Programming Abstractions for Large-scale Distributed Applications

Names against a section is only to indicate overall responsibility - not sole responsibility

November 24, 2008

Contents

Abstract

The number of coupled distributed computational infrastructures has recently increased. Some prominent examples are the US TeraGrid and OSG, the UK National Grid Service, and the European DEISA and EGEE. However, the number of applications that can utilize such a distributed infrastructure in novel and effective ways is limited. Most applications on such infrastructures just use a single parallel computer. [?, ?]

Even though individual computers are becoming more powerful, there remains and will remain a need for aggregating distributed resources for scientific computing. In the simplest case, the required amount of computing power at an instant of time may exceed the capacity of individual supercomputers, thus requiring the coupling of physically distributed resources. Additional reasons are the need for greater “peak” capacity arising from time varying resource requirements, the need for higher throughput, the requirement to use specialized hardware in conjunction with general purpose computing units, the need to do computing near data that is too large to be efficiently transferred, the desire to use distinct computer architectures on different parts of a single application, and collaborations between physically separated groups who each have to their own computing.

It is generally accepted that the ability to write distributed applications has lagged seriously behind the ability to develop and deploy infrastructure. Writing applications for distributed infrastructure is notoriously hard; acceptable models and abstractions are not yet available. There are many ad-hoc solutions employed to write distributed applications, which results in lack of generality and the inability of programs to be extensible and independent of infrastructure details. In most cases, though, the difficulty simply leads to potential distributed applications not being written.

The complexity of writing large-scale, distributed applications stems from combining the fields of high-performance computing and large-scale, distributed systems. No single solution from either field appears to be suitable for large-scale distributed applications. In high-performance computing today, performance is paramount, and systems are typically homogeneous, failure-free over the lifetime of the computation, and operated within a single administrative domain.

In large-scale, distributed systems, scalability, heterogeneity, fault-tolerance and security issues prevail; performance as measured by peak utilisation is often considered a secondary concern; and arguably, the most complicating factor is the requirement to compute across virtual organizations and administrative domains. In general, other than performance, the constraints in distributed programming are in many ways much harder to satisfy than in parallel programming, although as parallel machines grow in size and complexity, particularly at the highest end, fault tolerance is becoming increasing important there as well.

Thus, 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 such large-scale, distributed applications.

The scope of this survey paper is set by the application classes defined in Section ??; specific applications
considered in these classes are mostly scientific in nature. Interestingly, although confined to less than a
dozen types, scientific applications in these categories comprise a significant fraction of all applications when
viewed from the perspective of computing resource consumption. In this manuscript, we survey existing
approaches, models, and abstractions for large-scale distributed applications. We identify commonalities and
recurring structures, patterns, and environments. Gaps and omissions in the state of the art are identified,
outlining directions for future research.

Our work is unique in the sense that it is centered around real-world scientific applications. We provide a
novel and, arguably, the most comprehensive analysis of distributed applications. This is the first time that
programming patterns have been derived and analyzed from a wide range of real distributed applications. We
also identify commonly found distributed programming models and abstractions. A further unique value of
this manuscript lies in the integrated approach towards applications and programming models and patterns,
resulting in the ability to provide a critical assessment of programming distributed applications.

1 Introduction and Outline

Shantanu Jon Manish

Many problems that require intensive computations on large amounts of data would benefit from large-scale,
distributed application execution environments. Such data is found in problem domains within computational
science (e.g., climate modeling, genome analysis, and particle physics) and within business processes (e.g.,
stock market analysis and supply chain optimization). Writing applications for such large-scale problems is
notoriously hard; commonly acceptable models and abstractions are not yet available. Instead, many ad-hoc
solutions are currently being deployed by programmers.

The modifier “large-scale” can be applied to several concepts. Frequently, the amount of data to be analyzed
can be large (e.g., in data mining). Or, the data can be generated in real time in quantities that mandate
online processing as intermediate storage is not feasible (e.g., by the Large Hadron Collider (LHC) [?]).
Alternatively, data may be stored or generated in a large-scale physically distributed fashion (e.g., by the
Low Frequency Array (LOFAR) [?] or the envisioned SKA [?] distributed radio telescopes), imposing high-
volume, wide-area data transfers. Additionally, the required amount of computing power may exceed the
capacity of individual supercomputers, again requiring the coupling of physically distributed resources, a
scenario that fostered the area of grid computing.

The complexity of writing large-scale, distributed applications stems from the combined complexity of, and
overlapping but non-identical issues of importance in, the fields of high-performance computing and large-
scale, distributed systems. In high-performance computing, performance is paramount, and systems are
typically homogeneous, almost failure free (at least historically), and operated within a single administrative
domain. In large-scale, distributed systems, scalability, heterogeneity, fault-tolerance and security issues
prevail; performance is generally a secondary concern.

Both fields provide well established approaches, models, and abstractions to their respective application
programmers. In high-performance computing, models such as MPI and OpenMP are widely accepted. In
distributed systems, remote method invocations (such as Java’s RMI) and web services are in widespread use.
No single solution from either field, however, appears to be suitable for the class of applications investigated
here. Additionally, many legacy applications and programmers need to be supported for practical relevance
and protection of investments of the application users.

There is a both a perceived and genuine lack of distributed applications that can seamlessly utilize distributed
infrastructures in an extensible and scalable fashion. The reasons for this exist at several levels. First, there
exists insufficient support for the simple development of distributed applications. For example, there isn’t a
single unifying paradigm such as MPI for parallel applications that enables a large class of applications. At
a second level is the status of the infrastructure and the lack of support for the deployment of distributed
applications – whether they involve single components or multiple distributed components. There are multiple
reasons for this, not least of which is the fact that distributed infrastructure and associated support tools
remain relatively immature (compared with their traditional HPC/parallel counterparts). Finally, but equally
important, is the fact that commonly occurring usage modes associated with distributed applications are
typically not supported in any explicit fashion. In general, distributed applications differ from monolithic,
parallel applications in that their usage modes – which are distinguishable from deployment requirements – are an important aspect of the overall characteristics of the application, something typically not the case for parallel applications (i.e.; the equivalent of mpirun and the orchestration of sub-components, if any, may be done internal to the distributed application).

The ease with which distributed applications can be developed and deployed both need improvement. For example, there exist commonly occurring usage modes for which there is no infrastructural support. On the other hand, there are likely to be potentially useful but yet unidentified usage modes. Once these common modes are identified (say as patterns), then mechanisms to support them (say as abstractions) can be formulated. Supporting tools and approaches are needed at each of the three levels, in such a way that commonly occurring methods for programming distributed applications take into account the status of the infrastructure, programming tools and usage modes. In other words, programming abstractions should be determined with these in mind.

The motivation of this paper is to provide a comprehensive overview of the types of distributed applications and the current approaches to their development (programming), deployment (usage.) This work differs from earlier work in that the focus here is on real-world applications. The aim of this paper is to use the understanding gained primarily from these applications to provide a gap analysis between existing abstractions and what is required by current applications – which could then, in turn, involve new abstractions either at the programmatic, deployment, or usage level.

In this manuscript, we survey existing approaches, models, and abstractions for large-scale distributed applications. We identify commonalities and recurring structures and patterns. Gaps and omissions in the state of the art are identified, outlining directions for future research.

- What the distributed programming abstractions and models?
- How are they “offered to application developers”?, For example there have been libraries of parallel patterns
- How should they be offered? Patterns are heavily biased by software engineering...
- ...

Given the centrality of distributed applications, while focusing on Programming Models and Abstractions, we believe this is will be a unique paper.

The main aims of this paper are to:

- Compile and discuss a representative list of distributed applications
- Define and describe the use of Programming Models and Abstractions
- Map each application to the Programming Models and perform a GAP analysis

1.1 Definitions and Concepts

Jha
This section will define the fundamental concepts used in the paper. These definitions are at least partly based on common usage [6]. Several of the terms used are, however, over-defined, and are used in a variety of contexts and meanings in different communities. In particular, the well known notion of design patterns [7] is decidedly different from our definition.

**Programming Abstraction:** A programming abstraction is the generalization of a frequently occurring theme or concept related to application programming. For example, the term message is an abstraction of the frequently occurring concept of a small, opaque piece of data which is exchanged between different application components. End point is the abstraction of an entity that is able to send or receive such data. In another example, task is an abstraction of the frequently occurring concept of an encapsulated and self-contained piece of work. Abstractions can be hierarchical. For example, a job can be seen as a task or as a collection of tasks.

**Pattern:** Patterns are abstractions of frequently occurring compositions of abstractions. For example, the pattern broadcasting refers to the action of sending a message to all available end points.

**Programming Model:** A programming model is a collection of one or more related programming abstractions and one or more patterns that enable conceptual expression of applications.

For example, the programming model message passing can be seen as a collection of such patterns as unicasting, multicasting and broadcasting of messages between different types of end points and communication groups (and additionally, many more patterns.)

Programming models can contain one or more patterns. That (a) implies a semantic overlap between the notion of a pattern and that of a programming model, and (b) suggests that hierarchies and overlaps of programming models may exist. For example, the pipelining pattern may as well be part of the message passing programming model as of the peer-to-peer programming model.

**Implementation:** An implementation provides an explicit incarnation of a programming model (or parts thereof) to the application programmer or some other end user. For example, the MPICH-G library provides the message passing programming model (in its incarnation as the MPI standard) to distributed application programmers. The programmer can use that implementation to conceptually instantiate the programming abstractions in the application, and then perform actions for the application; he can thus, for example, create end points, and broadcast messages between them.

An implementation can implement a programming abstraction, a pattern, or a programming model. Also, an implementation can exist in a variety of forms. For example, the publish-subscribe programming model can be implemented as a library, as a service, or as a set of components in a virtual programming environment. All of these, however, expose an interface (API, WSDL, ports, . . . ) to the application programmer that allows her to use the implementation and to create and use instances of the implemented patterns and abstractions.

**Usage Mode:** A usage mode is a commonly occurring deployment pattern that influences the resource and infrastructure access for an application or a class of applications. A single application can have multiple usage modes, and multiple applications can have similar usage modes; usage modes are not unique to an application. For example, a Molecular Dynamics (MD) application can be used stand-alone or as a component that is just a part of a larger application. A stand-alone MD application could in turn be used in different ways. For example, identical ensembles of the same application could be used for parameter-sweeps (decoupled) or for Replica-Exchange (loosely-coupled) simulations. Additionally, a single instance of the application could be used in a “long-running” mode, or a sequential (multi-stage) mode. Finally, a single MD application can be either used on one machine or split over multiple machines. These are all different usage modes of a single application, and the infrastructure used and the tools required to support each usage mode are different. For example, some usage modes can be supported by a static configuration of resources (where the resource requirement can be determined in advance), while some may require resource selection and configuration, or co-scheduling might be required to support distributed tightly-coupled applications but not when run on a single machine.
In defining abstractions and patterns, we face a chicken-and-egg problem: abstractions and patterns are derived from observations of re-occurring concepts and themes that are used in programs. Application programmers can, however, only use the patterns and abstractions that are provided by some implementation of a programming model, and that have thus been observed earlier. So, where do new patterns and programming models come from?

One answer is that application programmers tend to encapsulate re-occurring pieces of code for easier re-use. As these pieces are re-used more and more often, they are generalized, with the abstractions becoming more uniform, and they are often collected in libraries of related code snippets. That is, in fact, the seeding point for new programming 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.

One must keep in mind that decoupling of software components, as implied by the usage of programming patterns, is favored for many reasons, e.g. simpler design, code re-use, and flexibility; and that this decoupling almost always implies a change in performance, memory consumption, and interoperability. However, that tradeoff is well understood in the community, and is widely accepted to support the decision in favor of decoupling.

Another source for programming models and patterns are theoretical discussions and musings of computer scientists (and others), which lead to potentially large and complex programming models without the need for very long evolution cycles. Examples of such models are the Globus libraries for Grid Computing, or the AVS realization of a flexible visualization pipeline. More often than not, it turns out however, that such models are too constrained or too complex, or are off-target for some application groups after all, which implies a later evolution of these models as well.

2 Example Applications

Dan Andre

There are a number of distributed applications that currently exist. Here, we have tried to select a representative but non-exhaustive set. The example applications are described in Section ??.

2.1 Descriptions of the Applications

Montage is an 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 uses files for input and output. The dependencies between the executable runs are stored as a DAG, and this DAG is run using Pegasus and DAGMAN. 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. At the Montage home site JPL/Caltech’s IPAC, a local cluster is available for typical small Montage uses, but large jobs are farmed out to distributed resources.

Numerical Relativity Simulation Exact solutions of Einstein’s equations are known only for a very small set of idealized physical problems. Numerical solutions of Einstein’s equations are thus used to confirm the real world predictions of relativity, and to analyze a wide range of astrophysical phenomenons. The numerical relativity simulations usually have small to moderately sized input data (a discrete description of the problem at hand), and run very compute intensive processes with frequent message based data exchange between them. Despite the small input data, the intermediate memory requirements of the simulations can be huge, due to the sheer number of physical variables involved in such settings, and due to the numerical resolution required to (a) being able to compute reasonable accurate predictions, and (b) obtain stable numerical behaviour (in particular, infinities are a prevalent problem in these calculations). Very often, dynamic mesh refinement techniques are exploited by the application codes to react to numerical instabilities on the fly.

Both, the large compute and memory requirements, motivate the utilization of multiple (and thus mostly distributed) resources. As the tight coupling of the individual processes make application scaling a real challenge, distribution at least allows to tackle problems which are otherwise not approachable on single resources at all.

Home-based patient monitoring and data analysis Support for individual-centered healthcare is an important trend in modern medicine, primarily aiming to identify how particular treatments and drugs impact individuals rather than a clinical group. This is also a recognition that a “one size fits all” healthcare solution may not adequately address and differentiate between individual-specific needs and particular priorities identified in a national health plan. In this context, the timely identification of individuals who pose a high risk of developing a particular disease has become an important priority. Diabetes is one such disease, the increase of which in adults worldwide is estimated to rise to 5.4% by 2025, with the number of adults with diabetes in the world rising from 135 million in 1995 to 300 million in 2025. The ability to capture data directly from an individual, and subsequently analyze this for trends that could indicate potential risk to the individual, provides a significant advance over current approaches to diabetes management. The increasing availability of low cost mobile devices, the ability to interconnect these devices to each other and the existing wired infrastructure provides opportunities for seamless patient monitoring. Such monitoring and analysis also needs to be undertaken in a time frame that is meaningful, as the timeliness of prediction is a key challenge for existing systems. In the context of healthcare provision, it is also necessary to account for the reliability of such predictions for computing individualised risk. This application scenario involves capture of data streams from wearable sensors onto a “hub” local to the patient (in the same house, for instance), and then periodic transmission of this data to a central repository for analysis. Three types of analysis may subsequently be undertaken on this data: (i) analysis and correlation of time series data from an individual patient (representing HbA1c concentration, blood pressure and blood lipid concentration for instance); (ii) analysis and correlation across a group of patients, indicating common trends; (iii) detection of templates within data to evaluate the occurrence of risk “signatures” within the recorded data. Data for a patient may be distributed across multiple repositories, or may involve data of different modalities (such as image data from MRI scans or retinopathy cameras, and numeric data from sensors). There are also significant constraints on how such data can be used and what trends can be reported – the use of privacy preserving data mining, especially over distributed infrastructure, becomes important in this context.

Coupling Fusion Simulations The DoE SciDAC CPES fusion simulation project is developing an integrated predictive plasma edge simulation capability to support next-generation burning plasma experiments such as the International Thermonuclear Experimental Reactor (ITER). The plasma edge includes the region from the top of the pedestal to the scrape-off layer and the divertor region bounded by a material wall. A
multitude of non-equilibrium physical processes on different spatio-temporal scales present in the edge region demand a large scale integrated simulation. The low collisionality of the pedestal plasma, magnetic X-point geometry, spatially sensitive velocity-hole boundary, non-Maxwellian nature of the particle distribution function, and particle source from neutrals, combine to require the development of a special, massively parallel kinetic transport code for kinetic transport physics using a particle-in-cell (PIC) approach. However, a fluid code is more efficient in terms of computing time, for studying the large scale MHD phenomena, such as Edge Localized Modes (ELMs). Furthermore, such an event is separable since its time scale is much shorter than that of the transport. The kinetic and MHD codes must however be integrated together for a self-consistent simulation as a whole. Consequently, the edge turbulence PIC code (i.e., XGC) has to be connected with the microscopic MHD code (i.e., M3D) to study the dynamical pedestal-ELM cycle.

The overall application workflow consists of the coupled parallel simulation codes, XGC and M3D, that will be run on different numbers of processors on different systems. The coupling begins with the generation of a common spatial grid. XGC then calculates two dimensional density, temperature, bootstrap current, and viscosity profiles in accordance with neo-classical and turbulent transport, and streams these to M3D. The input pressure tensor and current information are used by M3D to evolve the equilibrium magnetic field configuration, which it then streams back to XGC to enable it to update its magnetic equilibrium and to check for stability. During and after the ELM crash, the pressure, density, magnetic field and current will be toroidally averaged and streamed to XGC. During the ELM calculation, XGC will evaluate the kinetic closure information and kinetic $E_r$ evolution and streams them to M3D for a more consistent simulation of ELM dynamics. Note that the XGC and MHD codes use different formulations and domain configurations and decompositions. As a result, the data has to be transformed while it is being streaming using mesh interpolation module.

**Autonomic Oil Reservoir Optimization** A key issue in oil reservoirs is determining the optimal locations of the oil production and injection wells and their configurations. Optimal well locations and configurations depend on geological and fluid properties as well as on economic parameters, and determining these locations/configurations leads to a very large number of potential scenarios that must be evaluated using numerical reservoir simulations. The high costs of simulation make an exhaustive evaluation of all these scenarios infeasible. As a result, the well locations and configurations are traditionally determined by analyzing only a few scenarios. However, this ad hoc approach may often lead to incorrect decisions with a high economic impact as well as possible safety hazards. As a result, dynamic data-driven strategies that use sampling and optimization are desirable. However, the selection of appropriate optimization algorithms, the runtime configuration and invocation of these algorithms and the dynamic optimization of the reservoir are challenging problems. The process is also naturally distributed and dynamic data driven, as the data and services involved in the process are distributed and the current state of the reservoir is obtained from sensors deployed in an instrumented oilfield. Furthermore, the optimization process involves large numbers of instances of parallel simulations to be executed on distributed resources. Autonomic strategies can be used to effectively address these challenges.

For example, a typical end-to-end application scenario will consist of (i) instances of distributed multi-model, multi-block reservoir simulation components, (ii) optimization services, for example, based on methods such as Very Fast Simulated Annealing (VFSA) and Simultaneous Perturbation Stochastic Approximation (SPSA), (iii) economic modeling services, (iv) real-time services providing current economic data (e.g., oil prices), (v) historical data archives, and (vi) experts (scientists, engineers).

The overall process is autonomic in that the peers involved automatically detect sub-optimal oil production behaviors at runtime and orchestrate interactions among themselves to correct this behavior. Further, the detection and optimization process is achieved using policies and constraints that minimize human intervention. Policies are used to discover, select, configure, and invoke appropriate optimization services to determine optimal well locations. For example, the choice of optimization service depends on the size, nature and state of the reservoir. Similarly, policies can also be used to manage the behavior of the reservoir simulator, or may be defined to enable various optimizers to execute concurrently on dynamically acquired Grid resources, and select the best well location among these based on some metric (e.g., estimated revenue, time or cost of completion).

**Asynchronous Replica-Exchange Molecular Dynamics** Replica exchange is an effective
The asynchronous replica exchange formulation addresses these limitations and is better suited for large-scale distributed environments. It enables arbitrary walkers to dynamically exchange target temperatures and other information in a pairwise manner based on an exchange probability condition that ensures detailed balance. Individual walkers locally determine the ranges of interest and exchange decisions are made in a decentralized and decoupled manner, and the actual exchanges can occur between pairs of walkers 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.

**Galaxy Simulation** Galaxy and star formation using smoothed particle hydrodynamics generates large data files containing snapshots of an evolving system stored in 16 dimensions. Typically, a simplistic simulation would consist of around a million particles and may have raw data frame sizes of 60 Mbytes, with an overall data set size of the order of 6 GBytes. The dimensions describe particle positions, velocities, and masses, type of particles, and a smoothed particle hydrodynamic radius of influence. After calculation, each snapshot is entirely independent of the others allowing distribution over the distributed system for independent data processing and graphic generation. A user of the such an application would like to view the chronological changes in the Galaxy as an animation, with each simulation output written to a sequence buffer. A user would then view the contents of a sequence buffer to create the overall animation sequence. As each snapshot is independently calculated, it is possible to make use of high throughput, loosely coupled application. Three main data analysis functions are necessary: (i) file parsing – data files are parsed according to their format and the data loaded into data structures. Each data segment represents a distinct time frame or snap shot within the animation; (ii) data set projection – 3D data sets are projected down onto a 2D plane from a viewpoint; (iii) Visualization – the 2D frames are run to form the animation.

**Tissue Microarray Collaboratory** The tissue microarray (TMA) technique enables researchers to extract small cylinders of tissue from histological sections and arrange them in a matrix configuration on a recipient paraffin block such that hundreds can be analyzed simultaneously. A key advantage of TMA technology is that it allows amplification of limited tissue resources by providing the means for producing large numbers of small core biopsies, rather than a single section. Another major advantage of the TMA technique is the fact that each specimen is treated in an identical manner. As a result, TMA technology holds great potential for reducing the time and cost associated with conducting research in tissue banking, proteomics, and outcome studies. However analyzing and characterizing, and sharing TMA data presents a number of challenges. These include the large computational requirements of the analysis/characterization methods, warranting large-scale Grid-based solutions, as well as geographical distribution of the research sites across which the analyzed TMA data has to be shared as part of research/discovery workflows.

The tissue microarray collaboratory is a peer-to-peer infrastructure for imaging, analyzing, and seamlessly sharing tissue microarrays (TMA), correlated clinical data, and experimental results across a consortium of distributed clinical and research sites. Its overall goal is to facilitate cooperative oncology research.
aimed at improved understanding of the underlying mechanisms of disease onset and progression while simultaneously providing new insight into therapy planning and treatment. The collaboratory builds on Squid [?], a content based distributed hash table that supports indexing and flexible (using keywords, partial keywords, wildcards, ranges) and efficiently discovery of TMA data and metadata with search guarantees and bounded costs.

ClimatePrediction.net [?] has two intrinsic computational models, one for generating the data, and one for analyzing the data. The model for generating the data closely follows that of SETI@home [?], and in is in fact now built on the BOINC middleware [?]. Because it is running a scaled down version of a model that would traditionally be run on a high performance met office computer, the runtime is quite long. This has lead to the construction of trickles which are now integrated into the BOINC platform. Trickles are when the distributed jobs report back to the project server on a daily basis (or longer, as if the distributed host goes off line for more than a day it 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 way through the computation of its work unit. So in the case of the basic ClimatePrediction.net experiment, this takes the form of the initialization and stabilization of the model, a period with preindustrial CO2 and a phase with the CO2 levels doubled. The use of distributed computing in the data generation phase allows a 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 model for the analysis of the data is less well known. Here we have a variable number of highly distributed computing resources and data stores. The 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, it is necessary to provide some form of abstraction from the changing number of distributed resources. This was provided through a data parallel workflow language, Martlet [?] which contains a set of abstract constructs with which to build analysis functions. Other solutions such as Parallel Haskell [?, ?] are able to handle the changing numbers of computational resources, but the unique feature of this model of computing 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.

Operational Hurricane Forecasting [?] Operational hurricane forecasting is a computationally intense application with a somewhat unpredictable demand, that nonetheless requires near-real-time responsive from the infrastructure. When a tropical storm develops into a hurricane, operational hurricane forecasting must start quickly and produce the best possible estimates of the hurricane’s track, where and when it will make landfall, and its overall severity. This is to provide decision makers with best possible information for disaster mitigation efforts.

From a computational perspective, once a storm becomes a hurricane, massively parallel weather prediction models must be run that ingest data from multiple satellite image sources, aircraft sensors, terrestrial radar and other “ground truth”. These models must be run repeatedly when new data is available to constantly update and refine the predictions. 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 risk to human populations and civic infrastructures.

From an infrastructure resource perspective, it is difficult to adequately provision this entire computing effort in a timely fashion to enable effective disaster mitigation. It is not economically feasible 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; shared use of distributed resources makes the most economic sense. In this case, however, 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 are quite 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 when the resources are dynamically provisioned.
The NEKTAR - Human Arterial Tree project is motivated by a grand challenge problem in bio-mechanics - simulation of blood flow in the entire human arterial network. Formation of arterial disease such as 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 can 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. What makes this type of model amenable to distributed computing is that the waveform coupling between the sites of interest can be reasonably modeled by a reduced set of one-dimensional equations, which capture the cross-sectional area and sectional velocity properties. One can therefore simulate the entire arterial tree using a hybrid approach based on a reduced set of one-dimensional equations for the overall system and detailed 3D Navier-Stokes equations at arterial branches and bifurcations. The current human arterial tree model contains the 55 largest arteries in the human body with 27 arterial bifurcations. The inclusion of all 27 bifurcations in the simulation requires a total memory of around 7 terabytes, which is beyond the current capacity of any single supercomputing site available to the open research community in the US.

In the hybrid approach to the arterial tree problem, Nektar is used to simulate 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. 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 an enhanced performance. But by decomposing the problem into separate 1D, 3D problems this application is amenable to distribution, as typical data-transfers between the 1D-3D simulation components are much smaller, and thus internet scale communication does not lead to performance slowdown.

**GridSAT: Satisfiability**

### 2.2 Application Vectors

As shown in Table ??, each of the example applications is described by a set of *vectors*. If one thinks of the applications in a multi-dimensional space, the vectors are the axis that define the space. In a traditional space, each application would have a single value on each axis, and if two applications had the same values on all axes, they would also have the same distributed structure. 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.

The vectors that are used in the table are: Execution Unit, Data Exchange, Coordination, and Execution Environment. The last column of the table, Other Characteristics, is really a set of notes or comments, and not a vector.
<table>
<thead>
<tr>
<th>Application Example</th>
<th>Execution Unit</th>
<th>Data Exchange</th>
<th>Coordination</th>
<th>Execution Environment (for data exchange and execution units)</th>
<th>Other Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Montage</td>
<td>Individual sequential and parallel executables</td>
<td>files - mostly one-to-one, some group-to-group, one-to-all, and all-to-one; files transferred by DAG enactor</td>
<td>DAG</td>
<td>Supports dynamic process creation, executes DAG and moves files</td>
<td>(i) Message exchange is Event-driven and is tolerant to delays (ii) Centralized coordinator (iii) No control on execution ownership but component ownership is not important; ownership of resources.</td>
</tr>
<tr>
<td>Single Numerical Relativity simulation</td>
<td>multiple instances of single executable</td>
<td>concurrent coupling via messaging, bidirectional (MPI)</td>
<td>SPMD</td>
<td>MPI, co-scheduling</td>
<td></td>
</tr>
<tr>
<td>Home-based patient monitoring and data analysis. Involves period monitoring of patient data and data analysis at server</td>
<td>execution units include: (i) sensor-based data capture; (ii) data analysis of time series data</td>
<td>DAG</td>
<td>event-based - through the use of a workflow enactor</td>
<td>Web Services and Workflow (WebSphere)</td>
<td>periodic monitoring, use of pub/sub and messaging, workflow-based data analysis</td>
</tr>
<tr>
<td>Coupled Fusion Simulations</td>
<td>multiple simultaneously running parallel executables</td>
<td>loose coupling via data exchanged (streamed) between running executables</td>
<td>data</td>
<td>coupled simulations running in multiple systems</td>
<td>requires high throughput low latency data streaming, data transformation and data re-mapping capabilities, specially for strong coupling.</td>
</tr>
<tr>
<td>Autonomic Oil Reservoir Optimization (AORO)</td>
<td>parallel executables</td>
<td>data/event triggered dynamic workflow execution</td>
<td>data/event</td>
<td>decoupled coordination, content (pub/sub) messaging, dynamic discovery</td>
<td>tolerant to data/event loss/uncertainty; sensor/historical data driven workflow generation and execution, optimization with system level acceleration for parameter/uncertainty estimation, loosely coordinated workflows, opportunistic, closing the loop through simulations.</td>
</tr>
<tr>
<td>Asynchronous Replica-Exchange Molecular Dynamics (AREMD)</td>
<td>sequential or parallel executables</td>
<td>failure/latency tolerant loosely coordinated p2p</td>
<td>data</td>
<td>events/data</td>
<td>decoupled (tuple-space) coordination and messaging, dynamic task generation</td>
</tr>
<tr>
<td>Tissue Microarray Collaboratory</td>
<td>sequential workers corresponding to each TMA cylinder</td>
<td>reduction across multiple cylinders and TMAs; data sharing across multiple TMA data repositories</td>
<td>events, data</td>
<td>dynamic worker pools with loosely coupling, decoupled p2p data sharing</td>
<td>resilient to task, message failures, latencies and resource heterogeneity, opportunistically generates tasks.</td>
</tr>
<tr>
<td>Climate-Prediction.net (Data Generation)</td>
<td>master, sequential workers, and distributed data stores</td>
<td>sockets: input parameters (master to workers), status messages(workers to master), files (workers to data stores)</td>
<td>messages</td>
<td>supports dynamic worker population</td>
<td>uses BOINC with new features, such as partial results from workers and daily status messages (trickles)</td>
</tr>
<tr>
<td>Climate-Prediction.net (Data Analysis)</td>
<td>master, workers (sequential and parallel) at data stores and elsewhere</td>
<td>sockets: instructions, files (can move from data store to workers or can be processed at data store, results kept at data stores)</td>
<td>Martlet workflow, message-driven</td>
<td>distributed data store and processing</td>
<td>data locations stored in central server</td>
</tr>
<tr>
<td>Operational Hurricane Forecasting</td>
<td>parallel instances of weather prediction models, precipitation models, and flooding models.</td>
<td>data acquired from repositories and sensors, then marshaled for each model; output of one model is input for another</td>
<td>models are data driven</td>
<td>a set of parallel machines, potentially in different locations and administrative domains.</td>
<td>data outputs include intermediate and final data products from all models; must be viewable on high resolution maps. Data displays must allow decision makers to rapidly navigate both within an individual geographic data set, and also between alternative outcome data sets.</td>
</tr>
</tbody>
</table>

**Table 2: Application Characteristics**
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 that are run on processors. 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 date-capture unit that is streaming data to another piece of the application.

The next vector is *data exchange*. This explains the data flow between the executions units. Data can be exchanged by messages between entities or by files that are written by one element and then read by another. Data exchange can also be point-to-point, all-to-all, one-to-all (broadcast), all-to-one (reduce), or group-to-group. Messages can use sockets, MPI, or another mechanism.

The third vector is *coordination*, which describes the control flow between the execution units. Possible values for this vector are DAG (meaning that there is a DAG that defines the control flow), SPMD (where the control flow in implicit in all copies of the single program), data- and/or event-based (where there is a mapping of how either data or an event that enters the system causes something else to occur), or messages (which is similar to data- and/or event-driven control).

The last vector is *execution environment*. ... Additionally, Table ?? includes a final column that describes other characteristics of the application. This includes descriptions of ... fault tolerance ...

### 2.3 Application Classes

One can find clusters of applications in the multi-dimensional space defined by the application vectors. Each cluster is named and described in Table ???. The column labeled “Properties” described the values of each vector for each application class.

Loosely Coupled:

Tightly Coupled Homogeneous:

Tightly Coupled Heterogeneous:

Loose Coupling of Tightly Coupled:

Event-Oriented:

Dynamic Data Driven:

Irregular/Dynamic: Additionally, there are “first principles” Grid applications, such as GridSAT [?], and applications that are based upon resource aware “learning” algorithms [?], which need to explicitly marshal distributed resources. For these applications the resource utilization is often dynamic and unpredictable; interestingly, the resource requirements and utilization might possibly be dependent upon both the execution trajectory and underlying infrastructure.

Interactive:

Pleasingly Distributed:

### 3 Programming Models and Abstractions

Based on section ??, we can deduce that: (a) the emerging computational infrastructure (Grid/Cloud computing) that will support future scientific and engineering investigation will be distributed, and will present significant challenges in terms of scalability, performance, reliability, heterogeneity and dynamism, and (b) while several applications (many of which were originally traditional HPC applications) have effectively used this infrastructure to advance understanding in their disciplines, achieving this has required the applications to address the above challenges at the application level, middleware level, or both. For example, the asynchronous numerical formulations used by the replica exchange application can deal with non-uniform communication latencies and message loss. Similarly, the TMA Collaboratory and the Oil Reservoir Applica-
<table>
<thead>
<tr>
<th>Application Class (Cluster)</th>
<th>Properties (Values of Application Vectors)</th>
<th>Application Examples</th>
<th>General Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loosely Coupled Applications</td>
<td>Independent execution units; loosely coupled; data transfer through files.</td>
<td>Montage</td>
<td>Each execution unit is independently developed, and only exposes an I/O. Only an executable instance of each execution unit may be available. This application class involves a loose coupling between such executable instances.</td>
</tr>
<tr>
<td>Tightly Coupled Homogeneous Applications</td>
<td>Execution units interact via messaging or other mechanisms; strong dependency between units</td>
<td>Single Num. Rel. Simulation</td>
<td>Trivial generalization of MPI to WAN using MPICH-G2 and variants thereof, should also be considered here.</td>
</tr>
<tr>
<td>Tightly Coupled Heterogeneous Applications</td>
<td>Execution units interact via files; strong dependency between units; units may be implemented using different programming libraries</td>
<td>Molecular Dynamics Simulation (with separate components implementing different capabilities – such as force and velocity calculations, position of particles, etc)</td>
<td></td>
</tr>
<tr>
<td>Loose Coupling of Tightly Coupled Applications</td>
<td>Many applications need to be used in conjunction with other applications; they may not necessarily have been designed for this <em>ab initio</em>. Alternatively, certain commonly used algorithms call for replacing single long-time running simulations with multiple shorter time-duration simulations – but with possible infrequent communication between the individual simulations.</td>
<td>Replica Exchange simulations for protein folding; multi-physics scientific applications</td>
<td>There are two levels of communication; the first is internal to a single job/task (think monolithic MPI job), the second level is communication between the jobs/talks. The latter is less frequent and thereby more tolerant of latency and delays. Coordination of the many tasks/jobs is challenging.</td>
</tr>
<tr>
<td>Event-Oriented Applications</td>
<td>Execution units employ a publish/subscribe mechanism to coordinate; generally loose coupling between units</td>
<td>Distributed database search</td>
<td></td>
</tr>
<tr>
<td>Dynamic Data-Driven Applications</td>
<td>Covers a broad range of applications. Applications for which it is difficult to predict the resource requirements (i.e.; how long, how many). Dynamic in the sense that the resource requirements are constantly changing. Dynamic behavior maybe one cause of hard-to-predict / irregular behavior, but not the only.</td>
<td>GridSAT, Ensemble Kalman Filter(s)</td>
<td>In GridSAT there is a lot of sub-problems (sub-clause) solving, the complexity of which differs significantly. For Ensemble Kalman Filters, there is a global synchronization point, where all sub-simulations need to finish before next-stage models can be generated, but each sub-simulation varies both in complexity and time taken. Hence overall execution time is unpredictable.</td>
</tr>
<tr>
<td>Interactive</td>
<td>Execution Units need to be co-scheduled, and are (generally) closely coupled</td>
<td>Computational Steering and Multi-Party Scientific Visualization; Video Conferencing</td>
<td>There is a need for more than one specialized resource, such as compute and visualization, with well defined QoS lower-bounds. One of the few distributed applications where latency can be critical.</td>
</tr>
<tr>
<td>Pleasingly Distributed</td>
<td>Multiple instance of execution units exist which may be replicated. Each unit operates on local data</td>
<td>SETI@HOME; Folding@HOME; ClimatePrediction.net (data generation)</td>
<td></td>
</tr>
</tbody>
</table>

*Table 3: Characterizing Application Classes*
tions used self-organizing Peer-to-Peer overlays to address the dynamic system behaviors and dynamic data availability respectively. Other examples include popular middleware systems such as MPI, which have been extended to address requirements of wide-area scalability (MPICH-G) and fault-tolerance (FT-MPI), composition (workflow) system to enable the specification and enforcement of strict QoS requirements (ASKALON), and component/service-based programming systems extended to support dynamic resource-sensitive bindings (Ibis) and policy-driven runtime adaptations (Accord).

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 High Performance Distributed Computing (HPDC). Specifically this discussion will focus on composition and coordination, systems, components/service models and communication infrastructures. It is useful to note that composition and coordination provide the common themes that bring together the ideas in this section with those in sections ?? and ???. For instance, a number of workflow systems currently exist to support the construction of scientific applications by combining components and services. Although sharing a number of similarities, they also have differences in the way the workflow is described, or 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.

They (Lee & Talia) focused on frameworks etc.; we’re different – focus not on environment or tools but the focus is on application and application developers

Justify this approach

3.1 Categorization

The intent of this section is to define and describe the set of programming models and abstractions that the theme believes will capture all of the current work in distributed/grid computing. The models can be classified into four categories: Composition, Component Models, Communication, and Services

3.1.1 Composition

The first group, Composition, includes a number of items, such as: scripting, workflow, dataflow, functional languages, skeletons, and superscalar [?]. 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. However, this does not include the use of formal component systems, as this is discussed separately in section ???. When a unit is executed, it is instantiated, started, runs to completion, and then is removed from the system (unlike formal component systems.) Important issues here are the ordering of the pieces, and the flow of data or messages between the pieces. In many cases, the user has a GUI of some sort that can be used to specify ordering. In other cases, only dependencies are decided by the user, and some automated system then decides the ordering that will be used to completely run the applications. Another issue is the resources on which the pieces are run. This may be determined by either the user (explicitly) or by an automated system (implicitly.) Table ?? shows the relationship of these items.

3.1.2 Component Models

The second group is Component Models. Inside this group are DObjects [?], CORBA [?, ?], CCA [?]. This group is based in the area of object-oriented programming, and specifically, distributed objects. The main idea of all of the components models is that each component, which is really an object, is instantiated within a framework that provides some standard services and actually composes the components, and where one component can only interact with another component through an explicitly-specified portion of their interfaces. This group is 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, whereas in the Composition group, there is no framework or set of rules other than those created implicitly by the application developer. Additionally, for a component model, some of the methods of a component are permitted to be used by other components. In CORBA and CCA, this portion of the interface is defined in a language-independent manner, through IDL or SIDL, respectively,
where IDL is interface definition language, and S is scientific. DObjects is a purely-JAVA system, so it does not require any language independence. The issue of how the location (or distribution) of the components is determined is not specified by by CORBA or CCA, although it appears to be automated for DObjects, based on a metacomputing resource brokering mechanism. For CORBA, the framework is the object request broker (ORB). For CCA, there are multiple frameworks, some of which handle parallel computing, some of which handle distributed computing, and none of which currently handle both. For DObjects, the framework is H2O[?]. There are issues related to components and parallelism that haven’t been solved, such as how a parallel component running on 4 processors communicates with a different parallel component running on a different 4 processors. And this becomes even more complicated if the numbers of processes for the two components are not the same.

### 3.1.3 Communication

The third group, Communication, includes a number of items, such as: two-sided messaging, RPC and other one-sided messaging, event-driven systems, transactional systems, streaming, asynchronous operations, shared data, shared files, and shared address space. The main idea that ties these together is how the pieces interact with each other. Remote procedure calls (RPC) means that one pieces has exposed a procedure interface to the other piece, and that second piece can then call the exposed procedure within the first piece, eventually leading to a return value from the procedure. Two-sided messaging means the two pieces com-
communicating both have agreed to exchange data, and both participate in the exchange. One-sided messaging means that one piece can manipulate (get, put, or change) memory that belongs to the other piece, without the other piece being involved or perhaps even aware. Event-driven systems refers to a situation when the activities of each piece are driven by activities in other pieces, and a piece which is not responding to another piece is idle or waiting. In transactional systems, there is a sense that some state exists in each piece, and that this state changes based on activities from another piece. Once the state has finished changing, it is committed and becomes the new state. However, until that time, the state can easily be rolled back to the previous state without harm to the overall system. Streaming implies that there is a continuous flow of data from one piece to another, and that the second piece does some processing to the data, and likely sends one or more streams to other pieces. Shared data is a model where physically distributed memory is logically shared, with communications occurring as needed. Shared files is a model that assumes a shared file system, where one piece writes data to a file to be read by a second piece. Shared address space is a model in which threads can migrate across distributed systems which have physically distinct address spaces. This is implemented by communicating the state of a thread from one processor to another, killing the thread on the first processor, and starting a new thread with the transferred state on the second processor. Table ?? shows these items and their relationship.

**UNIFY (from Kentucky)** A system which implements wide-area shared memory; candidate for shared-data (and not parallel)

### 3.1.4 Services

The final group is called **Services**, and it includes semantic services, groups, and agents. A service is more coarse-grained than a component or object, and constitutes a relatively large, intrinsically un-associated, 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 identifying substitute 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.

<table>
<thead>
<tr>
<th>Model/Abstraction</th>
<th>Example</th>
<th>Support Async?</th>
<th>Support Sequential, Concurrent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPC</td>
<td>GridRPC, RMI</td>
<td>Y</td>
<td>Seq, Con</td>
</tr>
<tr>
<td>Two-sided messaging</td>
<td>MPICH-G2</td>
<td>Y</td>
<td>Seq, Con</td>
</tr>
<tr>
<td>One-sided messaging</td>
<td>MPICH-G2</td>
<td>Y</td>
<td>Con</td>
</tr>
<tr>
<td>Event-driven</td>
<td>Narada Brokering [?], Flaps [?]</td>
<td>Y</td>
<td>Seq, Con</td>
</tr>
<tr>
<td>Transactional</td>
<td>?</td>
<td>Y</td>
<td>Seq, Con</td>
</tr>
<tr>
<td>Streaming</td>
<td>Data Turbine [?]</td>
<td>N</td>
<td>Con</td>
</tr>
<tr>
<td>Shared Data</td>
<td>CAF [?], UPC [?], Titanium [?]</td>
<td>Y</td>
<td>Con</td>
</tr>
<tr>
<td>Shared Files</td>
<td>MatlabMPI [?]</td>
<td>Y</td>
<td>Seq, Con</td>
</tr>
<tr>
<td>Shared Address Space</td>
<td>D-CVM [?]</td>
<td>Y</td>
<td>Seq, Con</td>
</tr>
</tbody>
</table>

*Table 5: Communication Group*
A common theme of the “services” category is the encapsulation of loosely coupled, generally coarse-grained, functionality that has associated metadata annotations. The metadata annotations enable the use of these services in a variety of contexts.

3.2 Comparison to Other Work

This categorization was initially undertaken without reference to any particular existing effort, but afterwards compared with published work of Lee & Talia [?], Parashar & Brown [?], Soh [?], and Rabhi [?] and found to generally agree with these works, with differences that were well-understood.

Metrics for comparison:

- Models/abstractions
- Tools/Languages and Environments
- Applications

Lee & Talia [?]: This paper discusses properties and capabilities of grid programming tools. It 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 & configuration management. There is no discussion of specific applications.

Parashar & Brown [?]: This is primarily an analysis of Grid systems, with some discussion of tools/languages and environments. The treatment of programming models is confined to i) communication models and frameworks, ii) distributed object models, iii) component models, and iv) Service Models. No specific application classes are discussed.

Soh [?]: Section 2 of this paper 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) mappings to either.

Rabhi [?]: This book 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 (often described in a formal way with focus on high-performance systems) and a design pattern (often described using English and/or a combination of UML diagrams, and with a much broader focus). The book focuses 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 – and greater emphasis on techniques that may be use to a computer scientist.

4 Patterns to Support Distributed Applications

Phil Murray

The exploitation of patterns[?] as tools in the parallel programming process is increasingly widely promoted. For example, Snir [?] proposes that “patterns will not only act as a tool in evaluation of programming models, but also as a cookbook that will guide the programmers in making decisions in every stage of the parallel software development”, while Patterson et al. [?] propose 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”.

In similar vein, the aim of this section is to explore the ways in which a pattern-oriented programming model might enhance the distributed programmer’s ability to produce good (e.g., 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 \textit{composition} of more primitive operations or patterns.

Patterns allow good practice to be identified, and shared across application domains. A pattern is generally defined in an application independent manner, and used to encode particular useful behaviors. Patterns are aimed at capturing generic attributes which may be further instantiated, parameterized and/or refined to lead to an implementation. These are important issues for distributed computing applications, which generally need to operate in dynamic environments. Providing patterns will ease the task of distributed application developers, who may instantiate previously generated templates and then incrementally refine them based on their experience of use. Before proceeding to examine our pattern collection in section ??, we present some introductory remarks on the application of pattern concepts to distributed programming.

Patterns in the parallel software engineering sense [?] 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” [?], in the form of a concrete library, template or problem solving environment of some kind. For our purposes, as defined in section ??, patterns are abstractions of frequently occurring compositions of abstractions. The programmer is responsible for instantiation and specialization of the pattern implementation with application specific details, within a pre-prepared framework which provides pattern-generic code. For example, it would simple to present both informal and formal descriptions of a task-farm, and indeed we might argue that Condor represents a concrete instantiation of the softer task-farm concept. We will focus on concrete support for patterns in section ??.

A key aspect of distributed programming (in contrast to conventional parallel programming) is that handling characteristics of the environment in which an application is deployed can become as challenging as describing the core application itself, or may even be an inherent component of the problem definition. For example, in an “AtHome” application, the programmer may decide that the environment (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. Our collection includes patterns which span a range of positions, from those which correspond to existing parallel programming patterns, through some whose description encompasses issues which are driven by distributed concerns but which have some semantic content in the application space (e.g. the deliberate use of redundancy in “AtHome” applications) through to those which deal exclusively with challenges posed by deploying an application in a distributed environment.

4.1 Application Patterns in the Wild

In reviewing a collection of applications, including those from section ??, we have identified a number of application patterns. Most of these are already recognized as such in the patterns literature, so it is reassuring to see them appear in real applications. We believe that the \textit{AtHome} and \textit{Consensus} patterns have not previously been formalized. Table ?? summarizes the relationships between our applications and patterns: a • indicates that an application exhibits a pattern, either as its overarching structure, or within one or more components or phases.
<table>
<thead>
<tr>
<th>Applications and Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Montage</td>
</tr>
<tr>
<td>Montage</td>
</tr>
<tr>
<td>NumRel</td>
</tr>
<tr>
<td>MolDyn</td>
</tr>
<tr>
<td>RepEx</td>
</tr>
<tr>
<td>DDBSrch</td>
</tr>
<tr>
<td>ResMod</td>
</tr>
<tr>
<td>PatMon</td>
</tr>
<tr>
<td>GridSat</td>
</tr>
<tr>
<td>CompSteer</td>
</tr>
<tr>
<td>VidConf</td>
</tr>
<tr>
<td>SETI</td>
</tr>
<tr>
<td>OilOpt</td>
</tr>
<tr>
<td>Climate</td>
</tr>
</tbody>
</table>

*Table 6: Applications and Patterns*
4.1.1 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 back 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 are working on the same task with different input parameters, and the master is assembling the outputs into a some concatenated or derived product.

4.1.2 Client/Server

*Client/Server* is a well-known pattern, long established in the literature [?]. Clients requests services and wait for answers. The server fields and responds to requests. These are very common in distributed systems, e.g. database servers or print servers. Some programming languages support this concretely, for example the Java Distributed Object Model (JDOM) [?] underlying Java RMI explicitly identifies and provides support for clients and for servers. Similarly Erlang has a *generic server* behavior [?].

4.1.3 Publish/Subscribe

The *Publish/Subscribe* pattern allows greater flexibility than *Client/Server* because the publishers are not programmed to send their messages to specific receivers, i.e. subscribers or clients. Rather, published messages are characterized into classes and subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of what (if any) publishers there are. Subscribers may also republish messages. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology. The publish/subscribe pattern is closely related to event-driven programming and the *observer* design pattern. Example publish/subscribe systems include RGMA [?], Conoise-G [?].

4.1.4 Peer to Peer (P2P)

P2P generalizes the client server pattern by removing the distinction between clients and servers: a peer may act as both client or server. P2P distributed systems are are less centralized, and hence more scalable than a client server system which typically has relatively few servers. P2P is also most common in *ad hoc* networks. Example P2P systems include file-sharing systems like FTP [?] or Napster [?], or mail systems like SMTP [?].

4.1.5 Pipeline

Similarly well-known [?], the *Pipeline* pattern is distinguished from *Client/Server*, by the unidirectional flow of information, and the chaining together of several Producer-Consumer relationships. It can also be viewed as a restricted publish/subscribe topology.

4.1.6 Supervision Trees

A common means of engineering fault tolerance in distributed systems is to require one process to monitor the existence and correct performance of other processes. These *supervising* processes may themselves be supervised, and hence trees of processes supervising others are constructed. The Erlang programming languages supports supervision by providing the *supervisor* behavior [?].

4.1.7 Task Farm

*Task Farm* is a well-known pattern in the parallel programming literature. 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 and Apples are in effect, concrete instantiations of Task Farm.

4.1.8 Replication

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

4.1.9 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, which may further manipulate returned results. Its use of replication also makes it an instance of Replication. Here, replication allows the application to become more robust in the face of worker failure or of malicious or inadvertent miscalculation. 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.

4.1.10 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. Within AtHome applications, consider the decision on when to deem a replicated problem “solved”. In different applications this could be when two agreeing solutions have been returned, or a majority, or a unanimous complete set, and the definition of “agreement” may similarly require to be application specific. The pattern provides an implementation of the supporting marshaling of replicas. Thus, we can view Consensus as itself being both a component of AtHome and also a worthwhile pattern in its own right.

4.1.11 AllPairs

The All-Pairs pattern [?] abstracts computations in which a single function must be applied independently to all members of the cross-product 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.

4.1.12 MapReduce

MapReduce [?] 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.

4.1.13 Bag of Tasks

The Bag of Tasks pattern [?] is related to Task Farm, but distinguished by the fact that new tasks may be created and added to the “bag” dynamically. It is classically implemented in a Master/Worker style, though the Master has no application content, and operates merely as the guardian of the bag.

4.1.14 Stream

Streaming captures a communication pattern in which a unidirectional flow of information between agents can be processed as it arrives (in other words, there is interleaving and/or overlap of communication with computation).
**Application Example** | **Execution Environment (for data exchange and execution units)**
---|---
Montage | Supports dynamic process creation, executes DAG and moves files
Single Numerical Relativity simulation | MPI, co-scheduling
Home-based patient monitoring and data analysis. Involves period monitoring of patient data and data analysis at server | Web Services and Workflow (WebSphere)
Coupled Fusion Simulations | coupled simulations running in multiple systems - component models
Autonomic Oil Reservoir Optimization (AORO) | decoupled coordination, content (pub/sub) messaging, dynamic discovery
Asynchronous Replica-Exchange Molecular Dynamics (AREMD) | decoupled (tuple-space) coordination and messaging, dynamic task generation
Tissue Microarray Collaboratory | dynamic worker pools with loosely coupling, decoupled p2p data sharing
Climate-Prediction.net (Data Generation) | supports dynamic worker population
Climate-Prediction.net (Data Analysis) | distributed data store and processing
Operational Hurricane Forecasting | a set of parallel machines, potentially in different locations and administrative domains.

*Table 7: Application Environments - temporary table for working on this section (??)- will be deleted — need to figure out why each app really wants and why it uses the tool it does*

### 4.1.15 Co-allocation

Applications vary in the extent to which their components are expected or required to be active concurrently. For example, the typical DAG-based loosely-coupled computation, with files as the communication medium, does not require genuine concurrency (although it may be advantageous). In contrast, for a more tightly coupled application, with communication via MPI, it is essential - the application will fail without it. The process of ensuring (through look-up and 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 which will exploit it.

### 4.1.16 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 ??.

Well-known broker-based systems include the Common Object Request Broker (CORBA) [?], [?] and CCA [?], as discussed in section ??.

### 5 Environments for Supporting Distributed Applications

**Dan**

The applications, patterns, etc. that have been discussed previously are dependent and perhaps even inter-related to the environment in which they are (planned to be) run. For example, the MapReduce pattern doesn’t make sense without distributed file system. In some cases, it may be that the intended environment determines how an application is developed: which patterns are chosen, etc. In other cases, an application may have specific requirements on the environment. This section will discuss these environments, by which we mean that parts of the overall system that are needed to make the applications run, but that are not parts of the applications per se.

#### 5.1 Frameworks

Such as Ibis, Cactus, CCA/ccaffeine/XCAT/SciRun 2/Uintah ...

parameter sweeps: NIMROD/Apples

Need to categorize these into named categories, then match the real things here with the categories. These categories then need to be specified in the column of table 2 - meaning need to go back to table 2 and use the categories that are defined here, maybe adding in the text near there to see this section for the definitions of the categories.

Also mention @Home, BOINC
Figure 2: A high level classification of applications into Grid aware versus Grid unaware. This diagram provides a concrete realization of the relationship between abstractions, models, implementations and pattern for the set of abstractions provided by libraries (SAGA). Grid Application Library (SAGA), Application Support Tools (Frameworks) and Service/Resource abstraction layer (co-scheduling layer?) are all examples of “environments for supporting distributed applications”.

5.2 Libraries

The use of libraries is a very powerful mechanism to abstract the complexity and management of the details of distribution, as well as an simple way to provide portability and standardization.

For example, the SAGA grid-library is an implementation of several different programming models (rpc, communication, job launch etc.).

Any application can, to a first approximation, be thought of as a Grid-aware application if it is cognizant of the underlying distributed infrastructure, and/or for which there is a requirement to explicitly exploit, run or utilize the distributed infrastructure. A simple example of an application that uses a Grid-Library, is the class of tightly coupled MPI jobs – which are generalizations of parallel applications to parallelized and distributed applications. For example, MPICH-G2 applications require cognizance of the distributed resources in their RSL description. Both portals and meta-schedulers qualify as Grid-enabled Programming Environments as shown in Fig. ??.

5.3 Virtualization

either need to find an app in section 2 that needs this, or remove this subsection...

5.4 Tools to Support Patterns

such as Hadoop...

names of systems, libraries, codes that implement the patterns in section 4.

need to think about if we should talk about anything in this subsection that isn’t directly tied to an application - probably ok in this subsection but not in the other subsection.

6 Critical Assessment of Programming Distributed Applications

Jon Shantenu Manish

• What can we say about Usage Modes in our Gap analysis. Infrastructure needs to consider usage modes from the perspective of applications. Can this be distilled based on previous discussion of patterns – and does this emerge?

• Discuss issues related to non-functional attributes – such as security – especially if these are to be considered as first class entities. It would be useful to discuss here that such attributes/requirements impact the use of particular patterns. Aim here is primarily to flag that we are aware that this is an issue, but this has not been the focus of this paper.

The interdisciplinary nature of scientific discovery requires the integration of distributed data and computation at unprecedented scales. Despite the overwhelming need, the emergence of large-scale high performance distributed applications has been slow for two reasons:

• 1) distributed applications often appear to be a ”one-off” and not sustained - few distributed applications appear at all, and even fewer in ”production”

• 2) distributed applications are slow and time-consuming to develop
We believe that the reason for this are two inherent sources of complexity unique to the class of applications under study:

- 1) complex infrastructure interaction must be managed: fault tolerance, security, heterogeneity, ownership

- 2) complex patterns, models, and features must co-exist within the *same* application: e.g. master-worker, broadcast, encryption, pipelines, etc.

This complexity can be mitigated by closing the gap revealed in the analysis of applications, patterns, and supporting tools in the prior sections. There are gaps at multiple levels. First, examination of Table X in Section Y reveals that several of the hard patterns, are not supported directly by tools. Second, distributed applications often require several co-existing patterns, both hard patterns and infrastructure patterns. The tools that support different but needed patterns often do not interoperate. So the designer must make a choice as to which tool she can use directly, and then invest effort to either re-implement needed pattern functionality, or to perform custom integration of tools. In both cases significant programming effort is required, a barrier to developing or extending distributed applications. In summary, we believe that this paradigm can get the application designer maybe 10% of the way, but much more is possible in our view.

A simple example. Condor provides the ability to run arbitrary tasks in a distributed system. However, there is no direct support for the master-worker pattern, replication, and consensus. On the other hand, BOINC does not support the execution of arbitrary tasks (applications must be pre-installed), but does support master-worker, replication and consensus. It would be useful if an application that requires all of the patterns supported by both systems, could easily use both tools. Over time, some tools such as Condor, have realized these gaps, and have added new patterns or features over time. However, this is not a scalable solution in the long run.

It is not possible to anticipate all of the patterns that distributed applications will need down the road. There are just too many moving parts. Changes to infrastructure and application requirements cannot be predicted. For this reason, we do not believe a single complex tool is the right model. Instead, we advocate an approach in which tools provide open interfaces to enable more easy integration with other tools. Whether this can be best achieved by new extensible tools or ”tool orchestration” is an open question. This does not preclude the internal evolution of existing tools but this is not enough in our view. Tool designers are encouraged to think about ways in which tools can be extended and made interoperable. If enough buy-in is achieved, one could imagine developing standards for tool interoperation. We believe this could reduce the time-to-build for distributed applications, which would in turn, increase their number (points 1 and 2 above).

The question of longevity and sustainability is another issue. As a guiding vision, we can perhaps draw inspiration from what has happened in the parallel computing world. MPI provides a write-once, run-many, model. If a new parallel machine is introduced, an MPI program can be recompiled to run, perhaps with some performance tweaking. If a sufficiently rich set of interoperable tools are made available, can we make progress toward this vision for distributed applications?

BEGIN OLDER MATERIAL - not sure relevant Gap Analysis: A first pass.

Previously missed programming/application construction opportunities (i) ”If I tried to implement Application X using Model A, would I succeed” (ii) Support for Application by providing abstractions (iii) Will people write applications if ”patterns exist”...Possible Usage Scenario: Don’t want to rewrite hurricane modeling code want help to compose it...

Identify the patterns This should/will inform the tool developers (intermediate user) and possibly not the end-user (application)

Provide examples of the kinds of tools that could possibly be developed to support applications to use the identified patterns more naturally... for example does your favorite workflow enactment engine support bulk operations?

What is the support for these patterns?
Other possible analysis/assessment directions? What about infrastructure/resource providers? Especially if we go with resource/scheduling patterns.

Could we construct novel distributed algorithms from i) patterns ii) programming abstractions? Case study using Galaxy simulation, Triana and Patterns

Sketch out the different layers/levels of abstractions that an application developer thinks of.. and needs. For example, first decompose the application based upon functional requirements, then the abstractions of how to capture the functional requirements, then compose the different units of the applications, and (possibly) finally how to deploy and run the composed application

END OLDER MATERIAL

7 Conclusions

Acknowledgements