BSC vision on Big Data and extreme scale computing

Jesus Labarta, Eduard Ayguade, , Fabrizio Gagliardi, Rosa M. Badia, Toni Cortes, Jordi Torres, Adrian Cristal, Osman Unsal, David Carrera, Yolanda Becerra, Enric Tejedor and Mateo Valero

BSC

Science needs in Big Data at BSC

Solving new challenges in most application areas requires handling and processing huge amounts of data. Most computational groups at the Barcelona Supercomputing Centers already met this demands, with applications for molecular dynamics, genomic and protein docking analysis, air quality control, data assimilation, oil exploration, physiological human simulation, social graph analytics, smart cities and HPC performance analysis and modeling. We are also cooperating in projects in the human brain simulation and neuroscience analytics with equivalent demands,

An interesting observation is that each of these research teams has faced the issues from an individual perspective, experimenting and/or elaborating different solutions for their specific applications. We believe that this situation within our institution is probably representative of the current ad-hoc and fragmented approaches in the field, calling for a unification of efforts.

In this direction, BSC is committed in an internal effort (the Severo Ochoa project) to integrate the activities of application development departments (Life Sciences, Earth Sciences and Engineering) with the Computer Sciences department working on system software and architecture. The objective of this co-design project between application, programming models, runtime systems and architecture teams is to identify and/or develop an infrastructure to provide a unified, productive, easy to use and efficient environment for our broad range of applications.

Strategic considerations

Big data means big issues. Topics like applications, algorithms, system architecture and capacity/bandwidth balance at the different levels, scalability, middleware and APIs, security, resilience or provenance to name a few are extremely relevant to the area.

Among them, we would like to stress: the importance of algorithmic optimization and tuning, the increasingly dynamic usage patterns of the infrastructures; the importance of providing clean programming interfaces integrating the concurrency and data management models; the responsibility of the runtime in optimizing the mapping of computation and data to available resources; and the need to develop architectural support for the runtime and application functionalities. The following subsections further elaborate these topics, exposing the considerations that drive/inspire our efforts to develop an appropriate systems and middleware to efficient and productively support the needs of big data applications.

a) Reducing computational and data movement complexities

Algorithmic developments are extremely important. We believe that brute force algorithms are too often applied relying on the capability of scalable hardware to solve a problem. Better algorithms that drastically reduce the **computational complexity** to solve a problem have always been the best ways to reduce time and energy to solution. With the advent of the memory wall problem, the high energy cost and long latencies of data movement, the reduction of the **data movement complexity** needs to be considered in combination with the computational complexity. The introduction of **asynchrony**, avoidance of global synchronization, allowing for **non-structured patterns of concurrency** and large look-ahead capabilities are features that will allow our algorithms to tolerate the latency limitations of the infrastructures.

Today's algorithms are frequently used in very simple linear workflows that often correspond to an analysis process or methodology. We envisage that programs with more flexible control flow structures will be of great relevance and will bring increased intelligence to the coarse-grain **computational workflows**.

b) Usage models and resource management

In the current batch-oriented dominant practice users often launch a lot of computations that are analyzed (and not always) offline. We anticipate that the usage of big data and compute infrastructures will need to evolve to allow higher dynamicity and responsiveness. The incoming rates for streamed data sets may significantly vary along time depending on external data acquisition and measured system conditions. Interactive and steered usage will dynamically shift the focus and balance of computation and data processing, potentially deciding that computations initially scheduled are no longer relevant and unforeseen analyses or visualizations become suddenly important. Flexible and intelligent computational workflows will also present variable computational and data access demands as they proceed.

These usage patterns do generate strong requirements on the dynamicity of the resource management of the infrastructure. Sharing a large infrastructure by several applications and users has the potential to improve user satisfaction and save a lot of resources if appropriate resource management policies can be implemented.

Mechanisms supporting **malleability** have to be developed/introduced to make applications capable of adapting their internal parallel and data access structure to changes in the allocated resources (data, communication and compute) as they appear. **Dynamic resource management** policies will have to match the demand and availability, redistributing resources to maximize the efficiency in the use of the overall infrastructure and maximizing the quality of service experienced by users.

c) Programming models

Ideally, programming models should provide an interface by which the application developer expresses algorithms and ideas in a platform-agnostic way. The interface should also allow conveying information/hints to the runtime that could be dynamically used to optimize the actual mapping of computations and data access/movements to resources. In current Big Data practices, concurrency and data processing are often considered as independent aspects, with most of the efforts focusing on one or the other. Furthermore, they are offered through fairly large APIs, each of them introducing many concepts and essentially targeting a specific level of granularity. As a result, large programs with many lines of code dealing with platform specificities rather than problem related logic are written. The specialization of developers at a given granularity level can lead to applications that are not optimized at all levels from global orchestration to low level data access or computational kernels.

We believe that it is important to propose models that **better integrate the concurrency and data processing** aspects trying to rely on the same type of abstractions at the different granularity levels. Particularly relevant is the integration of flexible parallel control flow structures and query mechanisms to refer to huge data sets. Support for flexible parallelization strategies (asynchrony, nesting) is needed to free applications from latency limitations, converting them into throughput based applications where amount of resources (bandwidth, cores, $) is the limiting factor. By using **persistent object-based storage** layers we have the potential of closely mapping the data models in the programming languages to those used to provide data durability and also to provide a flexible shared communication space between partially coupled or independent applications. Such integration would simplify programs, eliminating the programmer need to consider the different models and a lot of the code that today is devoted to explicit I/O operations.

We consider that two features of our designs will be extremely useful to ensure productivity, maintainability, application integration and portability: first, being able to leverage current programming languages at the different granularities (sequential or parallel programming language, scripting languages, or DSL) with minimal extensions; second, using similar underlying concepts (ie. task based models) at the different granularity levels (workflow, parallel program).

The models must allow and encourage a holistic optimization considering all levels of granularity. This should even include potential architecture support with specialized functional units for computational/data movement tasks and potentially hardware support for the runtime. The models should integrate clean and simple mechanisms to enable intelligent data layouts and locality optimizations, allowing the runtime to minimize data movements and/or move computations close to the data.

d) Intelligent runtimes

The programming models provide the interface for the programmer to describe its requirements in terms of computations, data accesses and dependences or communications. It is then up to the runtimes to optimize the mapping of those requirements to the available resources when the application is run on a given architecture. Different implementations of the runtimes may vary in overhead, latency or bandwidth, targeting different platforms (from very distributed to fine-grain parallel) and thus supporting different granularities. Our philosophy is that runtimes should decide actual data placements, replication, transfers (whether data to computation, computation to data or intermediate). As stated in the previous section, the layer above can and should help by providing hints to the runtime but it is important that this interface is defined in as abstract terms as possible, resulting in conveying useful information without requiring the programmer or user to control the actual details of the architecture. This is certainly an area where concurrency and computation have to be integrally handled and where a lot of intelligence has to be developed and embedded in the system.

e) Architectural support

Finally, research in architecture and hardware support has a potential huge pay off. This applies both to processor design and storage system architectures. Of special interest is to propose processor optimizations for certain functionalities relevant for application algorithms and programming model support. A particular challenge is to holistically study the dimensioning of capacity and bandwidths of the different memory/communication/storage layers in the sight of new technologies, the locality and asynchronous data movement potential optimizations by the runtime and the algorithm characteristics.

BSC technologies

The BSC effort to develop a middleware to support the different internal application projects enumerated in the introduction builds around a **shared persistent object data management layer** cleanly integrated in the generic **StarSs programming model**. We also explore different architectural features that will provide an efficient support for both the data and computation aspects of these applications.

In the data management area, different teams at BSC have been working in the past on aspects relevant to Big Data: resource management in MapReduce focusing on task scheduling driven by actual resource consumption and high-level performance metrics; resource-aware task scheduling with completion time goals; decoupling user interfaces from data layout in non-relational databases (using Apache Cassandra as a representative of non-relational database); usage of specialized hardware architectures as IBM’s BGAS platform for running some of BSC’s genomics workflows; and integration of its programming models with the BGAS key/value store.

Within the Severo Ochoa Project we are integrating the research experience of those groups onto a unified data access and management layer. Our global objective is to devise a persistent object storage interface that can be mapped on top of different underlying data management infrastructures and cleanly integrated with the programming model developments described in the following paragraphs.

At the programming model level, the StarSs concept is that the programmer specifies tasks and the directionality of the data accesses they perform. The sequential execution of a control flow instantiates the tasks and their dependences are computed by the runtime based on the directionality annotations. Key philosophical considerations are malleability, support for automatic locality management based on the data access annotations that are used to build dependences, and resource independence (specification of tasks and not threads, processes or other abstractions). The concept is implemented in COMPSs at the medium/coarse grain (tens of milliseconds and up) level and in OmpSs at the fine/medium grain (few microseconds to tenths of milliseconds or seconds) parallel programming level. We think that their hierarchical combination provides a unified conceptual framework and a simple to use interface leading to programs focusing on the actual scientific problem they target. The models allow for very dynamic mapping of computations and data accesses to the available resources by intelligent runtimes and data management layers that we provide.

COMPSs aim is to support flexible computational workflows. Several language bindings are available: C, C++ and Java for the original implementation and PyCOMPSs, a recent development that introduces efficient parallel support in Python. The annotations of argument directionalities are provided through method interfaces in Java and through decorators in Python. In its original implementation, the COMPSs runtime computed dependences using the files read or written by the application, targeting only coarse granularities. In the current version, the arguments to the tasks can also be objects declared in the language type model and dependences are also computed based on accesses to such local objects. The execution engine actually offloads the objects and computations to different cores or nodes, supporting the efficient parallel execution of medium granularity programs. Tasks from the same computational workflow can actually be offloaded to different nodes within the local cluster or to external cloud resources in a transparent way.

PyCOMPSs offers an elegant solution to automatically parallelize/distribute applications in the widely used Python language. By cleanly integrating the persistent object model in it we expect to offer to application programmers the possibility to incrementally enable Big Data requirements in existing applications. The environment will also ease and encourage the productive programming of more dynamic workflows.

At the finer granularity level, OmpSs offers C, C++ and FORTRAN bindings. Its design follows the same philosophical considerations as COMPSs: asynchrony (through the dynamic building of the dependences and dataflow-based execution), nesting and heterogeneity (accelerators). Major features of the dependence specification mechanism proposed in OmpSs have been adopted in the latest specification of the OpenMP standard (version 4.0).

Our efforts try to integrate the philosophical guidelines presented in the previous section in the design and implementation of the different components of our environment. We expect that the cooperation within the Severo Ochoa project of scientific teams in the different areas working on the algorithmic aspects and building on top of the middleware and architectural developments proposed will lead to significant demonstrations of real social impact.