Distributed Computing Practice for Large-Scale Science & Engineering Applications

Shantenu Jha[1,2,3], Murray Cole[4], Daniel S. Katz[5,1,6],
Manish Parashar[7,8], Omer Rana[9], Jon Weissman[10]

[1]Center for Computation & Technology, Louisiana State University
[2]Department of Computer Science, Louisiana State University
[3]e-Science Institute, University of Edinburgh
[4]School of Informatics, University of Edinburgh
[5]Computation Institute, University of Chicago and Argonne National Laboratory
[6]Department of Electrical and Computer Engineering, Louisiana State University
[7]NSF Center for Autonomic Computing, Rutgers University
[8]Department of Electrical and Computer Engineering, Rutgers University
[9]Department of Computer Science, Cardiff University
[10]Department of Computer Science, University of Minnesota

It is generally accepted that the ability to develop large-scale distributed applications has lagged seriously behind other developments in cyberinfrastructure. In this manuscript we provide insight into how such applications have been developed and an understanding of why developing applications for distributed infrastructure is hard. Our approach is unique in the sense that it is centered around half-a-dozen actively used scientific applications; these scientific applications are representative of the characteristics, requirements, as well as the challenges of the bulk of current distributed applications on production cyberinfrastructure (such as the US TeraGrid). We provide a novel and comprehensive analysis of such distributed scientific applications. Specifically, we survey existing models, and methods for large-scale distributed applications, and identify commonalities, recurring structures, patterns, and abstractions. We find that there are many ad-hoc solutions employed to develop and execute distributed applications, which results in a lack of generality and the inability of distributed applications to be extensible and independent of infrastructure details. In our analysis, we introduce the notion of application vectors – a novel way of understanding the structure of distributed applications. Important contributions of this paper include identifying patterns that are derived from a wide range of real distributed applications, as well as an integrated approach to analyzing applications, programming systems, and patterns, resulting in the ability to provide a critical assessment of the current practice of developing, deploying and executing distributed applications. Gaps and omissions in the state of the art are identified, and directions for future research are outlined.

Categories and Subject Descriptors: J.2 [**Computer Applications**]: Physical Sciences and Engineering

General Terms: A.1 INTRODUCTORY AND SURVEY, C.2.4 DISTRIBUTED SYSTEMS, D.1.3 Concurrent Programming

Additional Key Words and Phrases: Distributed Applications, Distributed Systems, Cyberinfrastructure, Abstractions, Scientific Applications

1. INTRODUCTION, DEFINITIONS AND OUTLINE

The process of developing and deploying large-scale distributed applications presents a critical and challenging agenda for researchers and developers working at the intersection of computer science, computational science, and a diverse range of application areas. In this paper, we review the state of the art and outline, through a gap analysis, our vision for the future. Our analysis is driven by examining the features and properties of a number of active scientific applications. Specifically, we identify and encapsulate recurring patterns within these applications. Before presenting the details of our analysis, we clarify a number of terms and set the context in the discussion below. The focus in this paper is on computational science and the use of distributed computing in computational science and this influences our definitions and discussion.

1.1 Context

Even though individual computers are becoming more powerful, there remains and will remain a need for aggregating *distributed* computational resources for scientific computing. In the simplest case, the demand for computing power at any time may exceed the capacity of individual systems that are available, requiring the coupling of physically distributed resources. Alternatively, higher throughput may be achieved by aggregating resources, or there may be a need to use specialized hardware in conjunction with general purpose computing units. Similarly, application components require specific hardware, may need to store large data-sets across multiple resources, or may need to compute near data that is too large to be efficiently transferred. Finally, distributed computing may also be required to facilitate collaborations between physically separated groups.

However, despite the need, there is a both a perceived and genuine lack of distributed scientific computing applications that can seamlessly utilize distributed infrastructures in an extensible and scalable fashion. The reasons for this exist at several levels. We believe that at the root of the problem is the fact that developing large-scale distributed applications is fundamentally a difficult process. Commonly acceptable and widely used models and abstractions remain elusive. Instead, many ad-hoc solutions are used by application developers. The range of proposed tools, programming systems and environments is bewildering large, making integration, extensibility and interoperability difficult.

Against this backdrop, the set of distributed infrastructure available to scientists continues to evolve, both in terms of their scale and capabilities as well as their complexity. Support for, and investments in, legacy applications need to be preserved, while at the same time facilitating the development of novel and architecturally different applications for new and evolving environments, such as clouds. Whereas deployment and execution details should not complicate development, they should not be disjoint from the development process either, i.e., tools in support of deployment and execution of applications should be cognizant of the approaches employed to develop applications.

The modifier *large-scale* can be applied to several aspects. For data-intensive applications, large scales can arise when the amount of data to be analyzed is copious, or if the data generated is so large that it mandates real-time processing as intermediate storage is not feasible (e.g., Large Hadron Collider (LHC) [Ellis 2007]). Furthermore, data may be stored or generated in a large-scale physically distributed fashion (e.g., Low Frequency Array (LOFAR) [Zaroubi and Silk 2005] or the envisioned SKA [Carilli and Rawlings 2004] distributed radio telescopes), imposing high-volume, wide-area data transfers. Alternatively, it may be that the amount of computation required is large-scale, for example, requiring billions of independent tasks to be completed in a limited window of time; such large-scale applications require the coupling of physically distributed resources. Large-scale is also used to represent number of users, number of resources, and geographical distribution at scale. There exists a particular interest in large-scale distributed applications, as these reinforce the need for distributed computing, as well as serving as a reminder of the extreme challenges involved in designing and programming distributed applications.

In general, for large-scale distributed systems, issues of scalability, heterogeneity, fault-tolerance and security prevail. In addition to these non-functional features of distributed systems, the need to manage application execution, possibly across administrative domains, and in heterogeneous environments with variable deployment support, is an important characteristic. In traditional high-performance computing, typically peak performance is paramount, and while peak performance remains critical for distributed systems, there are additional performance metrics of significance. For example, in many parameter sweep applications that represent common, simple and prototypical application candidates for distributed usage, a requirement is to sustain throughput during execution; thus there is an emphasis on average performance over long-periods. Large-scale distributed science applications must maintain performance metrics such as throughput, despite the presence of failures, heterogeneity, and variable deployments.

The broad set of performance metrics for distributed applications reflects the typically broad requirements and objectives for distributed applications. Consistent with this is the fact that the execution modes and environments for distributed applications are important characteristics that often determine the requirements, performance, and scalability of the application; this will be a recurring theme through this paper. For example, an application kernel such as a molecular dynamics simulator will be packaged and tooled differently based upon the expected execution modes, and whether they are local or distributed. Additionally, the mechanism of distributed control of the application will be influenced by the specifics of the execution environment and capabilities offered therein.

The distributed science and engineering applications that we discuss in this paper are mostly derived from the e-Science or Grid community of applications, which in turn emphasize traditional high performance computing (HPC) application that have been modified to utilize distributed resources. The complexity of developing applications for such large-scale problems stems in part from combining the challenges inherent in HPC and large-scale distributed systems. Additionally, as the scale of operation increases, the complexity of developing and executing distributed applications increases both quantitatively and in qualitatively newer ways. The reasons for multiple and concurrent resource usage are varied and often application specific, but in general, the applications analyzed are resource intensive and thus not effectively or efficiently solvable on a single machine. Although many of the application characteristics will be reminiscent of transactional and enterprise applications, we will focus on science and engineering applications that have been executed on general purpose and shared production grid-infrastructure, such as the US TeraGrid [TeraGrid] and European EGEE/EGI [European Grid Initiative], and have not been tied to a specific execution environment. Striving for such extensibility and portability is an important functional requirement and thus a design constraint. Additionally, the science and engineering applications considered are typically used in single-user mode, i.e., concurrent usage of the same instance by multiple users is highly-constrained.

Before moving to provide definitions of terms used and the scope of the paper, it is also important to mention some application characteristics that do not typically influence the design and operating conditions: the need for anything beyond the most elementary security and privacy, as well as the requirement of reliability and QoS have not been first-order concerns and thus they have not been specific determinants of execution environments or imposed constraints in the deployment.

1.2 Definitions and Concepts

This subsection defines the fundamental concepts used in the paper. Many terms have previously been defined/used in different contexts and with multiple meanings, and as a result, it is important to state how they are used in this paper. We keep the formalism and definitions to a minimum, and provide a common and consistent set of terms and definitions to support the paper. The key concepts of abstractions and patterns, and their relationship to distributed computational science applications are as shown in Figure 1.

Distributed Applications:. Distributed applications are those that (a) need multiple resources, or (b) would benefit from the use of multiple resources; examples of how they could benefit include, increased peak performance, throughput, reduced time to solution or reliability.

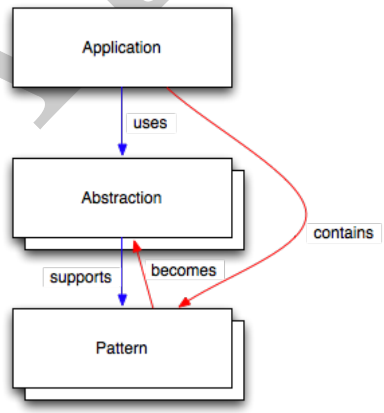

Fig. 1. Defining the key concepts used in the paper. Applications contain patterns, that can be abstracted. Once this has happened, the abstractions that now support the patterns, can be used to both reason about and develop new applications.

Pattern:. Patterns are formalizations of commonly occurring modes of computation, composition, and/or resource usage. Note that the use of the term patterns here is different from its use by the software engineering community. Patterns here refer to recurring requirements and usage, rather than just reusable solutions.

Patterns can be discerned at each of the three stages of development, deployment and execution for distributed applications. However, patterns often can not be pigeon-holed into a specific category, but may have elements belonging to different categories. MapReduce [Dean and Ghemawat 2008] is an interesting example: although it is primarily a programming pattern, depending upon the underlying infrastructure and support, it can also be classified as an execution pattern.

Abstraction:. An abstraction is any process, mechanism or infrastructure to support a commonly occurring usage (e.g., computation, communication, and/or composition). For example, **task**, is an abstraction for the frequently occurring concept of an encapsulated and self-contained piece of work. It is often the case that an abstraction can exist at multiple levels. For example, **task** can either be a simple stand-alone unit of computation or a component in a longer computation. A programming abstraction can be viewed as a way of supporting or implementing a pattern at a high-level within a programming system.

As mentioned, deployment and execution environments for distributed applications are heterogeneous and dynamic; thus there is a greater need, scope for, and possibility of increased impact of patterns and abstractions in the deployment and execution of distributed applications. For example, many distributed applications are developed to utilize the Master-Worker pattern; co-scheduling is a common example of a deployment pattern, whilst hierarchical job execution is an interesting example of an execution pattern.

In defining abstractions and patterns, we face a chicken-and-egg problem: abstractions and patterns are derived from observations of recurring concepts and themes that are used by applications. Applications can, however, only use the patterns that have thus been observed earlier, or abstractions that are provided by some implementation. So, where do new patterns and abstractions come from?

For programming abstractions, one answer is that application programmers tend to encapsulate recurring pieces of code for easier reuse. As these pieces are reused more and more often, they are generalized, with the abstractions becoming more uniform, and they are often collected in libraries of related code snippets. This is, in fact, often the seeding point for new programming abstractions or possibly models. Note that they need, however, to be compared and potentially merged with similar ‘approaches’ from other communities/groups/programmers, and they may be standardized during that process. A good example of this is the history of the MPI standard and the message passing programming model.

Beyond just programming, applications that are developed or deployed in a new area generally take advantage of previous work in other areas. Once a set of applications have been developed, one may look for concepts and themes across multiple applications, and define patterns based on these recurring elements. Abstractions then can generalize these patterns, so that the abstractions and the patterns they support can be used by new applications. By this process, the work in an initial set of applications can influence the development of a much larger set of applications.

1.3 Scope of the Paper and Related Work

The main aim of this paper is to explore and provide insight into the type of abstractions that exist, as well as those that should be developed so as to better support the effective development, deployment, and execution of large-scale (scientific) high-performance applications on a broad range of infrastructure. The advantages of such abstractions vary from better utilization of underlying resources to the easier deployment of applications, as well as lowered development effort.

As this paper relates to a number of themes in application development and distributed infrastructure, we attempt to better scope this work by relating it to real-world applications in scientific computing, along with an integrated analysis of application development (to include the tools and programming systems used) and their usage on distributed infrastructure. Consequently, our discussion of related work presented below is aligned with this focus.

The categorization of programming models and systems in this paper was initially undertaken without reference to any particular existing effort, but was later compared with published work of Lee & Talia [Lee and Talia 2003], Parashar &

Browne [Parashar and Browne 2005], Soh [Soh et al. 2007], and Rabhi [Rabhi and Gorlatch 2003]. It was found to generally agree with these works, with some differences that were well-understood, for example, previous classification attempts have been based around technology and have not considered the fundamental characteristics of applications.

Lee & Talia [Lee and Talia 2003] discusses properties and capabilities of grid programming *tools*. Their work begins with a discussion of the "fundamentals" of programming models and properties of parallel and distributed programming. There is a discussion of "programming issues" (e.g., performance and configuration management), however there is no discussion of these issues in the context of specific applications. Parashar & Browne [Parashar and Browne 2005] is primarily an analysis of Grid systems, with some discussion of tools/languages and environments. The treatment of programming models (communication models and frameworks, distributed object models, component models, and service models) is not related to any specific application classes. Section 2 of Soh [Soh et al. 2007] has a brief discussion of grid programming approaches; it focuses on grid programming models but also has a brief mention of resource composition and program decomposition. It has some mapping between the programming model and the environment, but does not provide any specific application (or application class).

Rabhi [Rabhi and Gorlatch 2003] is primarily focused on design patterns and skeletons as design abstractions. The editors motivate the book by the observation that, when faced with developing a new system, it is rare for a solution to be developed from scratch. Often the solution to a previous closely related problem is adapted to the current context. They differentiate between a skeleton and a design pattern, and focus on the use of skeletons and patterns, primarily to support design, but also to identify how such approaches can be used to support software development. The book shares may common themes with the overall aims of our work, with the key difference being the lack of emphasis on application development. Thus this paper builds upon important work done by previous researchers, but makes an important and logical progression by providing a comprehensive overview of the approaches to the development (programming), deployment and execution (usage) in the context of real-distributed applications in science and engineering.

1.4 Structure of the Paper

We begin this paper (Section 2) by discussing six representative applications that we believe capture the major characteristics, challenges and rationale behind large-scale distributed scientific applications. Each application discussion is structured around the following points: (i) Application overview; (ii) Why the application needs distributed infrastructure; (iii) How the application utilizes distributed infrastructure, and (iv) Challenges and opportunities for better utilization of distributed infrastructure. Later in the section, we introduce "application vectors", to describe the distributed characteristics of the applications, and discuss how the values chosen for these vectors influence the applications' design and development.

The detailed discussion of applications sets the stage for an analysis of the patterns that are integral to these applications in Section 3. This section introduces and discusses a range of application and system patterns, and classifies them in a way that is consistent with the vectors developed in Section 2. Although motivated by the applications discussed, the patterns are not confined to them. Section 3 provides a framework in which to analyze distributed applications and ultimately to shed light on ways that a pattern-oriented approach to distributed applications and systems can facilitate the development and deployment of large-scale scientific applications.

Section 4 discusses programming models and systems categorized by their primary focus, either composition, components, or communication models, and provides specific examples of the tools used to develop the applications. This section then sheds light on the questions of how the models are "offered to the application developers" as-is, as well as "what characteristics are not well supported". This is centered on the observation that many developers prefer to develop their own tools and programming systems. The analysis undertaken and the insight gained from Sections 2-4 are synthesized in Section 5, which also revisits several questions raised in earlier sections.

2. APPLICATIONS

High performance applications have explored distributed platforms for well over a decade. For example, the early High Performance Distributed Computing (HPDC) conferences featured several scientific and engineering applications targeting distributed environments: HPDC-1 (1992) featured applications in climate modeling [Mechoso, C. R. et al. 1992] and biological [Grimshaw, A. S. et al. 1992] domains, while HPDC-2 (1993) featured applications in engineering design [Manke et al. 1993] and computational physics [Coddington 1993]. Even though distributed and parallel applications are very different, we will frequently contrast the former in relation to the latter. We believe this is consistent with the historical development and the logical progression of many large-scale distributed scientific applications. Furthermore, doing so provides an opportunity to understand distributed scientific applications better, as the theory and formalism of distributed scientific applications is still not as well developed as the corresponding theory and formalism of parallel applications.

In traditional parallel computing, one often assumes: (i) there are usually no faults, and if there is a fault, everything will stop (fail-stop behavior); (ii) performance is critical, and in most cases, performance is measured by time-to-solution for a single problem; (iii) delays are predictable and messages do not get lost; and (iv) there is only one application on a part of a parallel system at any given time, and that application "owns" that part of the system until it completes, and so security and privacy are not relevant concerns. In distributed computing, these assumptions are not generally valid. Applications needs to be concerned with fault tolerance and security. Resources are shared and the set of resources for a single job may be controlled by multiple organizations. Replication of tasks and data may be needed for a variety of reasons. While time-to-solution for a single job is still important, throughput of many jobs on the system may be more significant. Developers of existing and future applications have to decide both the relative importance of these distrbuted concerns, and how they should be dealt with, which includes handling them at the middleware- or application-level.

2.1 Descriptions of the Applications

We have attempted to select a representative set of example applications that we describe in this section. These applications are similar in several respects, including belonging to the domain of computational science as opposed to business or entertainment, and having non-trivial resource requirements, implying that the performance of each application instance is important, not just overall throughput. Although the applications described reflect the expertise of the authors, we believe they cover the primary classes of applications that currently use distributed infrastructure. The applications are not just HPC; they cover a broad range of distributed application types, including data-intensive, sensor, data-driven simulations, etc. Many of these applications have tried to make use of production infrastructure existing at the time they were developed. Additionally, these applications expose the capabilities required in the development, deployment, and execution stages. The description of all applications address four points: a brief overview of the application, why the application is distributed, how is the application distributed, and what are the challenges in distributing the application.

Montage [Jacob, J. C. et al. 2007] is an image processing application that is built for a variety of systems, including single computers, parallel computers, and grids. Here, the grid version is discussed. Montage takes as inputs a set of images taken by one or more telescopes and builds a mosaic that approximates the single image that would be taken by a single large telescope. The work that is done by Montage is encapsulated in a set of executables (normally about 10), most of which are run multiple times with different data sets. Each executable uses files for input and output. The dependencies between the executable runs are stored as a DAG, and this DAG is run using Pegasus [Deelman, E. et al. 2005] and DAGMan [Frey et al. 2001]. If the systems used to run the executables do not share a file system, various files have to be moved from the system on which they were generated to the system on which they are needed; this is handled by Pegasus. Montage does not have any real gain from running on a distributed system rather than on a cluster, but it uses distributed systems to scale processing capability above what is possible on a local cluster. The main challenge associated with doing this is mapping the executables to specific resources in a way that optimizes time-to-solution for building each mosaic and also makes best use of the computational and network resources, since using resources on multiple machines leads to a need for data transfers between processes that would not be necessary if executables are run on a single machine. At the Montage home site (JPL/Caltech's IPAC), a local cluster is available for typical small Montage jobs, while large jobs are farmed out to distributed resources.

NEKTAR: Distributed Tightly-Coupled Fluid Dynamics [Boghosian, B. et al. 2006]: The NEKTAR - Human Arterial Tree project is motivated by a grand challenge problem in biomechanics: simulating blood flow in the entire human arterial network. Formation of arterial disease, e.g., atherosclerotic plaques, is strongly correlated to blood flow patterns, and is observed to occur preferentially in regions of separated and recirculating flows such as arterial branches and bifurcations. Interactions of blood flow in the human body occur between different scales.

The challenge in modeling these types of interactions lies in the demand for large scale supercomputing resources to model the three-dimensional unsteady fluid dynamics within sites of interest such as arterial branches. The parallel performance of the full simulation does not scale efficiently beyond a thousand processors, hence devoting more resources to it does not result in enhanced performance.

This simulation is amenable to distributed computing because the waveform coupling between the sites of interest can be reasonably modeled by a reduced set of 1D equations that capture the cross-sectional area and sectional velocity properties. Using a hybrid approach, NEKTAR simulates detailed 3D blood flow dynamics at arterial bifurcations while the waveform coupling between bifurcations is modeled through a reduced set of 1D equations. At each time step, the 1D results provide appropriate boundary conditions for the 3D simulations, such as the flow rate and pressure. This application represents a novel decomposition method and is an interesting example of how a *traditional* tightly-coupled application can be modified into an application using multiple loosely-coupled distributed components.

The human arterial tree model contains the 55 largest arteries in the human body with 27 arterial bifurcations, requiring about 7 TB of memory. At the time of these simulations (2003-05), these requirements were well beyond the capacity of any single TeraGrid/supercomputing site at that time. As more resources become available the model can be (and needs to be) simulated at finer level of detail and granularity, thus preserving the mismatch in peak need and peak capacity.

Coupled Fusion Simulations: The DoE SciDAC CPES fusion simulation project [Klasky, S. et al. 2005] is developing an integrated predictive plasma edge simulation capability to support next-generation burning plasma experiments such as the International Thermonuclear Experimental Reactor (ITER) [ITER]. These coupled fusion simulations allow physicists to study the dynamic interaction of kinetic effects that may limit tokamak reactor performance [Cummings 2008].

The resulting data-intensive application workflows consist of multiple heterogeneous and coupled processes that need to dynamically interact and exchange data during execution. For example, codes simulating different components of the physical phenomena, verification and validation services, visualization/analytics services, monitoring services, archival services, etc. can often run independently and on-demand resulting in interaction, coordination and coupling patterns that are complex and highly dynamic. These are typically loosely coupled simulations, which could run either as separate processes on the same physical machine, or on different machines, and thus are suitable for execution on distinct and distributed resources, as opposed to a single large dedicated machine.

As an example, in the time-stepping coupled formulation, the XGC0 [Chang et al. 2004] particle-in-cell (PIC) and microscopic MHD codes (i.e., M3D [Park et al. 1999]) are used together, and after each time step, the codes exchange data. As a consequence of different formulations and domain configurations/decompositions, independent executables, data must be transformed using mesh interpolation while it is being streamed between them. A key challenge in this application comes from the need to stream data between the two running parallel simulations while satisfying strict space, time and reliability constraints. Furthermore, transforming the data in-transit presents additional challenges. The current implementation [Docan, C. et al. 2010] of the application builds on custom coupling and data streaming framework. The former provides an abstraction of a semantically specialized virtual shared space with geometry-based access semantics, and the latter provides remote direct memory access (RDMA) based asynchronous IO support and autonomic data streaming capabilities using model-based online control [Bhat, V. et al. 2007].

Asynchronous Replica-Exchange Simulations: Replica-Exchange (RE) [Swendsen and Wang 1986; Geyer and Thompson 1995] is an effective sampling algorithm proposed in various disciplines, such as in bimolecular simulations, where it allows for efficient crossing of high energy barriers that separate thermodynamically stable states. In this algorithm, several copies (replicas) of the system of interest are simulated in parallel at different temperatures using "walkers". These walkers occasionally swap temperatures based on a proposal probability, which allows them to bypass enthalpic barriers by moving to a higher temperature.

Applications based on RE tend to be computationally expensive and can benefit from the large numbers of processors and computational capacities offered by parallel and distributed computing systems. These applications are naturally decomposed into tasks that are loosely-coupled (i.e., require very infrequent exchange) thus making them good candidates for distributed environments. Reflecting the above, there are multiple implementations of RE molecular dynamics for distributed environments [Gallichio et al. 2007; Luckow et al. 2009].

The asynchronous RE formulation [Gallichio et al. 2007; Li and Parashar 2007; Zhang, L. et al. 2006] enables arbitrary walkers, executing on distributed resources, to dynamically exchange target temperatures and other information in a pair-wise manner based on an exchange probability condition that ensures detailed balance. Exchange decisions between walkers are made in a decentralized and decoupled manner, and the exchanges can occur asynchronously and in parallel. Walkers can dynamically join or leave (or fail), allowing the application to deal with the dynamism and unreliability of the distributed environment. The current implementation of this application builds on a custom coordination and messaging framework based on the JXTA P2P substrate [Gallichio et al. 2007; Li and Parashar 2007; Zhang, L. et al. 2006].

Individual replicas can be treated as independent jobs, possibly on different machines, but centrally coordinated [Luckow et al. 2009]. Additionally, existing parallel implementations of RE are typically synchronous, i.e., the exchanges occur in a centralized and synchronous manner, and only between replicas with adjacent temperatures. A fundamental challenge of implementing distributed applications in general, but highlighted by the RE-based applications, is the need for a general-purpose, efficient, and scalable (with increasing distribution) coordination mechanism that work across a range of infrastructure types.

ClimatePrediction.net [Christensen, C. et al. 2005] (CPDN) has two intrinsic computational approaches, one for generating the data, and one for analyzing the data. The approach for generating the data closely follows that of SETI@home [Anderson et al. 2002], and is in fact now built on the BOINC middleware [Anderson 2004]; because it is running a scaled down version of a model that would traditionally be run on a single HPC computer, the runtime is quite long. This has led to the construction of Trickles that are now integrated into the BOINC platform. Trickles allow distributed jobs to report back to the project server on a daily basis (or longer, for if the distributed host goes off-line for more than a day, they will report to the server when it comes back on line) with a small amount of data regarding the state of the job. This allows accurate information on the state of jobs to be collated, and problems to be detected and resolved. The other feature that this example has that was not part of SETI@home, is the concept of phases, where a client run will return a useable piece of the result part during the computation of its work unit. In the case of the basic ClimatePrediction.net experiment, this takes the form of the initialization and stabilization of the model, a period with pre-industrial CO_2 and a phase with the CO_2 levels doubled. The use of distributed computing in the data generation phase allows collaborative processing, where the ClimatePrediction.net team can run a large set of models without having to own all of the needed computing and storage.

The approach for the analysis of the data is less regular. It uses a variable number of highly distributed computing resources and data stores. Data is distributed across the data stores in a regular way, with complete results from individual models always residing on the same server. The data is typically too big to transfer to a single location, so analysis must be performed in situ. As the users are not skilled with parallel programming, some form of abstraction for the changing number of distributed resources must be provided. This is done through a data parallel workflow language, Martlet [Goodman 2007], that contains a set of abstract constructs with which to build analysis functions. Other solutions, such as Parallel Haskell [Chakravarty, M. T. T. et al. 2006; Trinder, P. et al. 2002], are able to handle the changing numbers of computational resources, but an important feature is that the number of data sources is also changing. There is definitely a gap here to be filled by further abstractions, as the current constructs supported by Martlet are just a set produced as a proof of concept, and the extension with more powerful and useful ones is not hard to envisage.

SCOOP [Bogden, P. S. et al. 2007] [Mesonet] is the primary application of SURA Coastal Ocean Observation Program, which is a computationally intensive hurricane forecasting and hindcasting system, with a somewhat unpredictable demand, that requires near real-time, if not faster than real-time, response from the infrastructure and simulation capabilities. This is to provide decision makers with best possible information for disaster mitigation efforts on the required (short) time-scales.

A real SCOOP scenario typically involves some or all of the following: When a tropical depression develops into a hurricane, operational hurricane forecasting must start quickly (potentially using preemption to take over available resources) and produce the best possible predictions of the effects of the storm, e.g., the probability that the resulting storm surge will reach a certain height. Once a hurricane forms, ensembles of parallel prediction models that ingest data from track predictions, satellites, sensors, and radar must be run. These models need to be run repeatedly when new data is available, to update predictions using the latest information. Because the availability of new data can often be predicted, reservations can be used to run the non-initial models. Given an estimate of the hurricane's path, precipitation models are then run that consider the amount of water being transported by the hurricane and estimate where it will be deposited. Flooding models are subsequently run to determine risks.

SCOOP provides an example of an application that is distributed by nature and not by design: multiple distributed sources (of data) and multiple distributed sinks of this data (distributed high-end computers). Another important reason for depending upon distributed resources lies in the economics of infrastructure provisioning: it is not economically viable to maintain a large, dedicated compute facility just for this purpose since it would have a very small "duty cycle" over the course of a year.

Currently workflow technologies, such as Pegasus/DAGMan [Deelman, E. et al. 2005; Frey et al. 2001] are used in conjunction with advanced reservation capabilities such as HARC [MacLaren 2007] and SPRUCE.

SCOOP and many other distributed computing applications have competing demands: the workflow should be as simple as possible, and ideally it should be able to be redeployed immediately on demand. Shared use of distributed resources makes the most economic sense, but it can be difficult to reserve the computational resources needed in time because the occurrence of a hurricane cannot be reliably predicted, and the computational demands may be large compared to the competing workload. Even when resources are available, the management of data ingest from distributed sources, the coupling of models, and the distribution of data products to decision makers and first responders becomes a challenge for dynamically provisioned resources.

The effective execution of SCOOP and other dynamically data-driven applications remains a significant challenge. For example, it is still significantly difficult to adequately provision this computing effort in a timely fashion, there is an absence of general-purpose and extensible abstractions for coupling dynamically distributed data to multiple simulations, and there are many (single) points of potential failure which impede the successful completion of the workflow.

2.2 Application Vectors

Analyzing the characteristics of the distributed applications suggests that there are important issues in the development, deployment and execution that should be considered. There are three main issues of importance to be considered when developing a distributed application: The first is determining what is distributed, or what the pieces of the distributed application are. The next is determining how the pieces interact. The third is determining the flow of control and information between the pieces. In addition to the above three factors, one needs to determine what is needed to actually deploy the application and make it run. Overall, these four factors influence the way in which distributed applications are developed and deployed, and can be used to define a set of *application vectors*.

Analyzing applications using these vectors facilitates an understanding of why they use distributed infrastructure, at least in part (for there are social and economic pressures as well as technical reasons), and also possibly helps judge their suitability for use in a distributed mode. For each application, one then can ask what are the characteristics that determine how it is developed and deployed on distributed infrastructure, i.e., map the application characteristics onto these vectors.

Considering these applications in a multi-dimensional space, one can use the four

Application Example	**Execution Unit**	**Communication (Data Exchange)**	**Coordination**	**Execution Environment**
Montage	Multiple sequential and parallel executables	Files	Dataflow (DAG)	Dynamic process creation, workflow execution, file transfer
NEKTAR	Multiple concurrent instances of single executable	Messages	SPMD	MPI, co-scheduling
Coupled Fusion Simulation	Multiple concurrent parallel executables	Stream-based	Dataflow	Co-scheduling, data streaming, async. data I/O
Asynchronous Replica-Exchange	Multiple sequential and/or parallel executables	Pub/sub	Dataflow and events	Decoupled coordination and messaging, dynamic task generation
Climate-Prediction.net (generation)	Multiple sequential executables, distributed data stores	Files and messages	Master/ worker, events	AtHome (BOINC)
Climate-Prediction.net (analysis)	A sequential executable, multiple sequential or parallel executables	Files and messages	Dataflow (Forest)	Dynamic process creation, workflow execution
SCOOP	Multiple different parallel executables	Files and messages	Dataflow	Preemptive scheduling, reservations

Table I. Characteristics of the set of applications arranged according to the identified vectors

values to develop a vector space. Note that using a traditional notion of such a space, each application would have a single value on each vector, and if two applications had the same values on all vectors, they would also have the same distributed characteristics and structure – meaning that the applications are the same in terms of how they are distributed. The vectors here are similar to axes, in that each application has a value for each vector, but the values are members of a set that cannot easily be compared to a ranked set of reals or integers. In this section, we explain the specific vectors we have chosen to use, and define the value of each vector for each application.

Formally stated, the vectors we use (listed in the Table I) are: Execution Unit, Communication (Data Exchange), Coordination, and Execution Environment. The first vector is *execution unit*. This refers to the set of pieces of the application that are distributed, or the elements of the application that are decomposed over the distributed system. In all cases, these are executables; sometimes they are models that could run independently, such as a data processing step that reads some input files and produces some output files, and other times they require other elements to communicate with, such as a master in a master-worker model, or a data-capture unit that is streaming data to another piece of the application. Generally, the execution units can be a combination of one or more sequential or parallel executables, and one or more data stores.

The next vector is *communication (data exchange)*. This defines the data flow between the executions units. Data can be exchanged by messages (point-to-point, all-to-all, one-to-all, all-to-one, or group-to-group), files (written by one execution unit and then read by another), stream (potentially unicast or multicast), publish/subscribe, data reduction (a subset of messaging), or through shared data.

The third vector is *coordination*. This describes how the the execution units are coordinated. Possible values for this vector include dataflow, control flow, SPMD (where the control flow is implicit in all copies of the single program), master-worker (the work done by the workers is controlled by the master), or events (runtime events cause different execution units to become active). In the case of data flow and control flow, the overall dependencies may be described as a DAG or a forest.

The last vector is *execution environment*. This vector captures what is needed for the application to actually execute. It often includes the requirements for instantiating and starting the execution units (which is also referred to as *deployment*) and may include the requirements for transferring data between execution units as well as other runtime issues. Examples of values of this vector include: dynamic process/task creation, workflow execution, file transfer, web services, messaging (MPI), co-scheduling, data streaming, asynchronous data I/O, decoupled coordination support, dynamic resource and service discovery, decoupled coordination and messaging, decoupled data sharing, preemption, At-Home (meaning that a master is made aware of coming and going of workers that it does not control), and high-throughput (a master that launches and controls a set of workers).

Table I shows that each of the example applications can be described by such a set of *vectors* derived from the multi-dimensional space described above. It is not very surprising that most of the applications are composed of multiple but coupled units; this appears to be a widely used method of developing distributed applications. Typically distributed applications are concerned with composition of units into a single application, while parallel application are generally concerned with decomposing a single application into parallel pieces. Common mechanisms for communication are messages, files and pub/sub, which could either be via messages or other forms of notification. A large set of the applications use dataflow for coordination.

The execution environments of the application differ quite a bit, i.e., there is a broad range of values in this column compared to the range of values in the previous three columns; as alluded to, this appears to be an important and distinguishing feature of distributed applications. In general, distributed applications often have different values for some or all axes, whereas parallel applications only have distinct values on the communication axis; MPI applications normally have a single value on each of the execution unit, coordination, and execution environment axes.

We believe that applications can be classified based on the values of their vectors used to understand their main characteristics. In case of the "execution unit" vector, applications have either single or multiple units that can be either homogenous or heterogenous. Although communication can take place using a variety of mechanisms, such as messaging, file-exchange, shared-data, a better description for this vector is the frequency of exchange rather than the type.

Similarly, it is not the specific value of the third vector (coordination) that is important, but the relative flexibility in coordination. It is important to mention that the overloaded terms "loosely-coupled" and "tightly-coupled" in the distributed

Application Example	**Execution Unit**	**Communication**	**Coordination**	**Execution Environment**
Montage	-	-	DAGMan	Pegasus (abstract workflow); Condor DAGMan (for enactment)
NEKTAR	MPICH-G2	MPICH-G2	MPICH-G2	Distributed MPI (e.g., MPICH-G2), co-scheduling (e.g., HARC for advanced reservation)
Coupled Fusion Simulation	MPI/ OpenMP	PORTALS/ VERBS (over RDMA)	Data I/O: DART/ADIOS, Coupling: SIENE (over sockets), Kepler	DMA-based data extraction, data streaming tool, puts (gets) on remote memory
Asynch. Replica-Exchange Molecular Dynamics	MPI/ pthreads	Meteor (pub/sub notification over JXTA pipes)	Distr. shared space: COMET (over JXTA)	JXTA-based services, COMET (decentralized tuple space implementation) and Meteor
Climate-Prediction.net (generation)	-	BOINC (over http)	BOINC	AtHome (BOINC)
Climate-Prediction.net (analysis)	-	-	Martlet	Workflow execution through Martlet
SCOOP	-	LDM [LDM]	-	SPRUCE, HARC

Table II. The specific tools used to support and execute the example applications

computing sense are somewhat different from their typical parallel usages. In the latter case, they are used to refer to tolerance to latency, whereas for distributed applications they typically refer to the degree of synchronization and thereby the flexibility with which they can be distributed, scheduled and/or the dependencies between the execution units can be identified. By looking at coordination mechanisms between the different execution units, which can be either continuous or discrete, additional, orthogonal classes (e.g., event-driven, interactive) can be derived. Note that the application classes that emerge by looking at the values of the different vectors result in a classification, which is often close to the way computational and domain scientists would describe their application; this reinforces the concept of application vectors.

It is important to highlight that some application characteristics are not necessarily amenable to the classification in Table I, yet still play an important role in determining the development tools utilized and deployment mechanisms employed. For example, there are applications that need fault-tolerance of several types (task, messages) and at different levels (application, system), or applications that need in-transit data-transformation. Some of these “other characteristics” are functional and some are non-functional, but in both cases, they are important and are often delivered/engineered in ways unique to a specific application’s requirements.

2.3 Application Development Tools and Execution Environments

For a given application, with a given set of application vector values, specific tools need to be used to implement the application. Once the application developer has determined the values for each vector for the application being developed, she or he then needs to choose specific tools to implement these values. Table II lists the tools that have been used to decompose the applications into execution units, support applications and execution units, and to implement the communication and coordination between the execution units in the studied applications. Additionally, the last column of the table lists the tools that are needed to deploy and execute the application, which sometimes overlaps with the previous columns, as development and deployment are not separate concerns for many tool developers.

The first two rows of Table II illustrate some of the issues regarding tools that are used for more than one vector, and how deployment tools can, but do not have to, overlap with development tools. In case of Montage, Condor DAGMan is used to define the coordination between execution units, as well as to actually execute them (working with Pegasus). However, no tools are used to build the execution units (beyond compilers), and there is no communication tool as the communication is implemented by reading and writing files. In the NeKTAR application, MPICH-G2 [Karonis et al. 2003] (MPIg [Manos, S. et al. 2008]) are used for all the axes of development, as well as for deployment, although in order to use multiple resources simultaneously for the same application, co-scheduling is required, which is implemented using HARC [MacLaren 2007] for this application.

3. DISTRIBUTED APPLICATIONS AND SYSTEMS PATTERNS

The aim of this section is to explore the ways in which a pattern-oriented [Mattson et al. 2004] and abstractions-based development model might enhance the distributed programmer's ability to produce good (efficient, portable, robust) applications. We will do so by examining our application set in order to isolate candidate patterns as they occur "in the wild". We are particularly interested in finding patterns in which *distributed* issues or characteristics play a significant role, and in which the pattern encapsulates a concurrent *composition* of more primitive actions.

Patterns in the parallel software engineering sense [Mattson et al. 2004] can be characterized as semi-formal descriptions of "good solutions to recurring problems", collected and described in a "pattern language" through which common wisdom can be transmitted between engineers. This is quite a soft concept, leaving considerable scope for both specification and implementation detail in the hands of the expert programmer. Sometimes, more formal specification and support is provided, through "generic patterns of computation and interaction ... abstracted and provided as a programmer's toolkit" [Cole 2004], in the form of a concrete library, template or problem solving environment of some kind. For our purposes, as defined in Section 1.2, patterns are frequently occurring compositions of simpler operations, which may be suitable candidates for abstraction, both conceptually, and ultimately concretely in the form of tools. The programmer becomes responsible for selection, instantiation and specialization of the abstraction (via a concrete tool) with application specific details.

Patterns allow good practices to be identified and shared across application domains; they are generally defined in an application-independent manner. Abstractions that encapsulate patterns are aimed at capturing generic attributes that may be further instantiated, parameterized and/or refined. Patterns and abstractions are important for distributed computing applications, which generally need to operate in dynamic and heterogeneous environments. Implementing abstractions in tools will ease the task of distributed application developers.

Interest in patterns within the parallel programming process is increasing. For example, [Asanovic, K. et al. 2006] proposes a collection of "motifs" as a basis for evaluation of new parallel programming models, where each motif is "an algorithmic method that captures a pattern of computation and communication." Although the impact of patterns in parallel computing is still being determined, due to the unique concerns of distributed applications, we believe that patterns and abstractions will certainly have benefits for distributed systems and applications. A key aspect of distributed application development (in contrast to conventional parallel programming) is that handling characteristics of the environment in which an application is deployed and executed can become as challenging as developing the core application itself, or may even be an inherent component of the problem definition. For example, in an "AtHome" application [Anderson et al. 2002], the programmer may decide that the environment, consisting of untrusted and unreliable workers returning results of questionable provenance, requires a replication and consensus reaching mechanism on top of the core distribution, analysis and return facility.

As our analysis will show, there exist many applications and tools that either encounter, or address "similar" problems. They do so in ways that are often unique, even though there is no reason, barring possible historical ones, that they could not do so in a common manner. This suggests that a structured approach that encourages abstractions that support widely used application and system patterns will yield rich dividends; this has been shown to be true and effective for workflows – at least at the system level. One of the most widely cited example of this is the YAWL [van der Aalst, W. M. P. et al. 2004] system, which supports three main types of patterns: control-flow, data and resource [YAWL Patterns]. These patterns have been adopted within a number of workflow enactment engines and systems – such as WebSphere from IBM, Meteor, etc.

In reviewing the collection of applications from Section 2, we have identified a number of patterns. Most of these are already recognized in the patterns literature, so it is reassuring to see them appear in real applications. Our collection includes patterns that span a range of positions: from those which correspond to existing parallel programming patterns, through some whose description encompasses issues that are driven by distributed concerns, through to those that deal exclusively with challenges posed by deploying an application in a distributed environment. In this section we will examine these patterns, their application and support along a number of dimensions. For added value, we have included a small number of application patterns that, though not appearing in our chosen application set, nevertheless seem to be important in the wider literature.

We begin by introducing the patterns themselves, categorized according to their predominant concern, in the sense of addressing *coordination* or *deployment* issues. (Of course, this is not to imply that each pattern addresses only one issue

Coordination	Deployment
Master-Worker (TF, BoT)	Replication
All-Pairs	Co-allocation
Data Processing Pipeline	Consensus
MapReduce	Brokering
AtHome	
Pub-Sub	
Stream	

Table III. Categorizing patterns by predominant concern. While not mutually exclusive, coordination patterns focus on the application-level logical relationships between units of execution, while deployment patterns concern issues arising from the distributed infrastructure..

exclusively.) Table III summarizes this information, while Table IV captures the relationships between patterns and applications. Section 3.3 focuses on concrete support for the abstraction of patterns.

3.1 Coordination Patterns

Our coordination patterns are predominantly concerned with managing the relationships between concurrent activities, usually including aspects of both computation and communication.

Master/Worker. In *Master/Worker*, a single master controls a set of workers. Conceptually, the master repeatedly directs a worker to execute a task and gets the result from that worker. Workers do not communicate with each other. The only ordering that may exist is in how the master chooses to activate workers for specific tasks. Often, many workers work on the same task with different input parameters, and the master assembles the outputs into a concatenated or derived product.

Task Farm. *Task Farm* is a well-known pattern in the parallel programming literature, in which a static collection of independent instances of the same problem must be executed by a pool of workers. The result is simply the corresponding collection of results. The programmer provides a definition of the worker function, and a collection of inputs. The implementation allocates instances to workers dynamically, to adjust to variation in computational requirements of the instances. Task Farm can be seen as a more specialized instance of *Master/Worker*, in which there is no (application level) master code. Most parameter sweep applications employ Task Farms. In the distributed context, an implementation must deal with resource failure, and consequent re-computation of lost tasks. Systems such as Condor [Condor] and Apples [Berman, F. et al. 2003] are, in effect, instantiations of Task Farm.

Bag of Tasks. The *Bag of Tasks* pattern [Andrews 2000] is related to *Task Farm*, but distinguished by the fact that new tasks may be created and added to the "bag" dynamically, by any process, as a result of task executions. It is classically implemented in a *Master/Worker* style, where the Master coordinates access to the *bag*, though more distributed implementations are also possible.

All-Pairs. The *All-Pairs* pattern [Moretti et al. 2008] abstracts computations in which a single function must be applied independently to all members of the crossproduct of a data set with itself (i.e., to all possible pairs of items from the set), generating a set of results. It could be implemented as an instance of *Task Farm*, since all required computations are known in advance and are independent.

Data Processing Pipeline. The key concepts of this pattern are a set of data that can be processed in parallel, and some number of stages (perhaps one) of processing. Each stage of processing for a subset of the data can be undertaken as a single task. The dependencies between the stages and the decomposition of the data lead to a set of dependencies, that can be expressed as a DAG, and that need to be enforced. When one task finishes processing some data, another task can be started once the data needed is transferred as the output of the first task to be the input of the second task. Often, finishing one task will enable multiple tasks to start. Example applications use tools such as DAGMan [Frey et al. 2001] to execute the workflow.

MapReduce. MapReduce [Dean and Ghemawat 2008] captures a class of data-intensive applications in which computation occurs in two phases. In the first, a common function is applied concurrently and independently to all members of the data set. In the second, the results from the first are partitioned on a chosen key field, and members of each partition are collapsed together (or "reduced") by application of an associative combination function, generating a set of results, one per partition. The canonical implementation of *MapReduce* composes two Task Farms, one for the Map phase, one for the Reduce, augmented by the use of *Replication* (see below) to introduce fault-tolerance.

AtHome. AtHome can be seen as a specialized instance of *Master/Worker*, in which computation of each problem instance is deliberately replicated, and in which there may be explicit application level "master" code that may further manipulate returned results. It also exploits *Replication*. Here, in addition to conventional fault-tolerance, replication allows the application to become more robust in the face of malicious or inadvertent miscalculation by workers. The degree of replication becomes a parameter to the pattern, as does the mechanism by which consensus between replicas is obtained. For example, an application may require that a majority of copies concur, or that at least two concur, and so on. For full flexibility, consensus may be user-programmable. AtHome arises naturally in the context of Volunteer Computing [Sarmenta and Hirano 1999].

Two final coordination patterns focus solely on the mechanism for data exchange between activities, rather than the dependencies between units of execution.

Publish/Subscribe. The *Publish/Subscribe* pattern captures a relatively unstructured, event-driven model in which messages are characterized into classes and "subscribers" express interest in one or more classes, only receiving messages that are of interest, without knowledge of what (if any) "publishers" exist. Thus, publishers and subscribers are decoupled, enhancing modularity over schemes in which the relationship is embedded explicitly in code. Example publish/subscribe systems include RGMA [RGMA], Conoise-G [Concoise].

Stream. Streaming captures communication in which a unidirectional flow of information between agents is processed as it arrives (in other words, there is interleaving and/or overlap of communication with computation). It underlies the *Data Processing Pipeline* pattern, but with fewer constraints; we can stream information through more complex structures than the simple linear arrangements of DPPs.

3.2 Deployment Patterns

Deployment patterns have a strong focus on dealing with aspects of applications that are largely or even solely provoked by distribution and reliability issues.

Replication. The *Replication* pattern captures the principle that it may sometimes be appropriate to replicate computations or data, in ways that would be unnecessary in a homogenous, non-distributed environment. The reasons vary, but can include untrustworthiness of individual instances or fault-tolerance.

Co-allocation. Applications vary in the extent to which their components are expected or required to be active concurrently. For example, a typical DAG-based loosely-coupled computation with files as the communication medium does not require genuine concurrency (although it may be advantageous). In contrast, concurrency is essential in a more tightly coupled application with communication via MPI – the application will fail without it. The process of ensuring (often through reservation) that suitable resources will be available concurrently is known as *co-allocation*, and the pattern of negotiating to satisfy its requirement is common to all such applications, independent of the application patterns that will exploit it.

Consensus. The *Consensus* pattern arises with *Master/Worker* applications, when the Master must make some progress decision, based on results returned from some known subset of the workers. The decision is programmable, and must allow considerable flexibility to short-circuit, or otherwise discard the contributions of some or all workers. For example, within *AtHome*, applications must decide when to consider a replicated problem “solved”. This could be when two agreeing solutions have been returned, or a majority, or a unanimous complete set; the definition of “agreement” may be application specific. The pattern provides an implementation of the supporting marshaling of replicas. Thus, we can view *Consensus* both as a component of *AtHome* and also as a worthwhile pattern in its own right.

Brokers. If clients may have a choice of service providers then brokers enable them to select between servers. Both Client/Server and brokered servers realize the Service Oriented Architectures (SOAs) discussed in Section 4.1.2. Well-known broker-based systems include the Common Object Request Broker (CORBA) [OMG 1998][OMG 2002] and CCA [Bernholdt, D. E. et al. 2006], also discussed in Section 4.1.2.

3.3 Supporting the Abstraction of Patterns with Concrete Tools

Table IV demonstrates that a pattern-oriented approach can be used to conceptualize, develop and deploy applications. Interestingly, a significant number of tools and libraries support the patterns we have identified, as summarized in Table V. We observe that Master/Worker and its relatives are best supported, and we believe, most widely deployed. This probably reflects both the fact that these are quite generic patterns, and that such applications, built from independent, computationally intensive tasks, lend themselves well to execution on contemporary

Application Example	Coordination	Deployment
Montage	TaskFarm, Data Processing Pipeline	-
NEKTAR	-	Co-allocation
Coupled Fusion Simulation	Stream	Co-allocation
Async RE	Pub/Sub	Replication
Climate-Prediction (generation)	Master/Worker, AtHome	Consensus
Climate-Prediction (analysis)	MapReduce	-
SCOOP	Master/Worker, Data Processing Pipeline	-

Table IV. Applications and their pattern usage. A "-" indicates that no pattern can be identified, not that the application doesn't have any communication, coordination, or deployment.

Pattern	Tools That Support the Pattern
Master/Worker-TaskFarm	Aneka, Nimrod, Condor, Symphony, SGE, HPCS
Master/Worker-BagofTasks	Comet-G, TaskSpace, Condor, TSpaces
All-Pairs	All-Pairs
Data Processing Pipeline	Pegasus/DAGMan
MapReduce	Hadoop, Twister, Pydoop
AtHome	BOINC
Pub-Sub	Flaps, Meteor, Narada, Gryphon, Sienna
Stream	DART, DataTurbine
Replication	Giggle, Storm, BitDew, BOINC
Co-allocation	HARC, GUR
Consensus	BOINC, Chubby, ZooKeeper
Brokers	GridBus, Condor matchmaker

Table V. Tools and libraries that support patterns identified in Sections 3.1 and 3.2

distributed systems. Not surprisingly, MapReduce has received a lot of attention recently for these reasons and the fact that many data-intensive applications can be formulated using this pattern; the existence of fault-tolerant and simple to use implementations has also contributed to its uptake.

4. PROGRAMMING MODELS AND SYSTEMS

The objective of this section is to investigate how dominant programming concepts, models and systems used by the scientific and engineering community have been extended or adapted to address the unique requirements of large-scale distributed applications. Although driven the the set of applications presented in Section 2 and motivated by a desire to understand the programming models and systems, this section is not constrained by the application set. The first part of this section provides an informal categorization of programming models, the first two categories of which – composition and components/service models, primarily support different types of coordination; the third category discusses a number of programming models to support communication. The second part discusses programming systems and their use in distributed computing for science and engineering applications.

4.1 A Categorization of Programming Models

This subsection defines and describes the set of programming models and systems that we believe capture the bulk of existing work in large-scale distributed applications. The models can be broadly classified into three categories: Composition, Components (including Services), and Communication.

4.1.1 *Composition.* The first category, **Composition**, includes a number of techniques, such as: scripting, workflows, functional languages, skeletons, and compiler-based approaches (the latter may be considered as a technique for decomposition as opposed to composition). The main idea that ties these together is the concept of a user building an application out of independent units, such as subroutines or executables. Note that some members of the research community also view composition as being a subset of a formal component definition; we will compare and contrast the two in Section 4.1.2. Important issues related to composition are the ordering of the units, and the flow of data or messages between them. In many cases, the user has a graphical interface of some sort that can be used to specify ordering (the ordering is explicit). In other cases, only dependencies are specified by the user, and some automated system then decides the ordering that will be used to completely run the applications (the dependencies are explicit, but the ordering is implicit), for example, using a planning engine to support composition. Another issue is the resources on which the units are run. This may be determined by either the user (explicitly) or by an automated system (implicitly). Finally, the components themselves may either always exist (perhaps as services that are always available to be called), or they may be instantiated when needed (such as a set of executables that need to be run).

In the context of scientific computing, workflow techniques have become dominant in supporting such composition. The composition process itself can be dynamic, where portions of the workflow are executed before others and influence the structure and behavior of subsequent parts of the workflow, or static, where composition is undertaken once, and the composition graph execution is based on data flow or control flow dependencies. Composition may be achieved through textual editing (such as the Directed Acyclic Graph from Pegasus [Deelman, E. et al. 2005]), using scripting tools (such as SCUFL from Taverna [Oinn, T. et al. 2004]), using a functional language (such as Ptolemy II in Kepler [Altintas, I. et al. 2004]), using a graphical editor (such as in Triana [Taylor, I. et al. 2005] and VisTrails [Callahan, S. et al. 2006]), or via the use of software engineering notations, such as UML (in systems such as ASKALON [Fahringer, T. et al. 2005]). Table VI presents a characterization of these techniques and how they relate to each other.

4.1.2 *Component Models.* The design of component models such as CCA [Bernholdt, D. E. et al. 2006] and the CoreGrid Component Model [Aldinucci et al. 2008], may be divided into three levels: (i) the components themselves – identifying how components are implemented and internally managed; (ii) component interfaces – indicating the types of data that can be exchanged with a given component; and (iii) component properties – identifying specific capabilities made available for interaction between the component and the associated container services being made available by a hosting environment.

Abstraction	Example	Description	Dependencies (I or E)	Ordering (I or E)	Resource Decisions (I or E)	Component Existence
Scripting	SWIFT [Zhao, Y. et al. 2007]	Textual aggregation of atomic objects	E	E	E	As needed
Workflow (graph-based)	DAGMan [Frey et al. 2001]	Graph-based description & execution (with data flow)	E	E or I	I	Always
Functional Languages	Distributed parallel Haskell [Trinder, P. et al. 2002]	Declarative program, dependencies via function calls	I	I	I	As needed
Skeletons	Calcium [Caromel and Leyton 2007]	Abstraction of computation & communication patterns	I	I	I	As needed
Compiler-based approaches	Grid Superscalar [Badia et al. 2003]	Sequential semantics, automatic dependency evaluation & parallelization	I	I	I	As needed

Table VI. Techniques for composition (I = Implicit, E = Explicit)

Research in distributed objects has had a significant impact on the development of component models. Component models are related to the previous group (Composition); but here, the developer of the component model has defined a formal framework and a formal set of rules governing the instantiation and interaction of components that are integral to the programming model, whereas in the Composition group, there is no framework or set of rules other than those created implicitly by the application developer. Hence, a component must operate alongside a *component framework* which provides capabilities that a component needs to function and interact with other components, unlike in the Composition group, where the emphasis is on data exchange between components and there is no requirement for components to be developed with reference to a particular framework.

Services: A service may be viewed as more coarse-grained than a component or object; services constitute relatively large, intrinsically unassociated, persistent units of functionality. The service approach shares some similarities with components, in that metadata describing properties of a service is exposed, and the capability of the service is exposed as an interface. Interaction with and between services is supported through the use of messaging, enabling multiple types of messaging infrastructure (of varying degrees of reliability) to be used to carry these messages.

A Service Oriented Architecture (SOA) provides the framework through which such services may be combined together to form ad-hoc applications. The SOA approach necessitates the use of metadata associated with a service to gauge suitability for such a composition. Mechanisms to encode such metadata have ranged in complexity from the use of simple keywords to more complex approaches that model semantic relationships between keywords (such as RDF and OWL). The intention of augmenting metadata with such semantic relationships is to identify candidate services that could be used in the composition process. Hence, services may be "grouped" based on some measure of similarity between their metadata descriptions. In the same way, an "agent" may refer to the provider or consumer of services that are annotated with such semantic information.

A common theme of the "services" category is the encapsulation of loosely-coupled, generally coarse-grained, functionality with associated metadata annotations. These annotations enable the use of these services in a variety of contexts.

It is useful to tie the approaches used to support coordination as discussed in this Section, to those in Sections 2 and 3. As we saw, there exist a number of workflow systems currently exist to support the construction of scientific applications by combining components and services. Although they share a number of similarities, they also have differences in the way the workflow is described, or in the subsequent enactment process that is supported. Understanding patterns of usage within such composition environments is therefore useful to better understand how workflow description or enactment (for instance) could be adapted if the underlying infrastructure changes.

4.1.3 *Communication.* The third category, **Communication**, includes a number of communication models, such as: one/two-sided messaging, Remote Procedure Calls (RPC), event-driven systems, transactional systems, streaming, shared data, shared files, and shared address space. These approaches can be carried out in a sequential (single process/thread) or concurrent (multiple threads) mode of operation. The main idea that ties these together is how execution units (objects, components, services) interact and share data with each other. RPC involves one unit invoking a procedure interface at another unit, eventually leading to a return value from the procedure. Two-sided messaging means that the two communicating units have agreed to exchange data, and both participate in the exchange. One-sided messaging means that one unit can manipulate (get, put, or change) memory that belongs to the other unit, without the other unit being involved or perhaps even aware.

Events are timed messages, which can be used to drive the activities of coordinated units of execution. In transactional messaging systems, sending and receiving applications can indicate that they want to send or retrieve messages within the context of a transaction, where a sequence of messages will all succeed or all fail. When transactional messaging is used, the sending or receiving unit has the opportunity to commit to the transaction (all the messages have succeeded), or to abort the transaction (one or more of the messages failed). When a transaction is aborted all units are rolled back to the state when the transaction was invoked. Asynchronous operations mean that each piece can do other work while communication is occurring, rather than having to wait for the communication to complete. Streaming implies that there is a continuous flow of data from one unit to another, and that the second unit does some data processing and possibly sends one or more streams to other units. The shared data communication model involves sharing of physically distributed memory to support communication between processes. For example, shared files is a model that assumes a shared file system, where one

Communication Model	Example	Support Async?	Support Sequential, Concurrent?
RPC	GridRPC, RMI	Y	Seq, Con
Two-sided messaging	MPICH-G2 [Karonis et al. 2003]	Y	Seq, Con
One-sided messaging	MPICH-G2 [Karonis et al. 2003]	Y	Con
Event-driven	Narada Brokers [Pallickara and Fox 2003]	Y	Seq, Con
Transactional	SIENA [SERL]	Y	Seq, Con
Streaming	Data Turbine [T. Fountain]	N	Con
Shared Data	InterWeave [Scott, M. L. et al. 2000]	Y	Con
Shared Files	MatlabMPI [Kepner 2004]	Y	Seq, Con
Shared Addressing	Titanium [Yelick, K. et al. 1998]	Y	Seq, Con

Table VII. Communication models and examples

unit writes data to a file to be read by a second unit. Similarly, a shared address space is a model where multiple units can read and write data using shared references without knowledge of where the data is physically located. An example of this is the tuple space model, which uses an associative address space. Table VII presents these models and examples of systems that utilize a particular communication model. Most of the approaches can operate in both sequential or concurrent mode.

4.2 Programming Systems

According to [Parashar and Browne 2005], a "Programming System" may be defined as a tuple **(pm,l,am)**, where, **pm**: is a programming model, **l**: is a language syntax for instantiating the programming mode; and **am**: is an abstract machine on which the **pm** has been defined. In the context of the existing discussion, and generalizing the above definition, we define a programming system as one able to provide the composition, component and communication capabilities discussed in Section 4.1.

Table VIII shows the composition approach, components and communication techniques utilized by applications discussed earlier in this paper. There are a number of programming systems already available that support the component models, communication or composition techniques, for instance, and which, if utilized would enable the developer to focus on specifics of the application, rather than the implementation of such core infrastructure. However, very few of these are utilized extensively within the computational science community as developers prefer to "roll their own" capabilities, even though many of these exist in available programming systems. This may be due to a number of factors:

(1) Many programming systems lack robustness when used by developers in scientific applications – for instance, they may have limits on scalability.

(2) The level of maturity, and development & update schedule of programming system may not match the requirements of the application developers.

(3) It may be difficult to isolate a specific capability from a programming system, thereby limiting re-use of a part of such a programming system by an application developer requiring that capability.

(4) Utilizing a particular capability from an existing programming system may

Application Example	Components	Composition	Communication
Montage	-	Scripting	Shared file system
NEKTAR	-	-	2-sided messaging
Coupled Fusion Simulation	Accord	Workflow	Streams, shared address, eventing
Asynchronous RE		Scripting	Streams, shared address, eventing
Climate-Prediction (Generation)	-	-	2-sided messaging
Climate-Prediction (Analysis)	-	-	2-sided messaging, files
SCOOP	-	Scripting	2-sided messaging

Table VIII. Application usage of compositions, components and communication techniques

require installation of a number of other software libraries. Ensuring that all of these additional libraries are accurately versioned and updated pose significant overheads, which restrict their use by application developers.

Constraints remain within existing programming systems, and therefore act as barriers against the better use of existing programming systems within scientific applications. In Section 5 we have attempted to better understand and analyze the reasons for such barriers.

5. A CRITICAL ASSESSMENT OF DISTRIBUTED APPLICATIONS AND SYSTEMS

The discussion in the previous sections clearly indicates that computational infrastructures – both existing (grids) and emerging (clouds) – provide tremendous capabilities and opportunities to support scientific and engineering investigation at unprecedented scales. However, despite an overwhelming need and great potential, the number of production-level, large-scale high performance distributed applications has been relatively small although the effort to bring these applications to fruition has been high. We believe that the reason for this is twofold: (1) emerging distributed computational infrastructures present significant challenges, not faced by traditional HPC applications on parallel systems, such as reliability, heterogeneity, dynamism and security, and (2) appropriate abstractions that support application patterns and that can be used to address these challenges are lacking. Furthermore, a consequence of the small number of distributed applications is that infrastructures do not support these application, which in turn, further increases the efforts required to develop new applications. For example, there are a relatively small number of distributed applications on the TeraGrid, which led to a focus on supporting single site applications.

The goal of this section is to explore the existing *gap* between application requirements and the patterns and capabilities provided by the current generation of systems and tools. In what follows, we first revisit the applications presented in Section 2 with the aim of identifying recurring requirements and associated patterns throughout their lifecycle. We then correlate these with the tools and systems used to support their implementations, and identify gaps where new or different abstractions may be meaningful. In our discussion below we attempt to answer the following questions:

—Are the capabilities and abstractions provided by existing tools and systems sufficient for most distributed applications? And if not, for what aspects of and type of applications are they not sufficient and why?

—Which of the identified patterns are effectively supported by current tools and systems, and which are not (and why)?

—Which existing abstractions are used and which are not (and why)?

5.1 Applications and Tools Revisited

The discussion in Section 2 establishes that large-scale high-performance distributed applications face challenging requirements (see Table I). This is due to their inherent complexity coupled with that of the underlying distributed environment. An interesting observation from Table I is that many of these requirements are common across many of the applications, e.g., co-scheduling in the NEKTAR and Coupled Fusion Simulation applications is such a requirement. Other cross-cutting requirements include hiding delay and providing fault-tolerance, the need to address data/system uncertainty, and the need for dynamic discovery and adaptation, among others. Table IV captures these cross-cutting requirements in the form of recurring patterns. For example, across the six applications discussed, there is a recurring need for Data Processing Pipeline, Master-Worker coordination support, and support for co-allocation.

The applications in Table I (many of which were originally traditional HPC applications) have effectively used distributed infrastructure to advance understanding in their disciplines. However, to achieve this, they have had to implement the above mentioned capabilities at one or more levels, i.e., on the application, programming system, middleware, and/or infrastructure level. For example, the RE application addressed the requirements of latency hiding and fault-tolerance at the application level by using an asynchronous numerical formulation to deal with non-uniform communication latencies and message loss. Other examples include middleware systems that have been extended to address requirements of wide-area scalability (MPICH-G2) and component/service-based programming systems extended to support dynamic resource-sensitive bindings and policy-driven runtime adaptations (Accord).

It can be further observed that in some cases, appropriate abstractions and supporting tools have emerged that meet some of these requirements and that have been effectively reused across applications. For example, multiple applications have used DAGMan (e.g, Montage), abstractions for master-worker coordination provided by Condor and at-home deployment provided by BOINC. However, in many other cases, abstraction, underlying tools and programming systems have been developed to support just one, or at most a few applications, as can be seen from Table II. For example, the needs for pub/sub and/or notification, messaging, latency/fault tolerance, or asynchronous coordination in the RE application have been met by creating a customized set of abstractions and tools (e.g., Comet, Meteor) on top of the JXTA peer-to-peer substrate. Interestingly, as seen in Table V, most of the recurring patterns observed in the applications studied in this paper are supported by a multitude of tools. However such tool reuse does not appear to be pervasive.

5.2 Mind The Gaps!

The above observations, indicate the existence of a gap: a lack of readily available, reusable and extensible abstractions that support cross-cutting application requirements. The lack of appropriate abstractions has in fact resulted in "one-off" solutions that address challenges in a highly customized and non-reusable manner, and are, in general, expensive to develop and manage in terms of the time and effort involved. Based on discussions above, the reasons for this gap are:

(1) *Complexity:* Distributed applications present unique challenges resulting in increased complexity, and often have complex structures that compose multiple patterns at different levels as well as at different phases of application execution.

(2) *Incompleteness:* Existing tools and patterns are often incomplete or inflexible with respect to application needs. For example, tools that support the MW pattern often only address failures of workers and not failures of the master.

(3) *Customization:* Tools tend to be customized to specific applications and their requirements, with limited concern for reuse or extensibility. For example despite community efforts such as CCA, few applications, if any, mix-and-match components developed by different projects. Performance, semantic and interface compatibility are certainly barriers, but so are the design of components and the tools to compose different components. Finally, both applications and tools are often highly dependent on and tuned to a specific execution environment, further impacting portability, reusability and extensibility.

We now discuss specific instances where these factors have come into play at different stages of the application life-cycle, i.e., development (programming), deployment (mapping and instantiation), and execution (runtime management). Our focus is on the functional aspects of applications, tools, and systems, and the gaps among them. Clearly, non-functional aspects play an important role in the lack of production distributed applications; such an analysis, however, is outside the scope of this paper and is left for future work.

Gaps in application development:. In several instances, the surveyed distributed applications faced gaps in which existing systems and tools did not meet their needs. For example, ClimatePrediction needed a specific data processing workflow and the Martlet tool was created. Similarly, the RE application led to the creation of new tools to support dynamic adaptation, asynchronous coordination and interactions. Specifically, the Accord programming system was developed to support dynamic policy driven adaptations, the Meteor messaging substrate was developed to support pub/sub messaging, and the Comet infrastructure was developed to support the coordination requirements. This example also serves to illustrate an opportunity, as these tools, and the abstractions they provided were re-used across these applications. Finally, the Coupled Fusion application had unique requirements for asynchronous coordination and data communication within traditional HPC environments that led to the development of the Seine and DART frameworks [Zhang and Parashar 2006; Docan, C. et al. 2010].

In addition to addressing gaps at the programming system and tools level, gaps have also been addressed at the application level, i.e., application formulations have been adapted to address requirements due to their distributed implementations.

For example, the REu algorithm and implementation [Gallichio et al. 2007; Li and Parashar 2007; Zhang, L. et al. 2006] was adapted to be robust and handle system heterogeneity, failures, and latency tolerance.

Gaps in application deployment:. The applications studied also demonstrated gaps in application deployment. Ref. [Boghosian, B. et al. 2006] reported problems with deployment and execution of NEKTAR over federated grids. Often production systems still have single points of failure or reliance on a single middleware component; therefore as the number of resources and degrees-of-freedom increases, the likelihood of failure increases accordingly. In the case of the Montage application, the choice of where to deploy Montage components should be undertaken at runtime, and should depend on both system and network characteristics and usage; however, there are no simple tools that can do this. This is a fairly common problem for workflow-based applications. Similar issues also exist in case of the Coupled Fusion Simulation application, which needs support for co-allocation and scheduling. Finally, managing application dependencies during deployment remains a key gap, i.e., there is no tool that can help resolve the hierarchical dependencies that may exist between the software system and libraries that an application uses.

Gaps in application execution:. Many distributed applications require reliable execution. Montage relies upon Condor DAGMan to ensure individual tasks are reliable, but it does not guarantee that the enactor task itself will be reliable. This impacts Montage runs that have to be restarted upon failure. In the case of ClimatePrediction.net's data generation, BOINC had to be altered to support longer running workers, and to support phased worker completion. Similarly, reliability and robustness requirements for the RE application were addressed at the infrastructure level (handling peer failures in peer-to-peer overlay on which the infrastructure is built) as well as middleware level (Meteor can handle lost messages and Comet can handle lost *tuples*). Additionally, the runtime system was further adapted to deal with application task failures. Finally, in case of the Coupled Fusion application, which needs support for data steaming, an autonomic data streaming support using RDMA technologies and model-based control strategies was developed [Bhat, V. et al. 2007; Docan et al. 2010].

5.3 Looking Ahead

Distributed application designers have "voted with their feet" with respect to pattern and tool uptake. And in some cases, likely "voted" not to even embark on the effort. We have identified some of the reasons, but clearly there has been a perception and perhaps a reality that existing tools were inadequate. Some reasons could be: lack of compatibility with respect to other elements of the application (e.g., other tools already used by the application), a mismatch in requirements (e.g., the tool did not support some critical need), a mismatch in applicability (e.g., the tool has too much custom or domain-specific behavior or is tied to a specific infrastructure), perceived lack of robustness, or are simply lacking ease-of-use. It is important to understand why some tools have been more widely used and others not. For example, MPICH-G extended another widely used tool, MPI, and therefore was already a match with many existing MPI applications. Extending widely used tools with those that users are already comfortable with, to distributed versions seems to be a promising pathway. A similar story holds for Condor – users can program their applications in familiar languages and the environment takes care of distribution. MapReduce/Hadoop is another such example that provides abstractions and infrastructure to make it easier to use distributed infrastructure for a specific class of applications.

The success of BOINC may be based upon its core and singular capability: to provide applications with huge resource pools. Donated nodes do not interact and the model is fairly simple. Programmers may be willing to learn new paradigms and re-structure programs if the new model is simple. In addition, as an added feature, BOINC is flexible. For example, BOINC has a consensus mechanism based on replication, but allows the programmer to specify the policy for replication and consensus, thus widening its applicability in our view. Furthermore, the programmer may opt out of replication and consensus altogether. This suggests that another best-practice may be that tools should provide a “mechanism” (e.g., replication/consensus) that implements a feature, but allow the “policy” (e.g., majority consensus) to be easily programmed.

Another observation is that there is no common architecture upon which to develop tools. However, as JXTA has been used as a common platform to enable several tools, such a common architecture is a possibility. On the other hand, some tools and systems built explicitly for wide reuse, such as component models and frameworks, have seen slow adoption. We speculate that application designers may have concluded that the reward for this paradigm shift was not worth the risk or effort. A critical mass of adopters is needed to mitigate the risk of a paradigm shift and this has not happened for these systems. While many specifications and frameworks were proposed, the fact that none emerged as a leader may be another contributing factor.

Analyzing a set of applications has shown that recurring and cross-cutting patterns do exist. We believe these recurring and cross-cutting patterns represent opportunities where appropriate supporting tools and high-level abstractions can have a significant impact: by reducing time and effort required to build, deploy and manage large-scale high performance distributed applications, their impact on all computational disciplines increases. We recognize that it is not possible to anticipate all of the patterns, tools and programming systems that distributed applications will need down the road. There are just too many moving parts! Changes in infrastructure and application requirements cannot be predicted. For this reason, we advocate that tools provide open interfaces to enable more easy integration and interoperation with other tools. Whether this can be best achieved by new extensible tools or “tool orchestration” or a tool “framework” is an open question. This, of course, does not preclude the internal evolution of existing tools to implement new abstractions, but just this is not enough in our view.

6. CONCLUSIONS

Significant strategic investments coupled with tremendous technical advances have resulted in distributed national and global computational infrastructures that integrate computers, networks, data archives, instruments, observatories and embedded sensors/actuators, with the overarching vision of enabling of new paradigms and practices in computational science and engineering. Several large-scale distributed applications have also demonstrated the potential benefits of such infrastructures and have provided new scientific insights in their disciplines. However, such applications have been relatively few, and have also demonstrated the costs, in terms of time and resources, associated with developing, deploying and managing the applications. These costs constitute a barrier that is inhibiting large numbers of potential applications from benefiting from the infrastructure.

The application-driven critical investigation presented in this paper has been aimed at understanding the reasons behind this state-of-affairs. We observe that there does not exist a single unifying paradigm (such as message-passing for parallel programming) that can be used to enable distributed applications; this is a reflection of the fact that the concerns and objectives for distributed applications are both large in number and broad in scope. We find that at one level, the range of tools, programming systems and environments is bewildering large, making extensibility and interoperability difficult, whilst at another level, there exists insufficient support for simple *development integrated with deployment and execution* of distributed applications. In fact, it may be argued that the design and implementation of the most ambitious distributed infrastructures are often not consistent with the requirements of distributed applications they intend to support. Conversely, it can be argued that, had distributed infrastructures been *architected* with a greater focus on application characteristics and execution modes (such as the cross-cutting concerns mentioned in Section 5.3), the state of the theory and practice of distributed systems for large scale scientific applications would have been different.

An analysis of the requirements and implementations of a representative set of "successful" large-scale high-performance distributed applications clearly demonstrated cross-cutting requirements and recurring patterns; yet we find that commonly occurring execution patterns for distributed applications are typically not supported in any explicit fashion. For example, there exist commonly occurring execution modes for which there is no common infrastructural support, such as, hierarchical job submission. Once these common modes are identified (as patterns), then mechanisms to support them (as abstractions) need to be formulated.

The importance of this approach and that of supporting meaningful and powerful patterns is probably best demonstrated by MapReduce and the impact that explicit support (by Google and others) for this data-parallel programming pattern has had on a range of applications. There has been a rush of activity to reformulate applications using MapReduce, even where other patterns would probably have been more appropriate and effective. But this is just indicative of the chicken-and-egg situation referred to earlier; once patterns are well supported, their use (and even misuse) and uptake by application developers increases dramatically. There remains, however, scope for improvement in both the range and number of supported abstractions and the type of infrastructure over which such abstractions are made available. In fact, not only do distributed applications need abstractions to support the requirements through the individual stages of development, deployment and execution, support for applications through these stages needs to be a coherent integrated activity, as opposed to a disjointed effort [Jha, S. et al. 2009]. This will then enable methods for developing distributed applications that, for example, take into account the status of the infrastructure (deployment) as well as the execution modes of the application.

To summarize, our analysis has demonstrated significant *gaps* between patterns and the abstractions provided by the current generation of programming systems and tools. We believe that addressing these *gaps* is critical, and is absolutely necessary before distributed computational infrastructures can realize their vision of revolutionizing scientific investigation and leading to dramatically new insights and understanding. In fact, the definition of appropriate abstractions and the development of supporting tools and systems that can fill identified gaps remains the key research focus of the authors.

Acknowledgements

This paper is the outcome of the e-Science Institute sponsored Research Theme on Distributed Programming Abstractions. We would like to thank many people who participated in the workshops and meetings associated with the theme and help frame the problem landscape upon which this paper is built. In particular, we would like to thank the following people for their contributions to the paper and theme – Gabrielle Allen, Geoffrey Fox, Gerry Creager, Daniel Goodman, Craig Lee, Andre Merzky, and Phil Trinder. SJ acknowledges Malcolm Atkinson for many significant discussions, insight and advice on theme related issues.

REFERENCES

"Local Data Manager (LDM) `http://www.unidata.ucar.edu/software/ldm/`".

Aldinucci, M., Danelutto, M., Bouziane, H. L., and Perez, C. 2008. Towards software component assembly language enhanced withworkflows and skeletons. *CoreGrid Technical Report TR-0153. Available at http://www.coregrid.net/mambo/content/view/566/377/.*

Altintas, I. et al. 2004. Kepler: An extensible system for design and execution of scientific workflows. *16th Int. Conf. on Sci. and Stat. Database Management (SSDBM), IEEE Comp. Soc. Press*, 423–424.

Anderson, D. P. 2004. BOINC: A System for Public-Resource Computing and Storage. In *GRID*. 4–10.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. 2002. SETI@home: an experiment in public-resource computing. *Commun. ACM 45,* 11, 56–61.

Andrews, G. 2000. *Foundations of Multithreaded, Parallel and Distributed Programming.* Addison Wesley.

Asanovic, K. et al. 2006. The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Dept., UC Berkeley.

Badia, R., Labarta, J., Sirvent, R., Perez, J. M., Cela, J. M., and Grima, R. 2003. Programming grid applications with grid superscalar. *Journal of Grid Computing 1,* 2, 151–170.

Berman, F. et al. 2003. Adaptive computing on the grid using apples. *IEEE Trans. on Par. and Dist. Sys. 14*, 369–382.

Bernholdt, D. E. et al. 2006. A component architecture for high-performance scientific computing. *Intnl. J. of High Perf. Comp. Applications 20,* 2 (Summer), 163–202.

Bhat, V. et al. 2007. An self-managing wide-area data streaming service. *Cluster Comp.: The J. of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing 10,* 7 (December), 365–383.

Bogden, P. S. et al. 2007. Architecture of a community infrastructure for predicting and analyzing coastal inundation. *Marine Tech. Soc. J. 41,* 1, 53–71.

Boghosian, B. et al. 2006. Nektar, spice and vortonics: Using federated grids for large scale scientific applications. In *Proc. of Challenges of Large Applications in Distributed Environments (CLADE)*. IEEE Comp. Soc., 32–42.

Callahan, S. et al. 2006. Managing the evolution of dataflows with vistrails. *SciFlow: IEEE Workshop on Workflow and Data Flow for Scientific Applications*.

Carilli, C. and Rawlings, S. 2004. *Science with the Square Kilometre Array.* New Astronomy Reviews, vol. 48. Elsevier.

Caromel, D. and Leyton, M. 2007. Fine-tuning algorithmic skeletons. In *Proceedings of Euro-Par 07, LNCS*. Vol. 4641. Springer, 72–81.

Chakravarty, M. T. T. et al. 2006. Data Parallel Haskell: a status report. Tech. rep., Univ. of New South Wales and Microsoft Res. Ltd, Cambridge. November.

Chang, C. S., Ku, S., and Weitzner, H. 2004. Numerical study of neoclassical plasma pedestal in a tokamak geometry. *Physics Plasmas 11*, 2649.

Christensen, C. et al. 2005. The challenge of volunteer computing with lengthy climate modelling simulations. In *Proc. of 1st IEEE Conf. on e-Science and Grid Comp.*

Coddington, P. 1993. An analysis of distributed computing software and hardware for applications in computational physics. In *Proc. of 2nd Intnl. Symp. on High-Perf. Dist. Comp. (HPDC-2)*. IEEE Comp. Soc. Press, 179–186.

Cole, M. 2004. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing 30*, 389–406.

Concoise. Constraint Oriented Negotiation in Open Information Seeking Environments for the Grid, http://www.conoise.org/.

Condor. Condor home page – `http://www.cs.wisc.edu/condor/`.

Cummings, J. 2008. Plasma Edge Kinetic-MHD Modeling in Tokamaks Using Kepler Workflow for Code Coupling, Data Management and Visualization. *Comm. in Comp. Phy. 4*, 675–702.

Dean, J. and Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM 51,* 1, 107–113.

Deelman, E. et al. 2005. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Prog. J. 13,* 3, 219–237.

Docan, C., Parashar, M., and Klasky, S. 2010. Enabling high speed asynchronous data extraction and transfer using dart. *Conc. and Comp.: Practice and Experience*.

Docan, C. et al. 2010. Experiments with memory-to-memory coupling for end-to-end fusion simulations workflows. In *Proc. of 10th IEEE/ACM Intnl. Symp. on Cluster, Cloud and Grid Comp. (CCGrid)*. IEEE Comp. Soc. Press.

Ellis, J. 2007. Beyond the standard model with the LHC. *Nature*, 297–301.

European Grid Initiative. `http://www.egi.eu/`.

Fahringer, T. et al. 2005. Askalon: A grid application development and computing environment. *6th Int. Workshop on Grid Comp.*, 122–131.

Frey, J., Tannenbaum, T., Livny, M., and Tuecke, S. 2001. Condor-G: a computation management agent for multi-institutional grids. In *Proceedings of the 10th IEEE Symposium on High-Performance Distributed Computing*.

Gallichio, E., Levy, R., and Parashar, M. 2007. Asynchronous replica exchange for molecular simulations. *Journal of Computational Chemistry 29,* 5, 788–794.

Geyer, C. and Thompson, E. 1995. Annealing markov chain monte carlo with applications to ancestral inference. *J. Am. Stat. Assoc. 90,* 431, 909–920.

Goodman, D. 2007. Introduction and Evaluation of Martlet, a Scientific Workflow Language for Abstracted Parallelisation. In *Proc. of 16th Intnl. World Wide Web Conf.* ACM.

Grimshaw, A. S. et al. 1992. No pain and gain! experiences with Mentat on a biological application. In *Proc. of First Intnl. Symp. on High-Perf. Dist. Comp. (HPDC-1)*. IEEE Comp. Soc. Press, 57–66.

ITER. International thermonuclear experimental reactor (ITER) `http://www.iter.org`.

Jacob, J. C. et al. 2007. Montage: A grid portal and software toolkit for science-grade astronomical image mosaicking. *Intnl. J. of Comp. Sci. and Eng. 3*.

Jha, S. et al. 2009. Critical perspectives on large-scale distributed applications and production grids. In *10th IEEE/ACM Intnl. Conf. on Grid Comp.* Vol. 13-15. 1–8.

Karonis, N., Toonen, B., and Foster, I. 2003. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing 63,* 5 (May), 551–563.

Kepner, J. 2004. MatlabMPI. *Journal of Parallel and Distributed Computing 64*, 997–1005.

Klasky, S. et al. 2005. Data management on the fusion computational pipeline. *J. of Phys.: Conf. Series 16*, 510–520.

Lee, C. and Talia, D. 2003. Grid programming models: Current tools, issues and directions. In *Wiley Series in Communications Networking & Distributed Systems*. Wiley InterScience, 555–578.

Li, Z. and Parashar, M. 2007. Grid-based asynchronous replica exchange. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (Grid 2007)*. 193–200.

Luckow, A., Jha, S., Kim, J., Merzky, A., and Schnor, B. 2009. Adaptive Distributed Replica Exchange Simulations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 367,* 1897, 2595–2606.

MacLaren, J. 2007. HARC: The Highly-Available Resource Co-allocator. In *Proceedings of GADA'07 (OTM Conferences 2007, Part II), LNCS*. Vol. 4804. Springer-Verlag, 1385–1402.

Manke, J., Neves, K., and Wicks, T. 1993. Parallel computing for helicopter rotor design. In *Proc. of 2nd Intnl. Symp. on High-Perf. Dist. Comp. (HPDC-2)*. IEEE Comp. Soc. Press, 21–28.

Manos, S. et al. 2008. Distributed MPI Cross-site Run Performance using MPIg. In *Proc. of 17th Intnl. Symp. on High Perf. Dist. Comp. (HPDC)*. Boston, MA, USA, 229–230.

Mattson, T., Sanders, B., and Massingill, B. 2004. *A Pattern Language for Parallel Programming*. Addison Wesley.

Mechoso, C. R. et al. 1992. Distributing a climate model across gigabit networks, proc. of first intnl symp. on high-perf. distr. comp. (hpdc-1). IEEE Comp. Soc. Press, 16–25.

Mesonet, T. `http://mesonet.tamu.edu/`.

Moretti, C., Bulosan, J., Thain, D., and Flynn, P. 2008. All-Pairs: An Abstraction for Data-Intensive Cloud Computing. In *Proc. IEEE Intl. Parallel and Distributed Processing Symposium*. to appear.

Oinn, T. et al. 2004. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics 20*.

OMG. 1998. The complete formal specification/98-07-01: The CORBA/IIOP 2.2.

OMG. 2002. Corba component model, `http://www.omg.org/technology/documents/formal/components.htm`.

Pallickara, S. and Fox, G. 2003. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003, LNCS*. Vol. 2672. Springer-Verlag, 41–61.

Parashar, M. and Browne, J. 2005. Conceptual and Implementation Models for the Grid. *IEEE, Special Issue on Grid Computing 93,* 2005, 653–668.

Park, W., Belova, E. V., Fu, G. Y., Tang, X. Z., Strauss, H. R., and Sugiyama, L. E. 1999. Plasma simulation studies using multilevel physics models. *Phys. Plasmas 6*, 1796.

Rabhi, F. A. and Gorlatch, S. 2003. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, London.

RGMA. R-GMA: Relational Grid Monitoring Architecture, http://www.r-gma.org/.

Sarmenta, L. F. G. and Hirano, S. 1999. Bayanihan: Building and studying web-based volunteer computing systems using Java. *Future Generation Computer Systems 15*, 675–686.

Scott, M. L. et al. 2000. Interweave: object caching meets software distributed shared memory. *SIGOPS Oper. Syst. Rev. 34,* 2, 32.

SERL. Scalable internet event notification architectures, `http://serl.cs.colorado.edu/~serl/dot/siena.html`.

Soh, H., Haque, S., Liao, W., and Buyya, R. 2007. *Advanced Parallel and Distributed Computing: Evaluation, Improvement and Practice.* Nova Science Publishers, Chapter Grid Programming Models and Environments.

Swendsen, R. H. and Wang, J.-S. 1986. Replica monte carlo simulation of spin-glasses. *Phys. Rev. Lett. 57,* 21 (Nov), 2607–2609.

T. Fountain. NSF open source data turbine initiative, `http://www.dataturbine.org/`.

Taylor, I. et al. 2005. Visual grid workflow in Triana. *J. of Grid Comp. 3,* 4, 153–169.

TeraGrid. `http://www.teragrid.org/`.

Trinder, P. et al. 2002. Parallel and distributed Haskells. *J. of Functional Prog. 12*, 469–510.

van der Aalst, W. M. P. et al. 2004. Design and Implementation of the YAWL system. In *Proceedings of The* 16th *Intnl. Conf. on Adv. Info. Sys. Eng. (CAiSE 04).*

YAWL Patterns. `http://www.yawl-system.com/resources/patterns.html`. last accessed: January 2009.

Yelick, K. et al. 1998. Titanium: A high-performance Java dialect. In *Proc. of Workshop on Java for High-Performance Network Computing.*

Zaroubi, S. and Silk, J. 2005. LOFAR as a probe of the sources of cosmological reionization. *MNRAS L64*, 360.

Zhang, L. and Parashar, M. 2006. Seine: A dynamic geometry-based shared space interaction framework for parallel scientific applications. *Conc. and Comp.: Prac. and Exp. 18,* 15 (April), 1951–1973.

Zhang, L. et al. 2006. Salsa: Scalable asynchronous replica exchange for parallel molecular dynamics applications. In *Proc. of 2006 Intnl. Conf. on Par. Proc.* 127–134.

Zhao, Y. et al. 2007. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. *IEEE Workshop on Scientific Workflows*.