# Programming Abstractions for Clouds

Shantenu Jha[12], Andre Merzky[1], Geoffrey Fox[34]

[1] Center for Computation and Technology, Louisiana State University
[2] Department of Computer Science, Louisiana State University
[3] Community Grids Lab, Indiana University
[4] Department of Computer Science, Indiana University

August 18, 2008

**Abstract**

Clouds seem like 'Grids Done Right', including scalability, transparency, and ease of management. Virtual Machines are the dominant application environments for compute Clouds, however, that does not make application programming any less relevant than "non-virtualized" environments. The limited set of successful Cloud applications show that distributed programming patterns of the type of MapReduce and All-Pairs are required to make Cloud infrastructure a viable compute environment for a large class of problems. The existence of multiple implementations of these programming paradigms also makes clear, that application portability is, even for Clouds an emerging problem which needs addressing beyond the level of system virtualization. This paper discusses these and other challenges around cloud applications programming and development, and through a discussion of several applications, demonstrates potential solutions. We discuss how using the right abstractions – programming interfaces, frameworks that support commonly occurring programming and execution patterns – enable efficient, extensible and importantly *system-independent* implementations of common programming patterns such as MapReduce, i.e. same application is usable seamlessly on both traditional Grids and Clouds systems. We further discuss that lessons learned from programming applications for Grid environment also apply, to some extent, to Cloud environments.

## 1 Introduction

Going by interest garnered, Clouds seem to have emerged as a clear winner of the (perceived) battle of distributed infrastructures for a subset of applications, and for the time being at least. They allow loosely-coupled, data-intensive applications to be run with an ease, and with absolutely competitive scalability and throughput. Not surprisingly, dominant cloud applications utilize novel and hitherto, Cloud specific computing paradigms, such as MapReduce, BigTable, or Hadoop. These paradigms are supported by the inherent system capabilities of todays Clouds, which we call *affinities* [?]. Section ?? discusses these in some more detail.

For Clouds to be relevant to the wider scientific computing community and in general, beyond internet-backed computing, it is important to understand how other application classes will fare when migrated to Clouds, or Cloud-like distributed systems. At the moment it seems unclear if the system properties and affinities as offered by todays Clouds support, or even allow, for other application types to perform equally well. We will discuss several application classes, which seem to be most relevant for the academic computing community, in Section ??. Instead of just waiting for the 'right' affinities to emerge with new Cloud incarnations, we propose to rather attempt to predict what these affinities are, by abstracting the relevant programming patterns for these application classes, and deriving the respective system properties required to support these programming patterns (see Section ??).

We outline two specific applications, which to the best of our knowledge have not been used on Clouds: the first a replica-exchange based applications, which belongs to loosely-coupled ensemble of tightly-coupled, homogeneous distributed applications. The second example is MapReduce based application, as an example of loosely-coupled data driven applications. We specifically discuss dominant programming pattern, and describe exemplary Grid based implementations. We will then discuss, how these implementation change (i.e. simplify) for a Cloud system with the appropriate system affinities.

We will conclude our discussion with a proposed procedure for defining scientific-computing oriented Cloud properties, in Section **??**.

## 2 Cloud Usage Modes and System Affinities

In [**?**] it was shown how Grid system interfaces (in particular for general purpose Grids) tend to be complete (i.e. they try to expose a complete set of available system capabilities), whereas Cloud interfaces tend to be minimalistic (i.e. they expose only a limited set of capabilities, just enough to 'do the job').

### 2.1 Usage Modes

It is important to understand the reason for this difference. In our experience, general purpose Grids are mostly designed bottom-up: existing, often heterogeneous resources are federated as VOs, and their combined capabilities, plus additional capabilities of higher level Grid services, are offered to the end-user. This is not applicable for Clouds: the design of Clouds seems to be, mostly, top down. Clouds are designed to serve a limited, specific set of use cases and usage modes, and the Cloud system interface is designed to provide *that* functionality, and no other. Furthermore, the Cloud system itself, and in particular its high level services, may be designed to implement specific target use cases, while not supporting others (e.g., a Cloud could be homogeneous by design). These differences do not imply that Clouds are trivial to implement. In practice the opposite is most likely true (due to issues of scale, amongst other things). Clouds may very well build upon general purpose Grids, or narrow Grids, and at least face the same challenges; but their system interfaces do not expose those internal capabilities.

Specific users and user communities tend to create different applications but with shared characteristics. For example, the particle data community tends to focus on very loosely coupled, data intensive parameter sweeps involving Monte Carlo simulations and statistical analyzes. Systems used by these communities are thus designed to support these application classes before others.

The *Usage Mode* tries to catch the dominant properties of the main application classes, insofar they are relevant to the design of the system, and to the operational properties of the system. For example, the usage mode *'massively distributed, loosely coupled'* implies that the system's design prioritizes on compute resources (e.g. cycle scavenging, large clusters), and to a lesser degree on communication (no need for fast links between application instances), or on reservation and co scheduling.

In contrast, the usage mode *'massively distributed, tightly-coupled'* would imply a system's design to focus on compute resources, but importantly also on fast communication between near nodes, and on (physical) co-location of processes.

### 2.2 Affinities

Currently Clouds seem to be designed to mostly support exactly one usage mode, e.g. data storage, *or* high throughput computing, *or* databases, etc. This does not preclude Clouds targeting more than one domain or usage mode, however. The overarching design guideline to support the main target usage modes of Cloud systems, we defined as its ***affinity***. In other words, affinity is the term we use to indicate the type of computational characteristics that a Cloud supports. That property can very often be expressed as the need to use different aspects or elements of a system *together* (hence the term 'Affinity', in the sense of 'closeness').

For example, the usage mode *distributed, tightly coupled* implies that an application requires the use of multiple compute resources, which need to be 'near' to each other, together with fast communication links between these compute resources. The system needs to have a *compute-communication affinity*, and a *compute-compute affinity*.

Affinities are, however, not always mappable to 'closeness'. For example, we say that a system that supports 'persistent storage, data replication, data intensive' usage mode, may have 'bulk storage affinity' – in the sense that it needs to be designed to have bulk storage properties (availability guarantees, long term consistency guarantees etc). This example also shows that affinities are, in some sense, related to Quality of Service (QoS) properties exposed by the system, and thus to Service Level Agreements (SLAs) about these qualities of service.

### 2.3 Affinities and Programming Abstractions

Affinity is thus a high level characterization of the kind of application that could be beneficially executed on a particular Cloud implementation, without revealing the specifics of the underlying architecture. In some ways, this is the "ideal abstraction" for the end-user who would like to use infrastructure as a black-box. Some classic examples of affinity are: tightly-coupled/MPI affinity, high-throughput affinity (capacity), fast-turnaround affinity (capability), or bulk storage affinity. Our observation is that Clouds have at least one affinity, a corollary to which is that Cloud system interfaces are, designed to serve at least one specific set of users or usage modes

An affinity being the 'ideal system abstraction' has another important consequence, as it allows to express suitable programming abstractions easily, and natively. For example, it is certainly possible to implement MapReduce on a general purpose Grid, with no affinity supporting data replication, or data-compute-colocation. The implementation of that abstraction, i.e. the Map Reduce application framework, must then however implement these capabilities itself, *on top* of the system it uses. On the other hand, if a data/compute affine cloud provides these capabilities natively, as is the case for, for example, googles proprietary cloud with its google file system [?], then the MapReduce framework can focus on the core logic of the programming abstraction, i.e. on the algorithmic abstractions, and is thus much easier to implement. Note that for the application using the MapReduce framework, there is no difference [?].

Clearly, we are arguing for a separation of concerns: we argue that application frameworks should not have to deal with exposing, expressing, or implementing capabilities which are required *by* them, but are not part of their algorithmic core. Those should be provided at the system level, which makes the application frameworks *easily* implementable on any system providing these capabilities, i.e. on any system, which has the appropriate affinity.

## 3 Distributed Applications Usage Modes

Table **??** shows an overview of a number of application classes [?] which are widely used in scientific computing, and outside. Often applications in the same class, have similar programming models or use programming patterns; for example, *'pleasingly distributed'* applications, such as the numerous *'XYZ@Home'* type applications, all share the Master-Worker model, in one incarnation or the other. As compute affine Clouds (aka compute Clouds) support that programming paradigm, these applications can immediately utilize compute cloud resources with great success. For other application classes, such as *'tightly coupled, heterogeneous'* applications, this is not so obvious, as a compute cloud without compute-communication affinity can not easily run a communication intensive application efficiently.

| **Application Class** | **Data Driven** | **Compute Driven** |
|---|---|---|
| **Pleasingly Distributed** | SETI@home | Monte Carlo Simulations of Viral Propagation |
| **Loosely Coupled, Homogeneous** | Image Analysis | Replica Exchange Molecular Dynamics of Proteins |
| **Tightly Coupled, Homogeneous** | Semantic Video Analysis | Heme Lattice-Boltzmann Fluid dynamics |
| **Loosely Coupled, Heterogeneous** | Multi-Domain Climate Predictions | Kalman-Filter Fluid Dynamics |
| **Dynamic Event Driven** | Disaster support | Visualization |
| **First Principle, Distributed** | MapReduce-Based Web indexing | MapReduce-Based Motif Distributed search |

*Table 1:* **Examples of primary categories of distributed applications[?].**

We want to demonstrate using two examples, how the implementation of the respective programming patterns used by these application classes can be supported by the Cloud affinities[1].

## 3.1 Example 1: MapReduce

MapReduce [?] is a programming framework which supports applications which operate on very large data sets on clusters of computers. MapReduce relies on a number of capabilities of the underlying system, most related to file operations, but also related to process/data colocation. The Google file system, and other global file systems, provide the relevant capabilities, such as atomic file renames. Implementations of MapReduce on these file systems can focus on implementing the the dataflow pipeline, which is the algorithmic core of the MapReduce framework.

We have recently implemented MapReduce in SAGA, targeting general purpose Grid systems, where the system capabilities required by MapReduce are usually not natively supported – instead, a general purpose grid provides a much larger set of lower level operations. Some semantics, such as again the atomic file rename, is provided by the SAGA API layer, others, such as data/compute colocation are not. Our implementation is thus required to interleave the core logic with explicit instructions on where processes are to be scheduled when operating on specific parts of the data set [?]. Note that some of the required capabilities can be provided by higher level grid services – those are, however, often not standardized, and often not available in general purpose Grids.

The advantage of this approach is obviously that our implementation is no longer bound to run on a system providing the appropriate semantics originally required by MapReduce, but is portable to other, more generic systems as well. The drawback of the approach is obvious as well: our implementation is relatively more complex, as it needs to add semantic system capabilities at some level or the other, and it is inherently slower, as it is for these capabilities very difficult or near impossible to obtain system level performance on application level. But many of these are due to the early-stages of the implementation of SAGA, and not a fundamental limitation of the design or concept.

**Summary:** A data affine, and compute/data affine environment with the capabilities listed above provides a clear separation of concerns for MapReduce implementations.

## 3.2 Example 2: Replica Exchange

Replica Exchange [?] is a simulation method which improves the properties of Monte Carlo simulations: according to certain specific measure (here of temperature), two MC simulation instances exchange their configuration. That procedure improves the mixing properties of set of MC simulations.

In our specific application case [?], the implemented replica-exchange framework relied on the following system capabilities:

- efficient execution of many jobs
- fault tolerant job execution
- efficient exchange of small data items (MC temperature)

Again, our implementation targets general purpose Grids. Typical for Grids, as discussed in Section **??**, no production Grid system available to us provides that complete set of capabilities, but all provide a set of lower level capabilities, which can be used to implement the required ones.

We deployed an application level fault tolerance service, Migol [?], and interleaved checkpoint/restart instructions with the original application code, using the SAGA CPR package. Our implementation can thus operate on general purpose Grids, as it does not have strong requirements on system capabilities, but again the application development process is unnecessarily complicated, and definitely not focused on the core method implementation.

[1]Both applications have been implemented using SAGA [?], but we do not, in this paper, intent to focus on SAGA as a solution to the discussed problem space, but merely use it as means to an end.

## 4 Conclusions

We have argued above that affinities guide, as well as support, the implementation of high level programming abstractions. Clouds[2] currently however, offer a very finite list of affinities: They are either compute centric (e.g. EC2) data centric (e.g. S3) although some Clouds also offer data-compute colocation. There are no Clouds which are communication affine, and would thus invite users of tightly-coupled applications, or offer event-driven application execution, or pipeline execution modes, etc. If Clouds are to make a broader impact on Scientific Computing, this deficit will need to be addressed.

Trying to migrate applications to a cloud environment which does not offer the appropriate usage modes and affinities is very like a painful and tedious endeavor, and will likely show similar pain-points to the development of Grid applications. Also, performance and scalability may not be what the user expects.

On the other hand, a Cloud environment which does offer the suitable usage modes and affinities can significantly ease the work of the application developer, and likely yield the expected turnaround.

In many ways, that can be compared to the situation of application development for Grids environments: where the higher level grid services and abstraction layers are missing in Grid environments, application development and deployment is painful, with often limited returns. Clouds can inherently offer these higher abstraction layers – but not always the ones required.

For scientific computing, the road to successful Cloud deployment seems clear: depending on the target application classes, one needs to derive the required fundamental affinities the system needs to expose – which may, or may not, differ significantly from those offered by commercial Cloud providers.

## 5 Acknowledgments

This work is based upon the insight gained from many SAGA projects, especially the SAGA-MapReduce and SAGA-Replica Exchange implementation projects. We would like to thank Chris and Michael Miceli, Andre Luckow, Joohyun Kim, and the wider SAGA team at CCT.

## References

[1] J. Dean and S. Ghernawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[2] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[3] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Andre Merzky, John Shalf, and Christopher Smith. A Simple API for Grid Applications (SAGA). Grid Forum Document GFD.90, 2007. Open Grid Forum.

[4] Google Summer of Code Project. Project title: "A Map Reduce Framework Using SAGA".
`http://code.google.com/soc/2008/omii/about.html`.

[5] Shantenu Jha, Andre Merzky, and Geffrey Fox. Using Clouds to Provide Grids with Higher-Levels of Abstraction and Explicit Support for Usage Modes. accepted for publication in Concurrency and Computing Practice and Experience, preprint available at
`http://www.ogf.org/OGF_Special_Issue/` .

[6] A. Luckow, S. Jha, J. Kim A. Merzky, and B. Schnor. Distributed Reliable Replica Simulations using SAGA and Migol. In *Submitted to IEEE e-Science 2008*, Edinburgh, UK, 2008.

[7] A. Luckow and B. Schnor. Migol: A fault-tolerant service framework for mpi applications in the grid. In *Euro PVM/MPI 2005*, volume 3666/2005, pages 258–267. Springer, October 2005.

[8] Shantenu Jha et al. Programming Abstractions for Large-scale Distributed Applications. to be submitted to ACM Computing Surveys; draft available at
`http://www.cct.lsu.edu/~sjha/publications/dpa_surveypaper.pdf`.

[9] R.H. Swendsen and J.S. Wang. Replica Monte Carlo Simulation of Spin-Glasses. *Physical Review Letters*, 57(21):2607–2609, 1986.

[2] `http://cloudcomputing.qrimp.com/`