12

DATA AFFINITY

Up to this point, our discussion has centered on data regionalization, data synchronization, and data distribution. All are important and necessary data management features within the data grid. Combined, they provide the tools that enable the data grid, in particular and grid computing in general, to expand beyond simple computational problems to the broader category of the data-intensive applications typically found in private industry and government agencies.

But there is another important feature of a data grid that must be addressed: data affinity. This, in short, addresses the maximum usage of the most precious and costly resource within the grid—the network. Minimizing the movement of data across a network increases the efficiency and pliability of the grid on both processing and cost bases. Data regionalization, data distribution, and data synchronization each provides a necessary component that, when managed in synchronization, achieves the broader objective of data affinity.

There are breakeven points, points where it simply becomes too expensive to move data within a grid in comparison to the cost of performing the operation locally without using a grid. Efficient data management within a grid, or data affinity, can lower the breakeven point for a broader and more complex application set. The more applications the grid can support, the broader the acceptance and the more relevant the technology becomes—a spiraling cycle that feeds on itself, expanding the technology beyond critical mass; in essence, a paradigm shift in computing.

Distributed Data Management for Grid Computing, by Michael Di Stefano

A MEASURABLE QUANTITY

The cost-basis components of grid computing are processing power/capacity (e.g., CPU), disk storage, and network bandwidth. The first two are orders of magnitude more economical than the latter, the network. Network bandwidth is by far the most costly resource. It costs more to move data across a network than it does to store them on disk. For example, it costs more to move data across a network than to spend additional CPU cycles to regenerate the data over and over again.

A paper written by Jim Gray, on distributed computing economics,[23] categorizes some of these costs. Included in the costs of distributed computing are database access and disk storage; our discussions will not take these parameters into consideration. Long-term data storage is a constant, independent of the compute topology used in grid computing or client/server computing. In the situation where the data grid is file-based (e.g., GridFTP, distributed file system, or some other variant), the local node storage is disk, which is inexpensive in comparison to other resources. In the case where the data grid's memory is used for data storage (e.g., RAM), this, too, is fairly inexpensive. Therefore, the result is a drop in the ocean compared with the overall cost of operating an efficient grid architecture. The two variables we will examine are the costs of computation—that is, the associated cost of different grid nodes (e.g., tens, or hundreds, or thousands of computational nodes)—and the cost/efficiency ratio of the nodes in performing calculation-type tasks. The second resource that will drive the cost analysis is network bandwidth, the cost of moving data between the nodes of the grid to perform a task.

A useful metric is the ratio of computational cost versus the cost of data movement on the network. This parameter defines the point where it becomes too costly to perform an operation over a grid because of the required movement of data. A paper written by Hewlett-Packard[24] discusses the dynamics of distributed computing (see equation below). An important part of this is minimizing the overall network traffic by locating the tasks or services closest to each other. A quantitative analysis metric is proposed in the paper is the "partial objective function (POF)." POF is a way to measure the cost of moving data between nodes alongside the computational cost or processing capacity of the nodes

$$f_{\mathrm{POF}} = \frac{\beta}{\beta + (\alpha C_{\mathrm{T}} + (1 - \alpha) u_{\mathrm{T}})}$$

where C_{T} = sum of traffic costs between the services on a pair of servers weighted by the distance between these servers, u_{T} = variance of the processing capacity usage among the servers, α = balancing factor between 0 and 1, and β = chosen according to the maximum possible values of C_{T} and u_{T} in order to ensure a relatively uniform distributions of the POF values.

Therefore, in order for a data grid to be effective, it must be able to provide metered data, metrics (i.e., POF or something similar), and data management

policy controls (both manual and dynamic), and externalize data locality information to

- Allow the scheduling of a task or service to take into account data locality so as to move the execution of tasks to where the data reside in an effort to reduce network traffic.
- Enable the data grid to migrate or redistribute data to nodes where the task is most often performed, based on data movement patterns due to task execution.

What to Expect from Data Affinity

What improvement in performance can be expected by smartly routing task to data locality or by routing data to most probable task locality? Ian Foster conducted a study on just this topic,[25] where the task performance was measured both with and without an efficient data grid that takes into account data locality with task scheduling. The results with every aspect indicate that the performance of the grid increased. Metrics included the average response time per job, the average data transferred per job, the average idle time of processors, and the average response time. In each case, when data locality was factored in, the performance for each metric improved anywhere from one to two orders of magnitude.

HOW TO ACHIEVE DATA AFFINITY

In general there are two ways to achieve data affinity. The first uses the *compute grid*, where information is provided to it by the data grid so that tasks are routed to the data. The second uses the *data grid*, by observing data movement patterns, migrating (caching) data to those nodes where the tasks seem to be the most frequently computed. Individually, each technique will increase the data affinity levels. Optimally, combining the compute grid's routing task to data and the data grid's data migration techniques will yield higher data affinity levels, thus minimizing network traffic and increasing the performance of the grid.

Regionalization, Synchronization, Distribution, and Data Affinity

The objective of data affinity is to have the data and task collocated as closely as possible to eliminate network traffic and to increase performance by reducing latency. This can be achieved in two ways. The first method is proactive, whereby tasks are routed to the data by giving the compute grid's task scheduler the required information on data locality so that it can make smarter task routing decisions with regard to data location on the physical nodes of the compute grid. Keep in mind that this method may not always be 100% successful. This does not always mean that the node selected to perform the task is the best choice, since it may not have the data needed for the task already cached, thus forcing the data grid to move the data to that node.

The data grid manages the second method of data affinity. The following analogy highlights the role of the data grid as both predictive and reactive in nature and contrasts it to the task scheduling function of the compute grid. As the compute grid's task routing function is to the offensive unit of a football team, the data grid's data migration efforts are to the defensive unit of the opposing team. The defense, given all it knows about current situation of the game and history of the opposing team's offensive capability, will predict what the next play will be, and set up the appropriate defensive strategy. However, even before the ball is put into play, the defense has to react to the play as it unfolds. The same philosophy holds true for the data migration efforts of the data grid. The data grid is predictive; thus it anticipates the compute grid's task routing patterns and migrates data to the physical nodes ahead of it. The data grid is also reactive by making real-time adjustments to data migration as the "play unfolds." As the tasks are routed to the physical nodes, the data grid must react by routing the data to nodes where the data do not yet exist. The data grid can accomplish its predictive data migration objectives through the combinations of its data regionalization, data distribution, and data replication policies.

One interesting side note is the effect of the physical size of the data grid on the data migration efforts of the data grid. In the case where the grids are of small physical size, the data grid becomes less effective in its role of achieving data affinity, leaving the compute grid's task routing via data locality as the primary method. As the physical size of the grid increases, the effectiveness of data migration by the data grid increases its contribution to data affinity.

The following functions describe data affinity with regards to physical size of the grid. For grids of small physical size

$$DataAffinity = ComputeGrid(Task\text{-}to\text{-}Data)$$

As the physical size of the grid increases, the data grid's data migration efforts play an ever-increasing roll towards data affinity:

$$DataAffinity = ComputeGrid(Task\text{-}to\text{-}Data) + DataGrid(Data\text{-}Migration)$$

The data grid's management policies work together to achieve data affinity. The outermost container for data affinity is the data region, which is a logical partition of data within the data grid. The data region has a physical boundary within the grid, as discussed earlier. This boundary consists of the specific nodes that are allocated to a data region and contribute their physical resources to that data region. It is possible for any one physical node to be an active part of multiple data regions. If the compute grid is to route tasks to where the data are resident, it must be one of the nodes that physically support the data region and that contains the data necessary for the task to perform its operation. Within a data region's physical boundary, the individual data atoms are replicated. The data replication policy determines the exact replication pattern or the number of required copies. Each data atom, including all replicas, is distributed throughout the data region, based on the data distribution policy of the data region. The data synchronization policy determines how all the data atoms and all the replicas coordinate with each other within the data region.

Synchronization can be tightly bound, where a change in state of one data atom is transitionally reflected in all replicas, or loosely bound, where a change in state of one data atom is reflected in the replicas, but in a nontransactional manner. The data synchronization policy has increasing importance as the data region starts to span the following:

- Areas of varying network bandwidth; should the data region span across a wide-area network (WAN), then the coordination of the replicated data atoms distributed across the data region becomes necessary for data accuracy and performance of the system. These two aspects must be weighed against each other when setting the policies of synchronization and distribution within the data region.
- Nodes of the compute grid support applications or services of different, noncoexistent hardware and/or software configurations (applications that require different operating systems, libraries, or other software configurations that cannot be shared on a single machine). This forces the creation of subregions within a single data region of the data grid. Within each subregion, data affinity must be maintained. This is done through data synchronization between the subregions, and within each subregion separate data replication and distribution policies are also required. An alternative approach is to have separate data regions, each spanning a configuration set and leveraging interregion synchronization to keep the group of data regions cohesive and in a well-known steady state.

Through the combination of data regionalization, data replication policy, data distribution policy, and data synchronization policy, the data grid performs both proactive and reactive data migration methods to contribute to data affinity within the data grid.

Other considerations include *macro events*. From the macro level, the size and shape of a data region can contribute to a slow-moving data grid, and the distribution of data within the region can be equally slow to change. Macro events cause changes in the data region's size, shape, and data distribution. The macro events are usually peak and off-peak service loads that occur at various intervals, including daily, weekly, monthly, and yearly. However, there are external forces to the grid that affect data regions at the micro level that cause smaller changes to the region. Such events are hardware failures, the addition of new hardware to the grid/data region, and variations in service demand. Macro changes are predictable and can be planned for, while micro changes are not predictable and are harder to plan for. Thus, continual adjustments to data regionalization, distribution, replication, and synchronization policies must be made to maintain peak data affinity levels within the data region.

Data Distribution is Key to Data Affinity

Earlier, we discussed what the data grid could do to assist in achieving data affinity. Among the data management policies of synchronization, replication,

and distribution, the latter has the most impact. The data distribution policy determines on which nodes the data atoms will physically reside. Should the data grid via its data distribution policy estimate correctly the nodes of the grid where tasks are most often performed, the movement of data across the data grid will be minimized. This area of data management in the data grid is one that will receive a great deal of attention by computer scientists, mathematicians, and engineers alike going forward. Similarities can be drawn to the exotic derivatives sectors of the financial markets. Mathematical models are under constant flux to predict market conditions and volatility in the markets, and ultimately determine instrument pricing and risk exposure. In each area, certain assumptions are made. For example, to price an option, one has the choice of using the Black–Scholes, binomial (Cox–Ross), Adesi–Whaley, or a host of other models. The Black–Scholes model assumes that the price of the underlying instrument follows a lognormal distribution. The binomial model is based on the probability that the price of the underlying instrument has an equal probability of going up or down. The Adesi–Whaley model establishes a differential equation between the estimated and actual prices of the modeled instrument. Today, the area receiving the most attention is the prediction of market volatility, a key input parameter to all the pricing models mentioned.

Prediction of how to best distribute data in a data grid has the same characteristics as pricing an investment in the derivatives market. Assumptions will be made on many of the variable parameters of data distribution, and quantitative models will be derived on the basis of these assumptions. For example, one can assume complete randomness. Any task has an equal probability of being executed on any given node of the grid at any given time; therefore the data can be randomly distributed across the data grid. Or one may make the assumption that tasks will center around the physical "hot spots" in the grid but will dissipate or radiate outward like a bell curve, and therefore a data distribution bell curve with two, three, or four standard deviations will be required. An engineer—your author is one—will establish a feedback loop and dynamically adjust the physical location of each data atom, based on past data movement patterns that have been collected and analyzed. As you can see, the possibilities are bound only by our minds.

In Part IV of this book, we propose a hypothesis that data distribution patterns occur at two levels: namely, data atom distribution within a "data body," and a second distribution pattern of the "data bodies" themselves. A data body is a grouping of a single data type, such as market pricing data or a customer portfolio. Looking at the larger system of a business service or application, data bodies will exhibit natural forces of attraction toward each other within the space of the data grid. The data grid will in turn exert a resistive force on the two data bodies. When the resistive force equals the attractive force of the data bodies, an equilibrium distance is established. The point of equilibrium distance represents the point of minimal data movement within the data system (all the data bodies of the business service or application) of the data grid. This suggests a system model where data distribution describes data bodies in a fashion similar to those found for the most fundamental laws of nature and physics.

Regardless of the assumptions one makes and the resulting data distribution model, the objectives are the same. How best to predict data usage patterns within a data grid in order to minimize the data movement during the normal operation of a system within a grid? The better the prediction model, the faster an "equilibrium" or "steady state" can be reached for data distribution, thus resulting in the most efficient use of the precious resource of the grid, the network.

Data Affinity and Task Routing

The compute grid's task scheduling function for data is one tool for achieving data affinity. This is done by making the compute grid aware of the data locality so that it can be used as part of its formula for routing a task to a grid node. Armed with the knowledge of what data are required for what task and physically where in the grid those data are localized, the task scheduler can make a smarter decision on where to send the task for execution. The first choice is to eliminate network traffic by routing the task to where the data are already cached. If this is not possible, then a node with the network proximity closest to those where the data reside is selected, which again will minimize network traffic. The number of "hops" that the data must take from the node *A* where the data reside, to node *B* where the task is routed, will affect network traffic.

What is the task scheduler of a compute grid? The task scheduler is the logical unit of work in the overall work flow of the compute grid that maintains an active inventory of

- Task—work to be done by the compute grid
- Dependencies required to complete the task; some examples are
  - Operating system
  - Compiled libraries
  - Task/service type dependencies (e.g., Web server, queuing systems)
- Resources capable of executing the task
- Determining which capable resources are available at the point in time when the task is to be executed

According to this inventory, the scheduler matches the best available resource to execute each task and distributes the task to the compute node.

INTEGRATION OF COMPUTE AND DATA GRIDS

The data grid can monitor usage and data movement patterns within regions and adjust the data distribution policy in such a way as to minimize the movement of data within the region. For example, if the compute grid routinely routes tasks to a node where the data are not local, the data grid needs to move the required data to that node. If the data grid notices this pattern occurring often enough, the data

distribution policy should be adjusted so that the data reside locally on the node in question in order to limit movement of data. Thus, readjustment of the data distribution policy will increase performance by estimating future data locality needs. However, this becomes more of a reactive approach to solving the problem of data affinity. One way of being proactive is to give the compute grid's scheduling algorithms the additional information about data locality. Armed with this additional information, should the scheduling algorithm have a choice between two nodes to route a task—one node where the data are local and a second node where the data are not local—the smarter decision would be to route the data to the node where data are local. In this way, the compute grid, through its task scheduler with the knowledge of data locality, can increase the level of data affinity by scheduling tasks to data locality.

Recalling the diagrams of a compute grid environment consisting of parallel compute and data grid planes, interaction between the planes is nonexistent and they can run and operate independently of each other (see Figure 12.1).

However, we have just made a case for improved performance and broader application sets that a grid environment can support through data affinity. This will force sharing of information or an interaction between the data grid and the compute grid (see Figure 12.2). The type of information that needs to be shared is data locality from the data grid to the compute grid, which identifies the physical nodes where the data atoms are located.

Even though data affinity is not necessary for operating a grid environment, the overall benefits outweigh the extra effort required to establish a link between the compute and data grids. Currently, there is no standard interface between the compute and data grids, so in the absence of a standard, some of the minimum requirements of such an interface—all of which will require the cooperation of both the data grid and compute grid providers—are listed below:

- The compute grid will require an open interface to the task scheduler to which the data grid can publish.

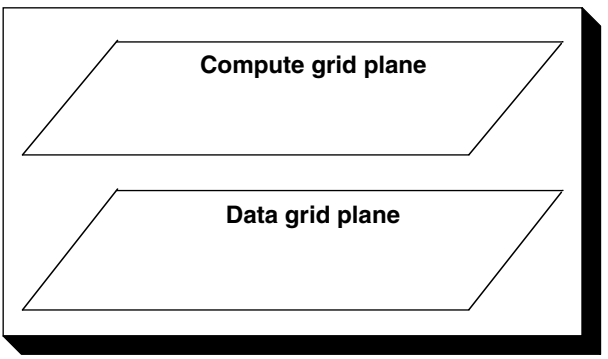

Figure 12.1. Parallel grid planes.

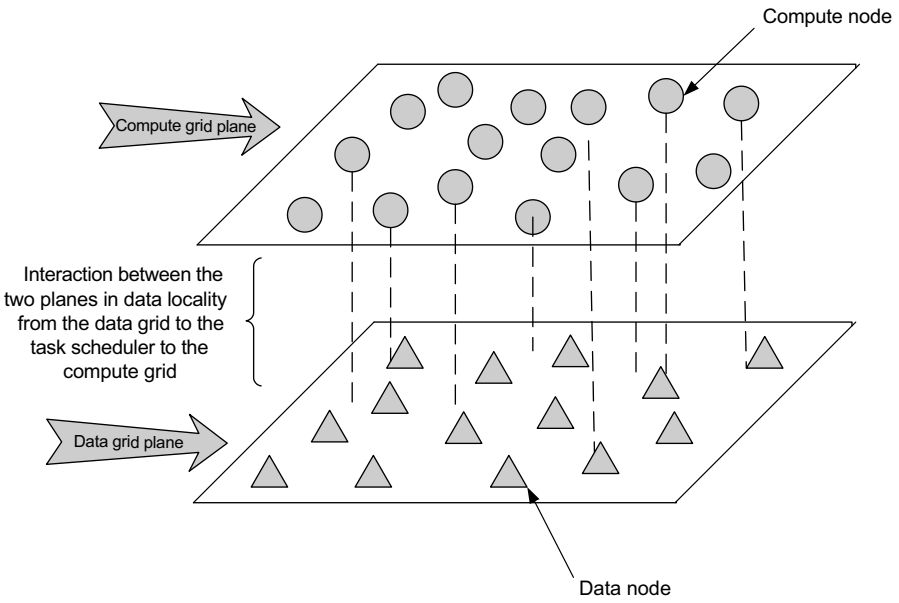

Figure 12.2. Interaction between compute and data grid planes.

- The data grid must provide a pull-based public interface, or a query capability so that the compute grid's task scheduler can query the location of data types, as well as and specific data atoms.
- (*Note*: This is an advanced method at the programmatic level.) At the application/task level integration (a programmatic API used to grid-enable an application), the data grid API can feed the specific data information required by the task via the compute grid public API, where the information is supplied into the compute grid, thus prepopulating the data locality requirements to the task scheduler.

EXAMPLES

Earlier, we discussed the separation of data management from the underlying engine of the data management system. Some implementations of the data grids can be metadata-dictionary-based, distributed-file-based, or distributed-cache-based. Each type of data grid is supported by its own unique engine. The separation of data management from the engine allows for the data management principles of regionalization, synchronization, distribution, and data affinity. The following are some examples of data grids that support data affinity:

- OceanStore: a project run at Berkeley, CA; distributes data (as files) across any number of servers in such a way as to promote data locality, robustness, and

fault tolerance. It analyzes usage patterns, network activity, and resource availability to proactively migrate data toward areas of use.

- A common query interface for individual data sources through the use of a shared metadata dictionary.
- Integrasoft's Grid Fabric, a data grid that establishes a federated cache space that spans the entire grid. Supports the distributed data management principles discussed in this book.