within some statistical bound.

2.4 Fault Tolerance

The dynamic nature of grids means that some level of fault tolerance is necessary. This is especially true for highly distributed codes, such as Monte Carlo or parameter sweep applications, that could initiate thousands of similar, independent jobs on thousands of hosts. Clearly, as the number of resources involved increases, so does the probability that some resource will fail during the computation. Grid apps 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.

2.5 Security

Grid codes will commonly run across multiple administrative domains using shared resources such as networks. While providing strong authentication between two sites is crucial, in time, it will not be uncommon that an application will involve multiple sites all under program control. There could, in fact, be call trees of arbitrary depth where the selection of resources is dynamically decided. Hence, a security mechanism that provides authentication (and privacy) must be integral to grid programming models.

2.6 Program Meta-Models

Beyond the notion of just interface discovery, complete grid programming will require models about the programs themselves. Traditional programming with high-level languages relies on a compiler to make a translation between two programming models. That is to say, between a high-level language, such as Fortran or C, and the hardware instruction set presented by a machine capable of applying a sequence of functions to data recorded in memory. Part of this translation process can be the construction of a number of models concerning the semantics of the code, and the application of a number of enhancements, such as optimizations, garbage-collection, and range-checking. Different but analogous *meta-models* will be constructed for grid codes. The application of enhancements, however, will be complicated by the distributed, heterogeneous grid nature.

3 A Brief Survey of Grid Programming Tools

How these issues are addressed 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. We now provide a brief survey of many specific tools, languages and environments for grids. Many, if not most, of these systems have their roots in "ordinary" parallel or distributed computing and are being applied in grid environments because they are established programming methodologies. We discuss both programming models and tools that are are actually available today, and those that are being proposed or represent an important set of capabilities that will eventually be needed. Broader surveys are available in [44] and [60].

3.1 Shared-State Models

Shared-state programming models are typically associated with tightly coupled, synchronous languages and execution models that are intended for shared memory machines or distributed memory machines with a dedicated interconnection network that provides very high bandwidth and low latency. While the relatively low bandwidths and deep, heterogeneous latencies across grid environments will make such tools ineffective, there are nonetheless programming models that are essentially based on shared state where the producers and consumers of data are decoupled.

3.1.1 JavaSpaces

JavaSpaces [22] is a Java-based implementation of the Linda tuplespace concept, in which tuples are represented as serialized objects. The use of Java allows heterogeneous clients and servers to interoperate, regardless of their processor architectures and operating systems. The model used by JavaSpaces views an application as a collection of processes communicating between them by putting and getting objects into one or more *spaces*. A *space* is a shared and persistent object repository that is accessible via network. The processes use the repository as an exchange mechanism to get coordinated, instead of communicating directly with each other. The main operations that a process can do with a *space* are to *put*, *take* and *read* (copy) objects. On a *take* or *read* operation, the object received is determined by an *associative matching* operation on the type and arity of the objects put into the space. A programmer that wants to build a space-based application should design *distributed data structures* as a set of objects that are stored in one or more *spaces*. The new approach that the JavaSpaces programming model gives to the programmer makes building distributed applications much easier, even when dealing with such dynamic, environments. Currently, efforts to implement JavaSpaces on grids using Java toolkits based on Globus are on-going [56, 55].

3.1.2 Publish/Subscribe

Besides being the basic operation underlying JavaSpaces, *associative matching* is a fundamental concept that enables a number of important capabilities that can't be accomplished any other way. These capabilities include *content-based routing*, *event services*, and *publish/subscribe* communication systems [45]. As mentioned earlier, this allows the producers and consumers of data to coordinate in a way where they can be decoupled and may not even know each other's identity.

Associative matching is, however, notoriously expensive to implement, especially in wide-area environments. On the other hand, given the importance of *publish/subscribe* to basic grid services, such as an event services which play an important role in supporting fault-tolerant computing, such a capability will have to be available in some form. Significant work is being done in this area to produce implementations with acceptable performance, perhaps by constraining individual instantiations to an single application's problem space. At least three different implementation approaches are possible [42]:

- *Network of Servers.* This is the traditional approach for many existing, distributed services. The CORBA Event Service [68] is a prime example, providing decoupled communication between producers and consumers using a hierarchy of clients and servers. The fundamental design space for server-based event systems can be partitioned into (1) the local matching problem, and (2) broker network design [33].

- *Middleware.* An advanced communication services could also be encapsulated in a layer of middleware. A prime example here is *A Forwarding Layer for Application-level Peer-to-Peer Services* (*FLAPPS* [51]). *FLAPPS* is a routing and forwarding middleware layer in user-space interposed between the application and the operating system. It is comprised of three interdependent elements: (1) peer network topology construction protocols, (2) application-layer routing protocols and (3) explicit request forwarding. *FLAPPS* is based on the store-and-forward networking model, where messages and requests are relayed hop-by-hop from a source peer through one or more transit peers en route to a remote peer. Routing behaviors can be defined over an application-defined *name space* that is hierarchically decomposable such that collections of resources and objects can be expressed compactly in routing updates.

- *Network Overlays.* The topology construction issue can be separated from the server/middleware design by the use of *network overlays*. Network overlays have generally been used for *containment, provisioning,* and *abstraction* [71]. In this case, we are interested in abstraction, since network overlays can make isolated resources appear to be virtually contiguous with a specific topology. These resources could be service hosts, or even active network routers, and the communication service involved could require and exploit the virtual topology of the overlay. An example of this is a communication service that uses a tree-structured topology to accomplish time management in distributed, discrete-event simulations [43].

3.2 Message-Passing Models

In message-passing models, processes run in disjoint address spaces and information is exchanged using message passing of one form or another. While the explicit parallelization with message passing can be cumbersome, it gives the user full control and is thus applicable to problems where more convenient semi-automatic programming models may fail. It also forces the programmer to consider exactly where a potential expensive communication must take place. These two points are important for single parallel machines, and even more so for grid environments.

3.2.1 MPI and Variants

The Message Passing Interface (MPI) [49, 50] is a widely adopted standard that defines a two-sided message passing library, i.e., with matched sends and receives, that is well-suited for grids. Many implementations and variants of MPI have been produced. The most prominent for grid computing is MPICH-G2.

MPICH-G2 [19] is a grid-enabled implementation of the MPI that uses the Globus services (e.g., job startup, security), and allows programmers to couple multiple machines, potentially of different architectures, to run MPI applications. MPICH-G2 automatically converts data in messages sent between machines of different architectures and supports multiprotocol communication by automatically selecting TCP for inter-machine messaging and vendor-supplied MPI for intra-machine messaging. MPICH-G2 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 multi-machine 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.).

The popularity of MPI has spawned a number of variants that address grid-related issues such as dynamic process management and more efficient collective operations. The MagPIe library [36], for example, implements MPI's

collective operations such as broadcast, barrier, and reduce operations with optimizations for wide area systems as grids. Existing parallel MPI applications can be run on Grid platforms using MagPIe by relinking with the MagPIe library. MagPIe has a simple API through which the underlying Grid computing platform provides the information about the number of clusters in use, and which process is located in which cluster. PACX-MPI [24] has improvements for collective operations and support for inter-machine communication using TCP and SSL. Stampi [30] has support for MPI-IO and MPI-2 dynamic process management. MPI_Connect [15] enables different MPI applications, under potentially different vendor MPI implementations, to communicate.

3.2.2 One-sided Message-Passing

While having matched send/receive pairs is a natural concept, *one-sided communication* is also possible and included in MPI-2 [50]. In this case, a *send* operation does not necessarily 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 [7].

A number of one-sided communication tools exist. One that supports multi-protocol communication suitable for grid environments is Nexus [18]. 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++ [10], and also MPI.

3.3 RPC and RMI Models

Message-passing models, whether they are point-to-point, broadcast, or associatively addressed, all have the essential attribute of explicitly marshalled arguments being sent to a matched receive that unmarshalls the arguments and decides the processing, typically based on message type. The semantics associated with each message type is usually defined statically by the application designers. One-sided message-passing models alter this paradigm by not requiring a matching receive and allowing the sender to specify the type of remote processing. Remote Procedure Call (RPC) and Remote Method Invocation (RMI) models provide the same capabilities as this, but structure the interaction between sender and receiver more as a language construct, rather than a library function call that simply transfers an uninterpreted buffer of data between points A and B. RPC and RMI models provide a simple and well-understood mechanism for managing remote computations. Besides being a mechanism for managing the flow of control and data, RPC and RMI also enable some checking of argument type and arity. RPC and RMI can also be used to build higher-level models for grid programming, such as components, frameworks, and network-enabled services.

3.3.1 Grid-enabled RPC

GridRPC [52] is an RPC model and API for grids. Besides providing standard RPC semantics with asynchronous, coarse-grain, task-parallel execution, it provides a convenient, high-level abstraction whereby the many details of interacting with a grid environment can be hidden. Three very important grid capabilities that GridRPC could transparently manage for the user are:

- *Dynamic resource discovery and scheduling.* RPC services could be located anywhere on a grid. Discovery, selection and scheduling of remote execution should be done based on user constraints.
- *Security.* Grid security via GSI and X.509 certificates is essential for operating in an open environment.
- *Fault Tolerance.* Fault tolerance via automatic checkpoint, rollback, or retry becomes increasingly essential as the number of resources involved increases.

The management of interfaces is an important issue for all RPC models. Typically this is done in an *Interface Definition Language* (IDL). GridRPC was also designed with a number of other properties in this regard to both improve usability and ease implementation and deployment:

- *Support for a "scientific IDL".* This includes large matrix arguments, shared-memory matrix arguments, file arguments, and call-by-reference. Array strides and sections can be specified such that communication demand is reduced.
- *Server-side-only IDL management.* Only GridRPC servers manage RPC stubs and monitor task progress. Hence, the client-side interaction is very simple and requires very little client-side state.

Two fundamental objects in the GridRPC model are *function handles* and the *session IDs*. GridRPC function names are mapped to a server capable of computing the function. This mapping is subsequently denoted by a function handle. The GridRPC model does not specify the mechanics of resource discovery, thus allowing different implementations

to use different methods and protocols. All RPC calls using a function handle will be executed on the server specified by the handle. A particular (non-blocking) RPC call is denoted by a session ID. Session IDs can be used to check the status of a call, wait for completion, cancel a call, or check the returned error code.

It is not surprising that GridRPC is a straight-forward extension of network-enabled service concept. In fact, prototype implementations exist on top of both Ninf [53] and NetSolve [2]. The fact that server-side-only IDL management is used means that deployment and maintenance is easier than other distributed computing approaches, such as CORBA, where clients have to be changed when servers change. We note that other RPC mechanisms for grids are possible. These include SOAP [62] and XML-RPC[74] which use XML over HTTP. While XML provides tremendous flexibility, it currently has limited support for scientific data, and a significant encoding cost [29]. Of course, these issues could be rectified with support for, say, double-precision matrices, and binary data fields. We also note that GridRPC could, in fact, be hosted on top of OGSA [59].

3.3.2 Java RMI

Remote invocation or execution is a well-known concept which has been underpinning the development of both originally RPC and then Java's RMI. Java Remote Method Invocation (RMI) enables a programmer to create distributed Java-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. RMI inherits basic RPC design in general, it has distinguishing features that reach beyond the basic RPC. With RMI, a program running on one JVM can invoke methods of other objects residing in different JVMs. The main advantages of RMI are that it is truly object-oriented, that it supports all the data types of a Java program, and that it is garbage collected. These features allow for a clear separation between caller and callee. Development and maintenance of distributed systems become easier. Java's RMI provides a high-level programming interface that is well-suited for Grid computing [26] that can be effectively used when efficient implementations of it will be provided.

3.4 Hybrid Models

The inherent nature of grid computing is to make all manner of hosts available to grid applications. Hence, some applications will want to run both within and across address spaces. That is to say, they will want to run perhaps multi-threaded within a shared-address space, and also by passing data and control between machines. Such a situation occurs in *clumps* (clusters of symmetric multiprocessors) and also in grids. A number of programming models have been developed to address this.

3.4.1 OpenMP and MPI

OpenMP [54] is a library that supports parallel programming in shared-memory parallel machines. It has been developed by a consortium of vendors with the goal of producing a standard programming interface for parallel shared-memory machines that can be used within mainstream languages, such as Fortran, C, and C++. OpenMP allows for the parallel execution of code (*parallel DO loop*), the definition of shared data (*SHARED*), and synchronization of processes.

The combination of both OpenMP and MPI within one application to address the clump and grid environment has been considered by many groups [61]. A prime consideration in these application designs is "who's on top". OpenMP is essentially a multithreaded programming model. Hence, OpenMP on top of MPI requires MPI to be thread-safe or requires the application to explicit manage access to the MPI library. (The MPI standard claims to be "thread-compatible" but the thread-safety of a particular implementation is another question.) MPI on top of OpenMP can requirement additional synchronization and limit the amount of parallelism OpenMP can realize. Which approach actually works out best is typically application-dependent.

3.4.2 OmniRPC

OmniRPC [57] was specifically designed as a thread-safe RPC facility for clusters and grids. OmniRPC uses OpenMP to manage thread-parallel execution while using Globus to manage grid interactions. Rather than using message-passing between machines, however, it provides RPC. OmniRPC is, in fact, a layer on top of Ninf. Hence, it uses the Ninf machinery to discover remote procedure names, associate them with remote executables, and retrieve all stub interface information at run-time. To manage multiple RPCs in a multi-threaded client, OmniRPC maintains a queue of outstanding calls that is managed by a scheduler thread. A calling thread is put on the queue and blocks until the scheduler thread initiates the appropriate remote call and receives the results.

3.4.3 MPJ

All of these programming concepts can be put into one package, as is the case with message-passing Java, or MPJ [9]. The argument for MPJ is that many application naturally require the symmetric message-passing model, rather than the asymmetric RPC/RMI model. Hence, MPJ makes multithreading, RMI and message-passing available to the

application builder. MPJ message-passing closely follows the MPI-1 specification.

This approach, however, does present implementation challenges. Implementation of MPJ on top of a native MPI library provides good performance but breaks the Java security model and does not allow applets. A native implementation of MPJ in Java, however, usually provides slower performance. Additional compilation support may improve overall performance and make this single language approach more feasible.

3.5 Peer-to-Peer Models

Peer-to-peer (P2P) computing [28] is the sharing of computer resources and services by direct exchange between systems. Peer-to-peer computing takes advantage of existing desktop computing power and networking connectivity, allowing economical clients to leverage their collective power to benefit the entire enterprise. In a peer-to-peer architecture, computers that have traditionally been used solely as clients communicate directly among themselves and can act as both clients and servers, assuming whatever role is most efficient for the network. This reduces the load on servers and allows them to perform specialized services (such as mail-list generation, billing, etc) more effectively. As computers become ubiquitous, ideas for implementation and use of peer-to-peer computing are developing rapidly and gaining importance. Both peer-to-peer and grid technologies focus on the flexible sharing and innovative use of heterogeneous computing and network resources.

A family of protocols specifically designed for peer-to-peer computing is JXTA [27]. JXTA is a set of open, generalized peer-to-peer protocols, defined as XML messages, that allow any connected device on the network ranging from from cell phones and wireless PDAs to PCs and servers to communicate and collaborate in a P2P manner. Using the JXTA protocols, peers can cooperate to form self-organized and self-configured peer groups independently of their positions in the network (edges, firewalls), and without the need of a centralized management infrastructure. Peers may use the JXTA protocols to advertise their resources and to discover network resources (services, pipes, etc.) available from other peers. Peers form and join peergroups to create special relationships. Peers cooperate to route messages allowing for full peer connectivity. The JXTA protocols allow peers to communicate without needing to understand or manage the potentially complex and dynamic network topologies which are becoming common. These features makes JXTA a model for implementing peer-to-peer grid services and applications [55].

3.6 Frameworks, Component Models, and Portals

Besides these library and language tool approaches, entire programming environments to facilitate the development and deployment of distributed applications. We can broadly classify these approaches as frameworks, component models, and portals. We review a few important examples.

3.6.1 Cactus

The Cactus Code and Computational Toolkit [8] provides a modular framework for computational physics. As a framework, Cactus provides application programmers with a high-level API for a set of services tailored for computational science. Besides support for services like parallel I/O and parallel checkpointing and restart, there are services for computational steering (dynamically changing parameters during a run) and remote visualization. To build a Cactus application, a user builds modules, called *thorns*, that are plugged into the framework *flesh*. Full details are available elsewhere in this book.

3.6.2 CORBA

The Common Object Request Broker Architecture (CORBA) [68] is a standard tool where a meta-language interface is used 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 [32]. This endeavors to improve the performance of CORBA not only by improving ORB performance, but by enabling “aggregate” processing in clusters or parallel machines. Some of this work involves supporting parallel objects that understand how to communicate in a distributed environment [13].

3.6.3 CoG Kit

There are also efforts to make CORBA services directly available to grid computations. This is being done in the CoG Kit project [73] to enable “Commodity Grids” through

an interface layer that maps Globus services to a CORBA API. Full details are available elsewhere in this book.

3.6.4 Legion

Legion [47] 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 that all Legion development is now being done as part of the AVAKI Corporation [3].

3.6.5 Component Architectures

Components extend the object-oriented paradigm by enabling objects to manage the interfaces they present and discover those presented by others [65]. 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 [58], the CORBA 3 Component Model [68], Enterprise Java Beans and Jini [14, 67], and the Common Component Architecture [25]. Of these, the Common Component Architecture includes specific features for high-performance computing, such as *collective ports* and *direct connections*.

3.6.6 Portals

Portals can be viewed as providing a web-based interface to a distributed system. Commonly 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. 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.

A number of examples are possible in this area. One is the Grid Portal Toolkit, aka GridPort [69]. 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.

Another very important example is the XCAT Science Portal [37]. In this effort, portals are designed using a *notebook* of typical web pages, text, graphics, forms, and executable scripts. Notebooks have an interactive script/forms editor based on JPython that allows access to other tool kits such as CoG Kit and the XCAT implementation of the Common Component Architecture. This 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.

More information on portals is available elsewhere in this book.

3.7 Web Service Models

Grid technologies are evolving toward an Open Grid Services Architecture (OGSA) ([20] and elsewhere in this book) in which a grid provides an extensible set of services that virtual organizations can aggregate in various ways. OGSA defines a uniform exposed service semantics (the so called grid service) based on concepts and technologies from both the Grid and Web services communities. OGSA defines standard mechanisms for creating, naming, and discovering transient grid service instances, provides location transparency and multiple protocol bindings for service instances, and supports integration with underlying native platform facilities.

The OGSA effort aims to define a common resource model that is an abstract representation of both real resources, such as nodes, processes, disks, file systems, and logical resources. It provides some common operations and supports multiple underlying resource models representing resources as service instances. OGSA abstractions and services provide building blocks that developers can use to implement a variety of higher-level Grid services, but OGSA services are in principle programming language- and programming model-neutral. OGSA aims to define the semantics of a grid service instance: how it is created, how it is named, how its lifetime is determined, how to communicate with it, and so on.

OGSA does not, however, address issues of implementation programming model, programming language, implementation tools, or execution environment. OGSA definition and implementation will produce significant effects on grid programming models because these can be used to support and implement OGSA services and higher-level models could incorporate OGSA service model offering high-level programming mechanisms to use those services in grid applications. The Globus project is committed to developing an open source OGSA implementation by evolv-

ing the current Globus Toolkit towards an OGSA-compliant Globus Toolkit 3.0. This new release will stimulate the research community in developing and implementing OGSA-oriented programming models and tools.

3.8 Coordination Models

The purpose of a coordination model is to provide a means of integrating a number of possibly heterogeneous components together, by interfacing with each component in such a way that the collective set forms a single application that can execute on parallel and distributed systems [48]. Coordination models can be used to distinguish the computational concerns of a distributed or parallel application from the cooperation ones, allowing separate development but also the eventual fusion of the these two development phases.

The concept of coordination is closely related to those of heterogeneity. Since the coordination interface is separate from the computational one, therefore, the actual programming languages used to write computational code play no important role in setting up the coordination mechanisms. Furthermore, since the coordination component offer a homogeneous way for interprocess communication and abstracts from the architecture details, coordination encourages the use of heterogeneous ensembles of machines.

A coordination language offers composing mechanism and imposes some constraints on the forms of parallelism and on the interfacing mechanisms used to composed an application. Coordination languages for grid computing generally are orthogonal to sequential or parallel code used to implement the single modules that must be executed, but provide a model for composing programs and should implement inter-module optimizations that take into account machine and interconnection features for providing efficient execution on grids. Some recent research activities in this area use XML-based [70, 23] or skeleton-based models for grid programming. Another potential application domain for grid coordination tools is *workflow* [17], a model of enterprise work management where work units are passed between *processing points* based on procedural rules.

4 Advanced Programming Support

While these programming tools and models are extremely useful (and some are actually finding wide use), they may be underachieving in the areas of both performance and flexibility. While it may be possible to hand-code application using these models and low-level, common grid services that exhibit good performance and flexibility, we also have the goal of making these properties as easy to realize as possible. We discuss several possibilities for advanced programming support.

4.1 Traditional Techniques

While tightly coupled applications that have been typically supported by a shared-memory abstraction will not be effective in grids, there are a number of traditional performance enhancing techniques that can be brought to bear in grid codes. Work reported in [1] describes many of these techniques all applied to a single, tightly coupled MPI solver code run between two institutions separated by roughly two thousand kilometers.

- Overlapping computation with communication. This requires a grid-aware communication schedule such that it is known when boundary data can be exchanged while computation is done on the interior.
- Shadow arrays. The use of overlapping "ghostzones" allows more latency to be tolerated at the expense of some redundant computation.
- Aggregated communication. Communication efficiency can be improved by combining many smaller messages into fewer larger messages.
- Compression. With the smooth data in this physics simulation, very good compression ratios were achieved such that latency, and not bandwidth, became more of a problem.
- Protocol tuning. By tuning communication protocol parameters, such as the TCP window size, applications can realize better communication performance.

With all of these well-known techniques, respectable scaling is claimed (88% and 63%) for the problem size and resources used. The outstanding issue here is how well these techniques can be incorporated into programming models and tools such that they can be transparently applied to grid applications.

4.2 Data-driven Techniques

Besides improving communication performance, the techniques in the previous section are also oriented towards providing a more loosely coupled execution of the MPI code. How can a more asynchronous, loosely coupled execution model be realized to support programming models? Clearly *data-driven* programming techniques can facilitate this. While such models can suffer from excessive operand matching and scheduling overheads, restricted, coarse-grain forms can realize significant net benefits. *Workflow* is an instance of this model. Another instance is *stream programming*.

As exemplified by the DataCutter framework [6, 5], stream programming can be used to manage the access to

large data stores and associate processing with communication in a distributed manner. Data sets that are too large to easily copy or move can be accessed through upstream filters that can do spatial filtering, decimation, corner-turns, or caching copies "closer to home". The co-allocation of filters and streams in a grid environment is an important issue. Stream programming can also have a wide variety of semantics and representations, as catalogued in [41]. This is also closely related to the notion of advanced communication services discussed below.

4.3 Speculative or Optimistic Techniques

Another method for producing a more asynchronous, loosely coupled execution is that of *speculative* or *optimistic computing*. By direct analogy with optimistic discrete-event simulation, speculative or optimistic computing is the relaxation of synchronization and communication requirements by allowing speculative execution among multiple hosts with the probability that some work optimistically computed will have to be discarded when it is determined to be incompatible or redundant. The goal is to control the level of optimism such that the benefits of loosely coupled execution are maximized while the overhead of wasted computation is minimized, thus hitting a "sweetspot".

An example of the use of optimistic computation in an application domain is that of *optimistic mesh generation*. The Parallel Bowyer-Watson method [11] for mesh generation allows an implementation where boundary cavity generation can be computed optimistically with a control parameter for the level of optimism. This should enable the generation code to stay in an "operating region" where reasonable performance and utilization is realized.

4.4 Distributed Techniques

Yet another method is the distribution of processing over the data. In a grid's deep, heterogeneous latency hierarchy, synchronous data-parallel language approaches will clearly be inappropriate. Assuming that synchronization and inter-communication requirements are not excessively dense, however, distributed techniques can achieve very high aggregate bandwidths between local data and processing.

This basic technique has been applied in contexts other than grids. The *macroserver* model developed to support the *processor-in-memory* (PIM) technology in the HTMT uses a coarse-grain, message-driven approach [75]. Besides executing code "at the sense amps", *parcels* containing code, data and environment *percolate* through the machine to the PIM where they execute. The goal, of course, is to hide all latency.

An analogous approach can be taken in grid environments, but on a completely different scale. The Grid Datafarm architecture [66] is designed to exploit access locality by scheduling programs across a large-scale distributed disk farm that has processing close to the storage. To promote tight coupling between storage and processing, the *owner computes* rule (as devised for data-parallel languages) was adopted here. The Grid Datafarm also provides a parallel I/O API.

4.5 Grid-Aware I/O

While I/O systems may concentrate on the movement of data, they can certainly have a large effect on how programs are written. Grid Datafarm files are distributed across disks but they can be opened, read, and written as a single, logical file. For communication within a program, the KeLP system [16, 4] uses a notion of *structural abstraction* and an associated *region calculus* to manage message-passing, thread scheduling and synchronization. While developed to support POOMA (Parallel Object-Oriented Methods and Applications) [38], SMARTS (Shared Memory Asynchronous RunTime System) [72] may have applicability in the grid environment. SMARTS uses macro-dataflow scheduling to manage coarse-grain data-parallel operations and hide latency. It would be interesting to determine if this approach is scalable to the grid environment.

4.6 Advanced Communication Services

Feasible programming models may depend on the infrastructure support that is available. Advanced communication services is part of this infrastructure. What is meant here by "advanced communication services" is essentially any type of semantics associated with communication beyond the simple, reliable unicast transfer of data from point A to point B, or even the multicast of data from one to many. Hence, what constitutes an advanced communication service can be broadly defined and can be motivated by different factors.

In grid computations, understanding and utilizing the network topology will be increasingly important since overall grid communication performance will be increasingly dominated by propagation delays. That is to say, in the next five to ten years and beyond, network "pipes" will be getting fatter (as bandwidths increase) but not commensurately shorter (due to latency limitations) [46]. To maintain performance, programming tools such as MPI will have to become *topology-aware*. An example of this is MagPIe [35]. MagPie transparently accommodates wide-area clusters by minimizing the data traffic for collective operations over the slow links. Rather than being governed by *O(nlogn)* messages across the diameter of the network, as is typical,

topology-aware collective operations could be governed by just the average diameter of the network.

Another motivating factor for advanced communication services is the need for fundamentally different communication properties. Such is the case for *content-based* or *policy-based* routing. A traditional multicast group, for example, builds relevant routing information driven by the physical network topology. Content-based routing would enable an application to control the communication scheduling, routing and filtering based on the application's dynamic communication requirements within a given multicast group, rather than always having to use point-to-point communication. Of course, this requires topology-awareness at some level.

Hence, advanced communication services can be classified into several broad categories. Some of these services are simply more efficiently implemented when topology-aware, while other are not possible any other way [39].

- *Augmented communication semantics.* Rather than changing fundamental routing behaviors, etc., much communication could simply be augmented with additional functionality. Common examples of this include caching (web caching), filtering, compression, encryption, quality of service, data-transcoding, or other user-defined functions.

- *Collective operations.* Applications may require synchronous operations, such as barriers, scans and reductions. These operations are typically implemented with a communication topology based on point-to-point operations. For performance in a wide-area network, it is crucial to match these operations to the topology defined by the physical or virtual network.

- *Content-based and Policy-based Routing.* Content-based routing is a fundamentally different paradigm that enables a host of important capabilities. By allowing applications to determine routing based on application-defined fields in the data payload, this enable capabilities such as publish/subscribe for interest management, event services, and even tuple spaces. Policy-based routing is also possible. Examples of this include routing to meet QoS requirements and message consistency models where a policy must be enforced on the message arrival order across some set of end-hosts.

- *Communication scope.* Some communication services could be expensive to implement, especially on a large scale. Hence, if applications could define their own *scope* for the service, then they could keep the problem size to a minimum, thereby helping the service to remain feasible. Communication scope could be associated with a *named topology* such that multiple scopes can be managed simultaneously for the same or separate applications.

Many of these services are suitable for the implementation approaches discussed in Section 3.1.2.

4.7 Security

Grid applications may want authentication, authorization, integrity checking and privacy. In the context of a programming model, this carries additional ramifications. Basic, point-to-point security can be accomplished by integrating a security mechanism with a programming construct. An example of this is the integration of SOAP with GSI [31]. In the large context, however, such RMI or RPC calls could exist in a *call tree*. Supporting security along a call tree requires the notion of *delegation of trust*. We note that cancellation of a secure call could require the revocation of delegated trust [40].

Signing and checking certificates on an RPC also represents an overhead that must be balanced against the amount of work represented by the RPC. Security overheads could be managed by establishing secure, trusted domains. RPCs within a domain could dispense with certificates; RPCs that cross domains would have to use them. Trusted domains could be used to limit per-RPC security overheads in favor the one-time cost of establishing the domain.

4.8 Fault Tolerance

Reliability and fault tolerance in grid programming models/tools are largely unexplored, beyond simple checkpointing and restart. 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.

These capabilities require that *event models* be integral to grid programming models and tools [40]. Event models

are required for many aspects of grid computing, such as a performance monitoring infrastructure. Hence, it is necessary that a widely deployed grid event mechanism become available. The use of such a mechanism will be a key element for reliable and fault tolerant programming models.

As a case in point, consider a Grid RPC mechanism. An RPC typically has a *call* and a *return* in a call tree, but it can also have a *cancellation* or *rejection*. For a chain of synchronous RPCs, cancellation or rejection must flow along one linear path. For multiple, asynchronous RPCs, however, cancellations and rejections may have to flow along multiple branches. Rejections may also precipitate cancellations on other branches.

Hence, a Grid RPC mechanism clearly needs an event service to manage cancellation and rejection. This is critical to designing and implementing an RPC mechanism that is fault tolerant, i.e., a mechanism where any abnormal operation is detected within a bounded length of time and reliable signaling occurs whereby the RPC service cleans-up any obsolete state. Reliability of any cancel and reject events is critical to achieving any fault tolerance.

While the simplest (and probably the most common) case of cancellation will involve one RPC handle that is carried on a single event delivered point-to-point, it may be useful to cancel RPCs en masse. In this case, RPCs could be identified as members of a process group. Such a process group may include (1) the single active branch of a call tree, (2) a parallel call tree with a single root, or (3) one or more branches of a call tree where the parent or root node is not a member.

Cancellation of an entire process group could be accomplished by point-to-point events. However, 1-to-many or some-to-all event notification would enable the entire group to be cancelled more quickly by "short-circuiting" the call tree topology. Such group event notification could be accomplished by membership in the group (as in membership in a multicast group) or by a publish/subscribe interface whereby remote RPC servers subscribe to cancellation events for the process groups of the RPCs they are hosting.

4.9 Program Meta-models and Grid-Aware Runtime Systems

Another serious issue in grid programming models is the concept of *program meta-models* and their use by grid-aware runtime systems. Regardless of how grids are ultimately deployed, they will consist of components and services that are either persistent or can be instantiated. Some of these components and services will become widely used and commonly available. Hence, many applications will be built, in part or in whole, through the composition of components and services.

How can such composition be accomplished automatically such that characteristics such as performance are understood and maintained, in addition to maintaining properties such as security and fault tolerance? This can only be done by producing *meta-models* that define a component's characteristics and properties. Meta-models could be produced by hand, but they could also be produced automatically.

Hence, compilers and composition tools could be responsible for producing meta-models and using them to identify and enforce valid compositions. In this context, "valid" can mean more than just whether the interface arguments are compatible. Valid could mean preserving performance characteristics, security properties, or fault tolerance. Based on a high-level program description (as in a portal scripting language, for instance), a "compiler" could map higher-level semantics to lower-level components and services. This raises the possibility of the definition of a "grid compiler target"; not in the traditional sense of a machine instruction set, but rather as a set of commonly available services.

Preliminary work has been done in this area by the Grid Application Development Software (GrADS) project [34]. The GrADS approach is based (1) the GrADS Program Preparation System, and (2) the GrADS Program Execution System. The Program Preparation System takes user input, along with reusing components and libraries, to produce a *configurable object program*. These objects are annotated with their resource requirements and predicted performance. The Execution Environment uses this information to select appropriate resources for execution, and also to monitor the application's compliance with a performance "contract".

This type of approach is a cornerstone for the Dynamic Data-Driven Application Systems (DDDAS) concept [12] being developed by the National Science Foundation. DDDAS promotes the notion of a *run-time compiling system* that accomplishes *dynamic application composition*. The ultimate goal of DDDAS, however, is to enable dynamic applications that can discover and ingest new data on-the-fly, and automatically form new collaborations with both computational systems and physical systems through a network of sensors and actuators.

5 Conclusion

We have considered programming models for grid computing environments. As with many fundamental areas, what will comprise a successful grid programming model consists of many aspects. To reiterate, these include portability, interoperability, adaptivity, and the ability to support discovery, security, and fault tolerance while maintaining performance. We identified a number of topics where further work is needed to realize important capabilities, such

as data-driven and optimistic programming techniques, advanced communication and I/O services, and finally program meta-models.

Regardless of these insights, however, the programming models and tools that get developed will largely depend which models and tools are considered to be the *dominant paradigm*. Enhancements to these models and tools will have a lower *barrier to acceptance* and will be perceived as potentially have a *broader impact* on a larger community of users. There is also a distinction to be made between the capabilities supported in the common infrastructure and the programming models and tools built on top of them.

With regards to common infrastructure, the tremendous commercial motivation for the development of web services means that it would be a mistake not to leverage these capabilities for scientific and engineering computation. Hence, for these practical reasons, we will most likely see continued development of the Open Grid Services Architecture.

With regards to programming models and tools, and for the same practical reasons, we will also most likely see continued development in MPI. MPI is a standard with an established user-base. Many of the potential enhancements discussed earlier could be provided in MPI with minimal changes to the API.

Other models and tools, however, will see increasing development. Frameworks that provide a rich set of services on top of common grid services, such as Cactus and XCAT, will incorporate many of the capabilities we have described. For other applications, however, a fundamental programming construct that is grid-aware, such as GridRPC, will be completely sufficient for their design and implementation.

Finally we discuss the issue of *programming style*. This evolution in available computing platforms, from single machines to parallel machines to grids, will precipitate a corresponding evolution in how programming is done to solve computational problems. Programmers, by nature, will adapt their codes and programming style to accommodate the available infrastructure. They will strive to make their codes more loosely coupled. "Problem architectures" will be conceived in a way to make them better suited to the grid environment.

This raises a concomitant issue. Besides the tremendous flexibility that grids will offer for virtual organizations, what will be their limits for computational science? Will computational science be limited to the size of "single-chassis" machines, such as the ASCI machines [64] and the HTMT [63]? Or can the problem architectures for science and engineering, and their associated computational models, be made sufficiently *grid-friendly* such that increasingly large problems can be solved? Much work remains to be done.

References

[1] G. Allen et al. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. In *Supercomputing*, November 2001.

[2] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4. Computer Science Dept. Technical Report CS-01-467, University of Tennessee, Knoxville, TN, July 2001.

[3] AVAKI. The AVAKI corporation. *www.avaki.com*, 2001.

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

[5] 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.

[6] 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.

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

[8] Cactus Webmeister. The Cactus Code Website. *www.CactusCode.org*, 2000.

[9] B. Carpenter et al. MPJ: MPI-like Message-passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[10] 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.

[11] N. Chrisochoides and D. Nave. Parallel delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 1(1), 2001.

[12] F. Darema. Dynamic data-driven application sytems. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*. CRC Press, 2002.

[13] A. Denis, C. Pérez, T. Priol, and A. Ribes. Programming the Grid with Distributed Objects. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*. CRC Press, 2002.

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

[15] 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.

[16] 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.

[17] L. Fischer, editor. *The Workflow Handbook 2002*. Future Strategies, Inc., 2002.

[18] 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.

[19] I. Foster and N. T. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Supercomputing*. IEEE, November 1998. www.supercomp.org/sc98.

[20] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, pages 37–46, June 2002.

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

[22] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[23] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. An Integrated Grid Environment for Component Applications. In *Second International Workshop on Grid Computing (Grid 2001)*, pages 26–37, 2001. LNCS Vol. 2242.

[24] 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.

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

[26] V. Getov, G. von Laszewski, M. Philippsen, and I. Foster. Multi-Paradigm Communications in Java for Grid Computing. *Communication of the ACM*, pages 118–125, October 2001.

[27] L. Gong. JXTA: A Network Programming Environment. *IEEE Internet Computing*, 5(3), 2001.

[28] L. Gong, editor. *IEEE Internet Computing*, volume 6(1), Special Issue on Peer-to-Peer Networking. IEEE, 2002.

[29] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Proceedings of SC'2000, Dallas, TX*, 2000.

[30] 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.

[31] K. Jackson. pyGlobus: a Python Interface to the Globus Toolkit. *Concurrency and Computation: Practice and Experience*, 2002. To appear. Available from http://www.cogkits.org/ papers/c545python-cog-cpe.pdf.

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

[33] H.-A. Jacobsen and F. Llirbat. Publish/subscribe systems. In *17th International Conference on Data Engineering*, 2001. Tutorial.

[34] K. Kennedy et al. Toward a framework for preparing and executing adaptive grid programs. In *Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium)*, April 2002.

[35] 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.

[36] 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.

[37] S. Krishnan et al. The XCAT Science Portal. In *Supercomputing*, November 2001.

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

[39] C. Lee. On active grid middleware. *Second Workshop on Active Middleware Services*, August 1, 2000.

[40] C. Lee. Grid RPC, Events and Messaging. *www.eece.unm.edu/~apm/WhitePapers/APM_Grid_RPC_0901.pdf*, September 2001.

[41] C. Lee. Stream Programming: in Toto and Core Behavior. *www.eece.unm.edu/~apm/WhitePapers/stream.pdf*, September 2001.

[42] C. Lee, E. Coe, B. Michel, J. Clark, and B. Davis. Using advanced communication services in grid environments. In *Third International Grid Computing Workshop*, November 2002. Submitted.

[43] C. Lee, E. Coe, C. Raghavendra, et al. Scalable time management algorithms using active networks for distributed simulation. *DARPA Active Network Conference and Exposition*, May 29-30 2002.

[44] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz. A Grid Programming Primer. http://www.gridforum.org/7_APM/APS.htm, submitted to the Global Grid Forum, August 2001.

[45] C. Lee and B. Michel. The use of content-based routing to support events, coordination and topology-aware communication in wide-area grid environments. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*. CRC Press, 2002.

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

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

[48] D. Marinescu and C. Lee, editors. *Process Coordination and Ubiquitous Computing*. CRC Press, 2002.

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

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

[51] B. Michel and P. Reiher. Peer-to-Peer Internetworking. In *OPENSIG*, September 2001.

[52] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. GridRPC: A Remote Procedure Call API for Grid Computing. Technical report, Univ. of Tennesse, June 2002. ICL-UT-02-06.

[53] H. Nakada, M. Sato, and S. Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 15(5-6):649–658, 1999.

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

[55] O. F. Rana, V. S. Getov, E. Sharakan, S. Newhouse, and R. Allan. Building Grid Services with Jini and JXTA, February 2002. GGF2 Working Document.

[56] D. Saelee and O. F. Rana. Implementing services in a computational grid with jini and globus. In *First EuroGlobus Workshop*, 2001. See www.euroglobus.unile.it.

[57] M. Sato, M. Hirono, Y. Tanaka, and S. Sekiguchi. OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP. In *WOMPAT, LNCS 2104*, pages 130–136. Springer-Verlag, 2001.

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

[59] S. Shirasuna, H. Nakada, S. Matsuoka, and S. Sekiguchi. Evaluating Web Services Based Implementations of GridRPC. In *HPDC-11*, 2002.

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

[61] L. Smith and M. Bull. Development of mixed mode mpi/openmp applications. In *WOMPAT*, 2000.

[62] Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP/, May 2000. W3C Note.

[63] T. Sterling and L. Bergman. Design analysis of a hybrid technology multithreaded architecture for petaflops scale computation. In *International Conference on Supercomputing*, June 1999.

[64] Subcommittee on Computing, Information, and Communications R&D. DOE's ASCI Program. Technical report, DOE, 2000. www.hpcc.gov/pubs/blue00/asci.html.

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

[66] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid Datafarm Architecture for Petascale Data Intensive Computing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, pages 102–110, 2002.

[67] The Jini Community. The community resource for jini technology. *www.jini.org*, 2000.

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

[69] M. Thomas et al. The Grid Portal Toolkit. *gridport.npaci.edu*, 2001.

[70] R. Tolksdorf. Models of coordination and web-based systems. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*. CRC Press, 2002.

[71] J. Touch. Dynamic Internet Overlay Deployment and Management Using the X-Bone. *Computer Networks*, 2001.

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

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

[74] XML-RPC. http://www.xml-rpc.com/.

[75] H. Zima and T. Sterling. Macroservers: An object-based programming and execution model for processor-in-memory arrays. In *Proc.International Symposium on High Performance Computing (ISHPC2K)*, October 2000.