1

WHAT IS GRID COMPUTING?

Grid computing has emerged as a framework for supporting complex compilations over large data sets. In general, grids enable the efficient sharing and management of computing resources for the purpose of performing large complex tasks. In particular, grids have been defined as anything from batch schedulers to peer-to-peer (P2P) platforms.

Grid computing has evolved in the scientific and defense communities since the early 1990s. As with most maturing technologies, there is debate as to exactly what grid computing is. Some make a very clear distinction between cluster computing and grid computing. *Compute clusters* are defined as a dedicated group of machines (whether they are individual machines or racks of blades) that are dedicated for a specific purpose. Grid computing uses a process known as "cycle stealing": grabbing spare compute cycles on machines across a network, when available, to get a task done.

Since both compute clusters and grids coordinate their respective resources to perform tasks, when does a compute cluster start to become a grid? Specifically, does a compute cluster become a grid when it is leveraged to perform operations other than those for which it was originally intended?

THE BASICS OF GRID COMPUTING

Grid computing is an overloaded term. Depending on whom you talk to, it takes on different meanings. Some terms may better fit your practical usage of the

Distributed Data Management for Grid Computing, by Michael Di Stefano

technology, such as clusters. For the purposes of this discussion, however, we shall define grid computing as follows:

> Grid computing is any distributed cluster of compute resources that provides an environment for the sharing and managing of the resource for the distribution of tasks based on configurable service-level policies.

A grid fundamentally consists of two distinct parts, compute and data:

- *Compute grid*—provides the core resource and task management services for grid computing: sharing, management, and distribution of tasks based on configurable service-level policies
- *Data grid*—provides the data management features to enable data access, synchronization, and distribution of a grid

If the proliferation of jargon is a measure of a technology's viability and its promise to answer key issues that businesses are facing, then transformation of jargon to standards is a measure of the longevity of the technology in its ability to answer concretely those key business issues. The evolution of *grid computing* from jargon to standard can be measured by a number of converging influences: history, business dynamics, technology evolution, and external environmental pressures.

The drivers behind grid technology are remarkably similar to those that corporations are facing today: a starving business need for powerful, inexpensive, and flexible compute power, and limited funds to supply it. In the early 1990s, research facilities and universities used increasingly complex computational programs requiring the processing power of a supercomputer without the budget to supply it. Their answer was to create a compute environment that could leverage any spare compute cycles on campus to perform the required calculations.

Today, grid technology has evolved to the point where it is no longer a theory but a proven practice. It represents a viable direction for corporations to explore grid computing as an answer to their business needs within tight financial constraints.

There are additional forces in play that will present a fundamental paradigm shift in how computing is done. As it migrates from the hands of artistry to the realm of engineering—via the application of tried-and-true engineering principles—computing becomes a fundamental utility in the same way that gas and electricity generation and delivery is a utility. The quality of the service will be measured by its ability to meet the supply-and-demand curves of the producers and consumers.

Leveling the Playing Field of Buzzword Mania

There are many analogies in the development and adoption of grid computing to those of client/server technology. Both are fundamental paradigm shifts in the way computing is performed. As client/server technology ushered in the broad acceptance of relational database technology, grid technology will usher in new

data management paradigms to address the specific topology of the physical compute grid.

To see how this is happening, it is best to untangle the concepts of data management in grid form by drawing on a fundamental baseline that we are all familiar with. The people who are going to use grid technology—developers, architects, and lines of businesses—are accustomed to thinking in terms of client/server technology and the relational data management features within a client/server paradigm. Irrespective of the compute topology—client/server, computer clusters, or a computer grid—from the user perspective, these data management service levels need to be consistently maintained.

In the early days of client/server technology one would attend a seminar sponsored by a relational database vendor, promoting relational technology in general, and the supplier's product in particular. The message was that the new compute paradigm of the client/server topology required new, more flexible data management techniques than do those currently in use. As a result, relational databases became synonymous with client/server technology and the standard for data management.

People attending those seminars were used to writing their own disk controllers for data storage, so popular questions centered on disk management. How fast does your product write to and/or read from disk? How efficient are your indices? How well does your product manage physical data positioning on the disk? The bulk of the seminar was spent on addressing these questions, and the only discussion of data management centered on the use of a new language called *Structured Query Language* (SQL) for storage and querying of the data. If you were interested, there were SQL training classes to attend, where only the basics of how to form a query were taught.

Figure 1.1 illustrates the parallels of the vocabulary and fundamentals between data management within relational databases and that within grid computing. This comparison is useful in two aspects: (1) it relates to terms that most are already very familiar with and (2) more importantly, it suggests that any data management system in grid computing must provide the same levels of service quality as within relational databases.

Figure 1.1 links a baseline of data grid vocabulary to well-known relational database terms. Relational database implementations have two fundamental components: (1) the underlying engine that manages physical resources, in this case a disk and (2) a layer on top of that to provide all the data management features and functionality that architects and developers would rely on for data management, querying, arrangement of data in highly ordered structures such as tables, the ability to transact on data, leveraging stored procedures, event triggerings, and transacting in and out of the database with external systems. These are the management features and functions that today are where our true interest lies. How do I manage tables/row locking? How do I structure indices for maximum performance? Very little attention today is given to the underlying engine.

In the same way that *relational database* is a generic term, so is *data grid*. Companies will offer implementations, products of their vision of what a data grid is. To analyze the differences between the products offered, it is possible to apply a

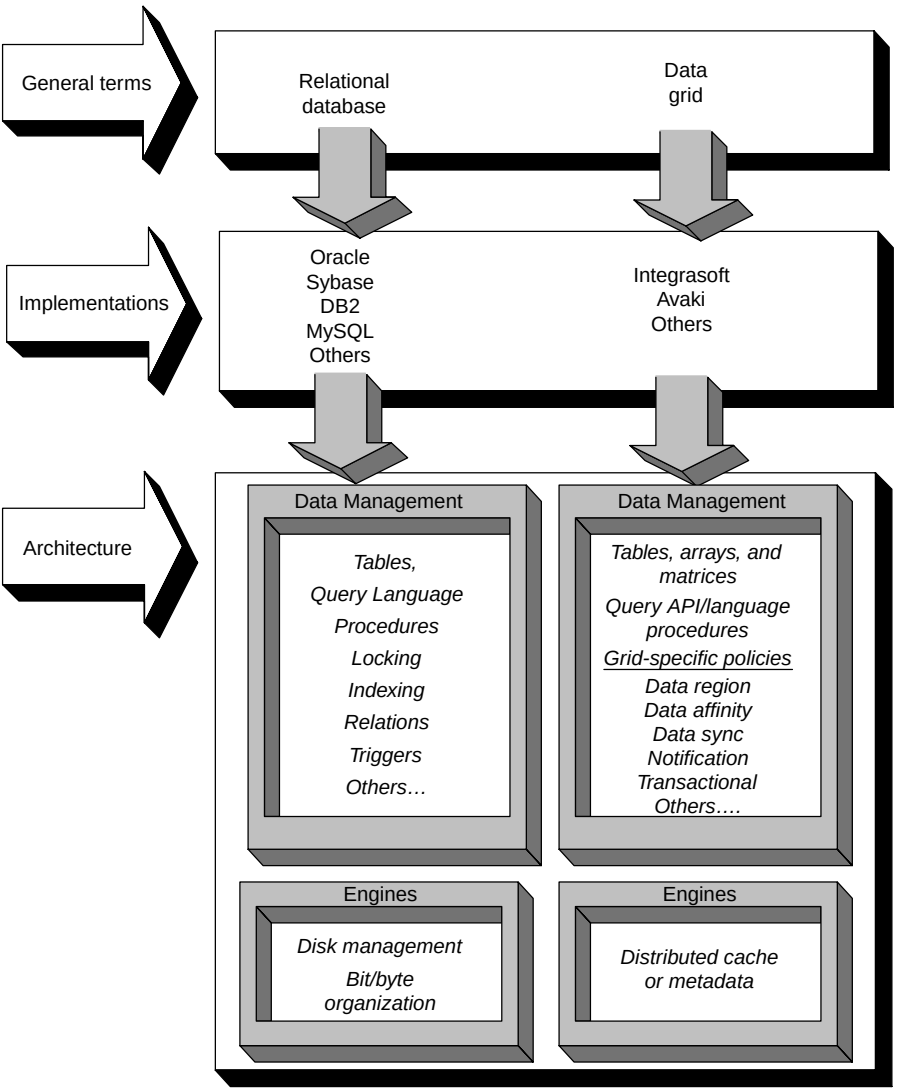

Figure 1.1. Baseline of terms and function.

baseline consisting of generic term, implementation, data management, and engine. Each implementation of a data grid will have an engine. That engine may be a metadata dictionary or a distributed cache. It will also handle the data management aspects of this data grid, defining how to structure data in tables, arrays, or matrices; how to query data; and how to transact on the data.

Depending on the exact implementation of this engine—whether it is a metadata dictionary that routes requests to the true long-term persistent stores, or a distributed cache that spans all computers in the grid to form one virtual space—there are

specific data management issues for this new topology. How to synchronize, how to transact on the data, how to address data affinity? These are all data management issues; issues that, no matter who the architect or application developer is, will need to be addressed within their applications. These are the quality-of-service (QoS) levels that are required of the data grid. If a data grid does not provide such service, then developers will have to write down to the lowest, most fundamental level of bit and byte management.

Data grid support for true data management extends to facilitation of the adoption and widescale acceptance of grid technology. Developers can easily transit from client/server-based applications to a grid topology by leveraging a product that provides the same levels of service quality that have become the standard with relational databases.

PARADIGM SHIFT

The technology concepts behind grids had their origins in distributed computing networks based on Distributed Computing Environment (DCE) and Common Object Reguest Broker Architecture (CORBA). The approach and value proposition, however, are radically different.

DCE- and CORBA-based distributed computing applications sought to separate client and server, and to move processing off to a server or set of servers, thereby reducing the requirement for large clients. Grids seek to harness large blocks of processors into a virtual pool. Once virtualized, these pools are managed by the grid, which provides a standard set of services that address

- Security
- Data management
- Discovery
- Reliability

Heterogeneity is key, and these pools range from desktop PCs for the purpose of AIDS and cancer research, to large servers for problems in computational physics and biology.

Beyond the Client/Server

Traditional client/server applications are typically configured as a client process connecting to a utility server such as a database. The client/server architecture can be further refined as to what a server is and what a client is. Clients that process the business logic ("fat" clients) can become "thin" clients by moving business logic processing to a separate server process, sometimes called an *application server*. The application servers would then in turn connect to the utility server (i.e., a database), thus forming a chain: clients connecting to an application server connecting to databases (see Figure 1.2).

Thus, client/server topology fundamentally is a piping of clients and applications. Operationally, for each line of business application, this implies a strict discipline of dedicated machines running the respective application and database servers. When planning the capacity of a data center, the rule of thumb is that the server capacity is twice that required at peak load. However, the peak load may occur only a few times a day for short intervals. Thus, for most of the time the machines are running far below their capacity (typically less than 30%). This leaves vast amounts of wasted compute capacity.

The use of distributed middleware products—such as a messaging—transforms the client/server piping topology into a "message bus" topology. Servers can now handle "requests" via the middleware messaging bus. Clients issue requests to the middleware, which routes the message to the appropriate the service. This is the beginning of a distributed processing environment, the decoupling of the physical resource to logical service. However, the capacity planning of the data centers follows the same rules as does the client/server topology, thus doing little to harness the vast, untapped compute capacity of the servers.

Grid computing is a further evolution of distributed computing that attempts to better utilize unused compute capacity. It enables the freedom to choose the hardware that is best suited to run the service at a specific point in time. This offers a better utilization of the physical resource. For example, machine A in a client/server topology was dedicated to one service. That same machine in a grid

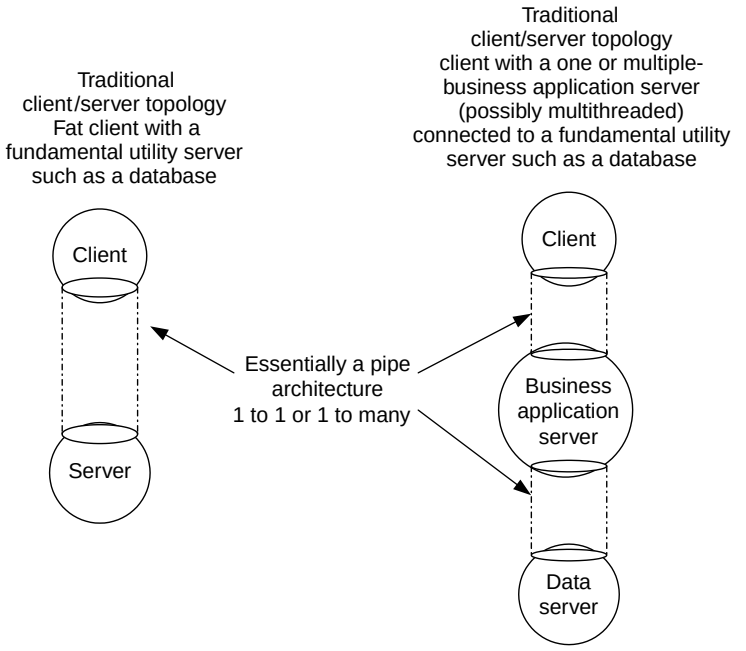

Figure 1.2. Traditional client/server topology.

topology can now support any service, with the limitation matching the machine's hardware/software provisioning to what is necessary to run a specific service.

Within a client/server environment, threading of servers allows for similar request processing—one thread for one request—thus allowing a single-server process to handle multiple clients at the same time. However, there is an upper limit to the practical number of threads that can efficiently run in that single process. Within grid technology, there is a similar concept. What would run in a thread can now be run on the best available machine in the grid. The end result is the elimination of any upper bound that exists in a single-machine, multithreaded process.

In a grid, a service can be further subdivided into tasks or worklets. The tasks can now be "sprayed" across the entire grid, thus transforming a sequential process into an *n*-way parallelizable event. What was a long-running process can now be completed in a fraction of the time.

As more capacity is needed to support the business, more hardware can be added to the grid. Once a service is grid-enabled, there are no programming changes necessary to take advantage of the additional capacity. This sets up the scenario of an infinitely wide grid, with "worklets" simultaneously accessing resources such as a database. What was a piping of client to server now resembles a funnel of clients trying to reach a single resource: orders of magnitude more "clients" trying to access data from a resource not designed for this wide-mouth funnel of requests (see Figure 1.3).

In attempting to handle large numbers of client requests efficiently, software companies have split up the servers by sharing or "striping" the workload across

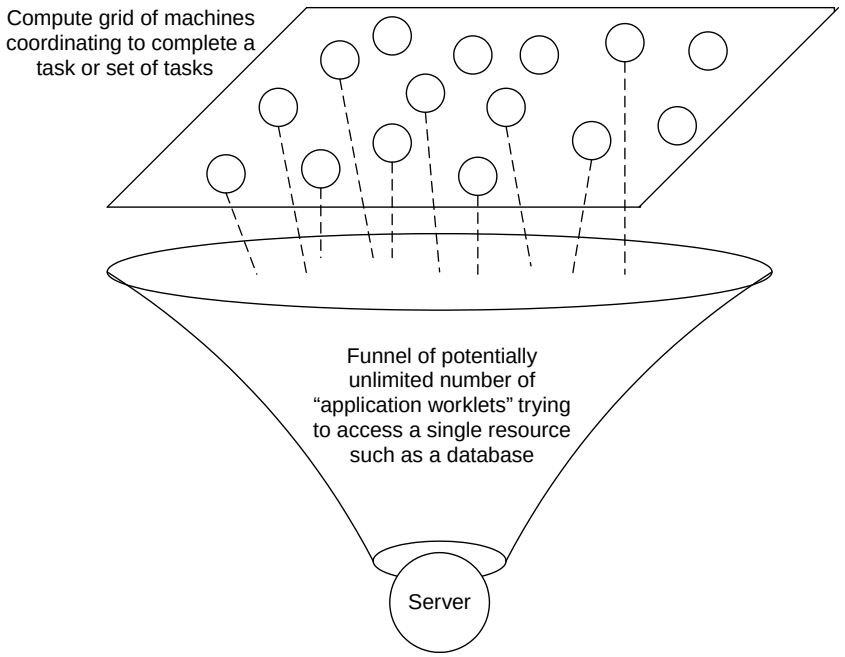

Figure 1.3. The grid funnel to data sources.

multiple server peers. This does increase the processing capacity of the servers behind the server wall but does not address the client request/response bottleneck. Attempting to use faster client/server technology in this way simply creates a processing hourglass (see Figure 1.4): wide client grid, and wide server process fanout with a bottleneck at client access to the server.

Data management in grid computing addresses the widening of the throat of the hourglass to the width of the grid to eliminate data access bottlenecks (see Figure 1.5).

NEW TOPOLOGY

Grid computing builds on established concepts of distributed computing to create a physical topology that is very different from that of the client/server. A computer becomes a network of smaller machines coordinating with one another to complete

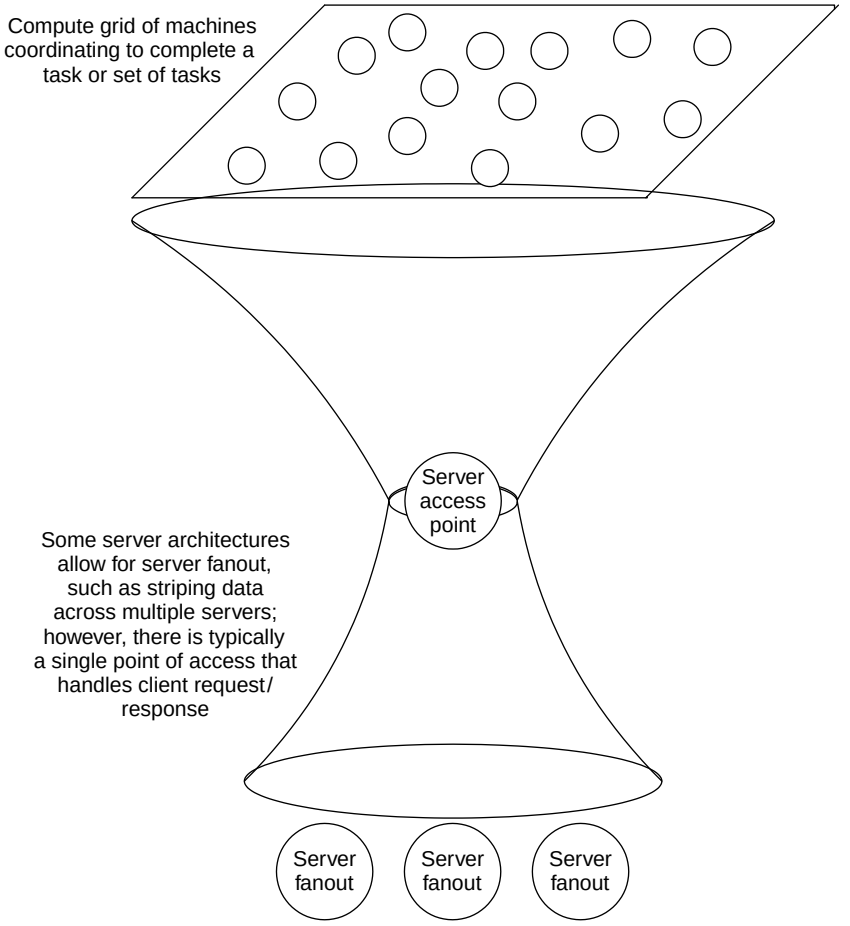

Figure 1.4. Grid and server hourglass.

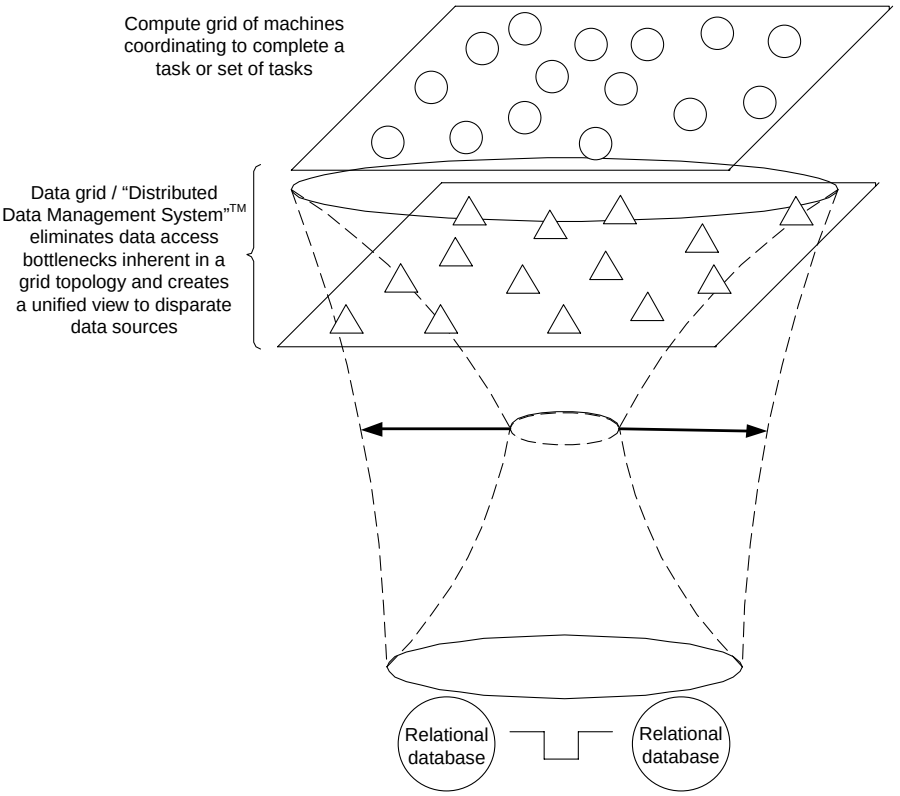

Figure 1.5. Distributed data management in grid eliminates data access bottlenecks.

a variety of tasks—a collection of reconfigurable nodes for performing a variety of different tasks without human intervention, in contrast to the siloed/specialized data centers of today:

- *Elasticity*—Information technology (IT) spending is being tied directly to business volume, forcing greater transparency and other benefits.
- *Pervasiveness*—There are a proliferation of uses of IT resources for basic needs much like a utility (electricity, telephone, etc.).
- *Defense spending*—IT spending is closely controlled by the upper management and corporate CIO/CFO.
- *Moore's law*—The cost of hardware is decreased.

Each of these forces has rippling effects throughout a grid architecture, thus forcing grid acceptance:

- *Elasticity*—increased emphasis on metering usage, and the utility concept within IT. For example, one utility must support multiple functions such as high-performance computing and Web Services.

- *Pervasiveness*—increased commoditization of basic functions [DNS (Domain Name System), Mail, Web, etc.].
- *Defense spending*—increased R&D in data integration, prediction, reliable infrastructures (à la ARPANET).
- *Moore's law*—increased emphasis on encoding more functions on chips themselves [i.e., Flash, PROM (programmable read-only memory), and RAM (random access memory) in everything, and nothing else].
- *Data management*—how to maintain the same "user experience" in data management and not hinder the realization of the full potential of the grid environment.