14

CALCULATION-INTENSIVE APPLICATIONS

DESCRIPTION

As I identified in an earlier chapter, many applications naturally lend themselves to the grid architecture. These applications are classified as being very computation-intensive and where the computational paths are data-dependent, they therefore can be run in parallel across many of the compute grid's nodes. The ability to run pieces of the computational process in parallel is where the first class of use case for grid computing comes into play. The use cases are responsible for defining the compute grid and the early stages of the data grid.

Today, many areas of research that use the grid technology, including

- Various engineering disciplines using computer-aided design[26]
- Genome research[27]
- Physics research: high-energy physics[28] and fusion research[29]
- Earthquake research[30]

The data sets for these applications are, for the most part, static in nature. For example, DNA sequences, seismic data, and collision patterns of high-energy beams in a particle beam accelerator, do not change once the data are recorded. In the commercial industry, an area of interest is risk management. There is one distinct difference between the risk management analysis and the abovementioned applications, which is that the data are far from static and tend to be dynamic.

The data continually change on a real-time basis (at varying time intervals: seconds, minutes) to more batch-type updates, which can be daily, monthly, quarterly, and so on.

Regardless of whether the application processes static and/or dynamic data, there is one commonality in all these applications: that all these application produce interim state data sets, during the analysis process where large temporary data sets are generated, used, and ultimately deleted. We can see that performance optimizations can be gained by smart data management of these interim data sets, independent of and in addition to any programmatic and procedural optimizations.

USE CASES

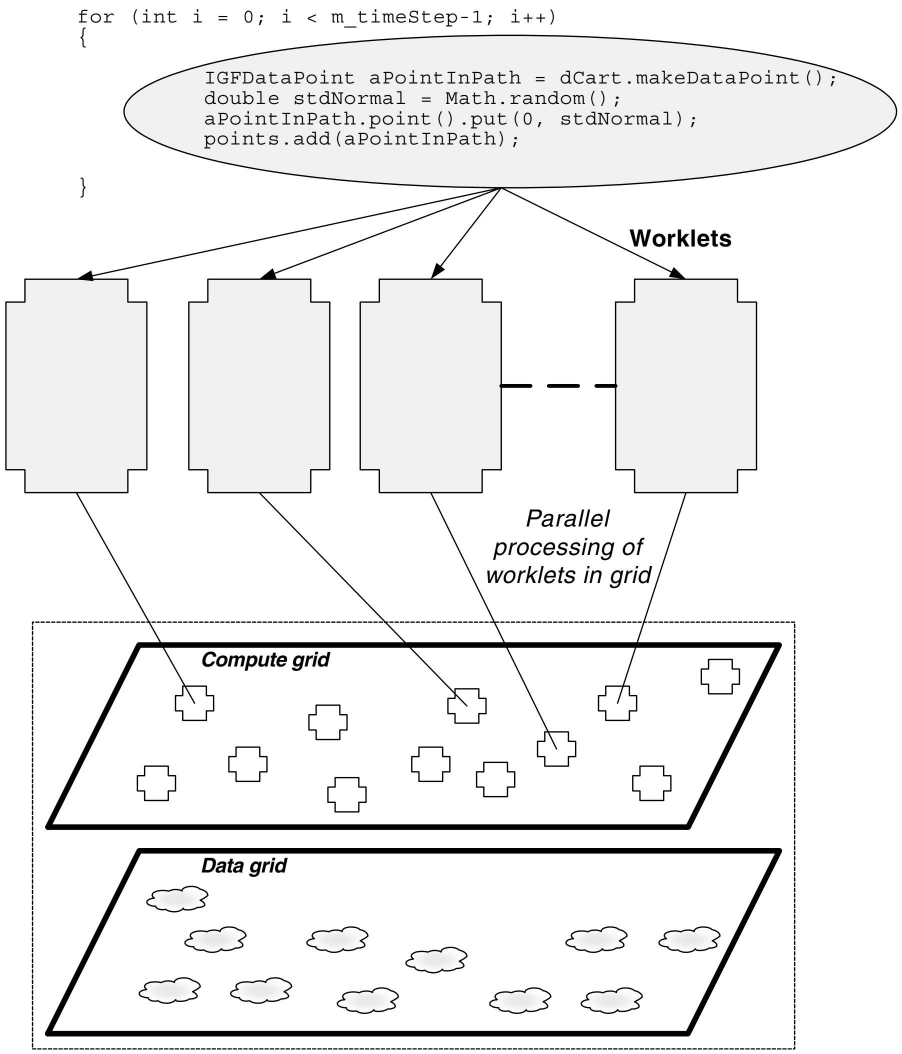
Calculation-intensive applications tend to naturally process the same algorithm repeatedly at varying iterations where only the input data set differs between all the iterations used to perform calculations. A large set of financial service applications fit this paradigm, including Monte Carlo simulations, binomial approximations, and Black–Scholes models, for example. Thus, parallel processing (parallelizable) can be easily achieved for these types of applications.

What does it mean for an application to be parallelizable? An example would be an identical unit of work that is typically run inside a processing loop and given a different data set to operate over for each iteration of the loop wherein each pass through the loop is independent of the ones that came before it; such a unit of work can be parallelized. Figure 14.1 illustrates how such applications can be parallelized through worklets running at the same time independently of each other in a grid compute environment.

We will start out be identifying each unit of work as a "worklet." Worklets can be assigned to execute on different machines in the compute grid depending on capacity levels. Applications that consist of these worklets are parallelized applications.

Worklets executed in parallel in the compute grid take advantage of the inherent nature of the compute grid, where a large numbers of machines are available to execute computational tasks. Worklets, via the compute grid, are assigned to the numerous compute nodes of the grid to be run in parallel, transforming a serial process into a parallel one and therefore reducing the overall time of execution and leveraging ideal resources. This feature is becoming increasingly essential in today's business. The window of operations continues to decrease, therefore mandating more and more near-real-time results as soon as the data are available and with the lowest latency possible.

In the early 1990s, universities and other areas of research found themselves with large complex computational analysis problems to solve without a supercomputer on which to run these computations. Necessity is the mother of invention; then came grid compute, since the objective was to make the most of what was available. All systems, including networked computers in labs, student unions, libraries, and classrooms, were utilized to perform such tasks. The hardware resources that were used in such research environments can be classified as represented in the



```
for (int i = 0; i < m_timeStep-1; i++)
{
        IGFDataPoint aPointInPath = dCart.makeDataPoint();
        double stdNormal = Math.random();
        aPointInPath.point().put(0, stdNormal);
        points.add(aPointInPath);

}
```

Figure 14.1. Parallelizable processes.

following formula:

$$\mathit{TotalComputationalResource} \cong \sum_{1}^{\mathit{computers-on-campus}} \mathit{NetworkedComputerComputationalResource}$$

The total hardware resources were a function of all the available resources in the whole facility. With this, they were able to divide the overall problem into atomic work units or worklets and parallelize the workload across all the available and capable machines on the network. Glue together the results into a cohesive data set, and the effect is a high-powered computer or the compute grid.

The need to perform more and more complicated, data-intensive tasks and limited resources has been the driving force behind the evolution of grid technology. Very similar forces are also found in the commercial enterprise and are causing widespread interest in further developing grid computing. The first commercial grid applications are similar in nature to those that gave birth to grid computing, where intensive analytical applications needed to perform with greater speed and the number of data sets that they required continued to increase. The applications spanned many industries, as indicated below:

- *Energy exploration*—where and how to best drill for oil and gas through the analysis of seismic data
- *Biopharmaceutical*
  - *Protein folding*
  - *Clinical trials*—drug interactions with the human body through simulation
- *Government*—various types of government applications for the analysis of large data sets and pattern detection
- *Financial services*—quantitative analysis to accelerate risk reporting
- *Computer-aided design*—aerospace, chip design, and other applications

The common thread connecting these diverse industries is that the business applications require complex data analysis over increasingly larger data sets. The analytical process that traverses and mines these data sets can require execution times spanning days. Time is our most precious and irreplaceable commodity, so any reasonable effort to maximize its utilization is well spent. Shortening the analysis times yields more available time and resources to perform increasingly complex analysis. Grid technology offers a reasonable solution to not only maximum utilization of time but also to provide a powerful, flexible, fungible, and cost-effective computing environment.

GENERAL ARCHITECTURE

The general architecture for this class of applications in the compute grid has focused on the management of computer resources and task distribution, which is the compute grid, and less on data management or the data grid. Historically, grid vendors have been perfecting and commercializing the core of the compute grid technology. The compute grid is used to manage compute resources—which machines are currently available and of these, which are capable of executing the worklet; the management of task assignment, data assignment, and retrieval for each worklet; and finally the assembly of the individual worklet's result data set into the larger application result data set format. The final job of the compute grid is to perform data packaging from each worklet since each has been assignment to a different compute node. Figure 14.2 illustrates this data assembling processes and final storage of the data to some data store, such as a database or a file on disk.

In the workflow shown in Figure 14.2, the compute grid manages data retrieval and distribution along with the worklets across the compute grid. This is a six-step process starting with data retrieval required by each compute node; this main process retrieves the necessary data from the various external data sources either before or during the worklet creation/assignment process of the compute grid. The compute grid management process then packages the input data necessary for the worklet to perform its task and ships it along with the worklet to the respective compute node to perform the task. However, before shipping the data–worklet combination, the data must be packaged. Part of this packaging is data marshaling. Once the compute node receives the worklet, it must unpackage the input data before it can start to process the tasks as defined in the worklet. The unpackaging process involves reversing the data marshaling process to enable the compute node to read the data in a format that it understands and can operate on. Once the task is complete, the resulting data sets must be packaged (and marshaled) before being sent

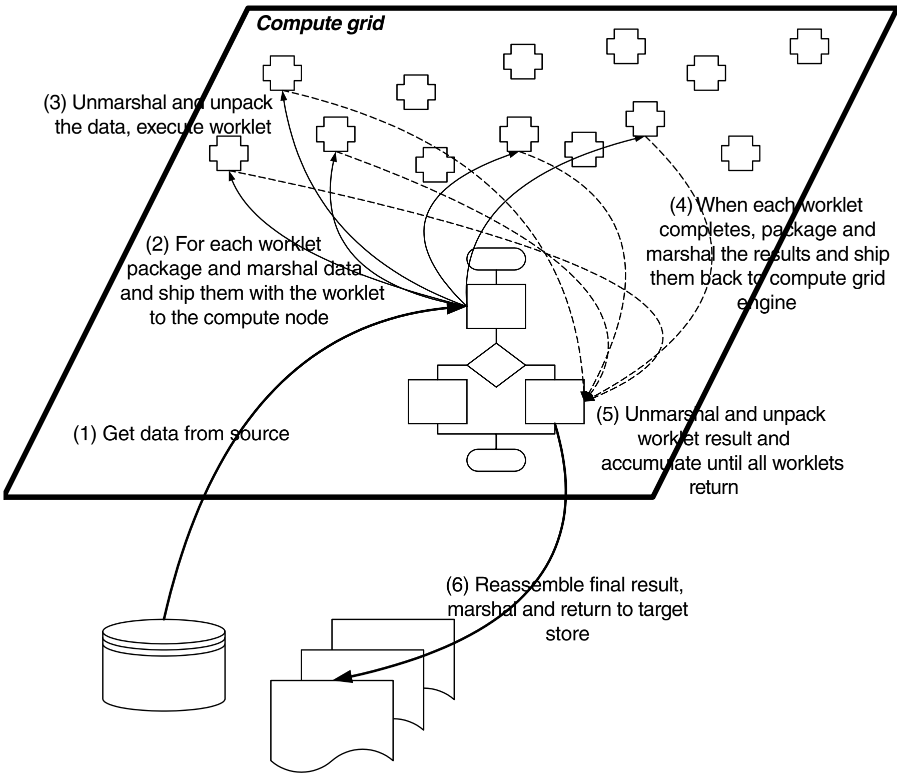

Compute grid
(3) Unmarshal and unpack the data, execute worklet
(2) For each worklet package and marshal data and ship them with the worklet to the compute node
(4) When each worklet completes, package and marshal the results and ship them back to compute grid engine
(1) Get data from source
(5) Unmarshal and unpack worklet result and accumulate until all worklets return
(6) Reassemble final result, marshal and return to target store

Figure 14.2. Workflow without a data grid.

back to the compute grid management process. The compute grid management process receives the resulting data sets from all the worklets that it dispatched, unpackages each one (again, part of the unpacking process consists in unmarshaling the data), and assembles them back together to form one cohesive data result set for final storage in some sort of persistence.

As can be seen, this process of packing, marshaling, sending, receiving, unmarshaling, and the unpackaging of data must happen twice, once for the input data to the worklets and once for the resulting data set of the worklets. This process must occur for each worklet followed by a data assembly process so that the worklet results can be understood at the application level. The more worklets that the compute grid dispatches, the better the parallelization of the overall process will be, but this, too, has a price. The efficiency gained by parallelizing the work is counterbalanced, either in part or in whole, by the overly complex data packaging/marshaling work.

Some level 0 data grids mitigate this performance consequence by creating a distributed file system across the compute nodes of the compute grid or via GridFTP. Each transfers the responsibility of data packaging and transport from the compute grid to the level 0 data grid; however, there may still be data marshaling involved by either the data grid or the worklets–application combination. However, these solutions do not address performance enhancement techniques such as data affinity.

The current commercially available grid solutions are tailored to these calculation-intensive applications with the static data sets. However, as corporate America adopts grid technology and begins to leverage it throughout the organization to encompass tasks beyond those of static data sets, compute grid solutions will need to be augmented with a data grid that also manages dynamic data, a level 1 data grid.

Unless running overnight batch processes, the majority of business applications is dynamic in nature and requires level 1 data grids. Some of the examples listed earlier for risk management, in either financial services or government security, are characteristically real-time and dynamic data sets. Figure 14.3 illustrates the integration of the data grid into the compute grid architecture, where many worklets perform their defined tasks in parallel.

Level 1 data grids inherently lend themselves to the transient data sets produced by calculation-intensive processes such as a Monte Carlo simulation. These processes generate and leverage vast amounts of interim data that are used throughout the running simulation to produce an end result but are not part of the end result itself. A comparison of the quantity of data input and output from a Monte Carlo simulation to that of the interim data generated by the running simulation is analogous to an iceberg, with the input and output data representing the tip of the iceberg and the interim data as the majority of the iceberg that you do not see.

In the specific case of a Monte Carlo simulation used in the financial markets, the interim data sets can be, but not limited to, random-number surfaces and yield curves. Worklets produce and consume these interim data on a regular basis. Some produce the random-number surfaces; others produce the yield curves, while others will use parts of each to produce other interim data surfaces, all leading to the final resulting data surface. In a later chapter an example of the code for part of

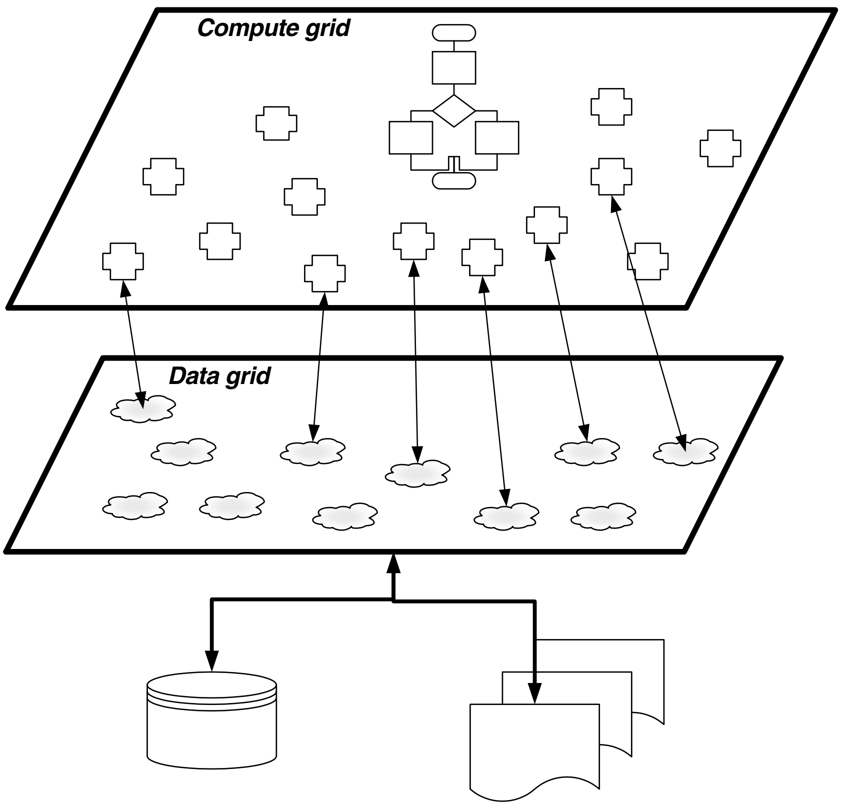


Figure 14.3. Workflow with the data grid: level 1 data grid.

a Monte Carlo simulation is provided using a level 1 data grid. One of the main objectives in creating and maintaining a Monte Carlo simulation is performance tuning. There are various methods of increasing the performance of a simulation. Here I will introduce performance enhancement techniques that a level 1 data grid offers above and beyond that gained by traditional computational performance enhancement techniques.

Data enhancement techniques take two forms: data reuse and data affinity. For some simulations, many of the interim data surfaces can be reused from one simulation to another. Keeping these surfaces active in a data grid eliminates the need to continually regenerate them from one simulation run to another. While a level 0 data grid can address data reuse, it may not yield any performance benefits as it may be faster to regenerate on a local node than to query, locate, transport, package, and marshal the data out of the data grid and to the compute node where they are needed. A level 1 data grid inherently addresses many of the data accessibility issues inherent into a level 0 data grid, thus minimizing much of the work and performance bottlenecks to such an extent that it would be cheaper to reuse interim data surfaces than to regenerate them from one simulation to another.

Data affinity is addressed by level 1 data grids. *Data affinity* is the ability to group interim data sets to the compute nodes that most often generates and uses them, thus further reducing data movement. With this strategy, data are locally resident on the compute node, eliminating the entire data packaging–movement process overhead. Between data reuse and data affinity in the data grid, the overall processing time of a Monte Carlo simulation can be dramatically reduced by as much as half.

DATA GRID ANALYSIS

In order to analyze calculation-intensive application with the data grid, I will leverage the Monte Carlo simulation as a model. The first step is to start with the application definition expressions as presented in an earlier chapter.

The application definition equation for a distributed environment is

$$Application(Work(\,), Data(\,), Time(\,), Geography(\,), Query(\,))$$

where

$$\begin{aligned}
&Work(batch/atomic, synchronous/nonsynchronous)\\
&Data(overallsize, atomicsize, transactional, transient, queryable)\\
&Time(Real\text{-}Time, NotReal\text{-}Time, NearReal\text{-}Time)\\
&Geography(Topology, NetworkBandwidth)\\
&Query(basic, complex)
\end{aligned}$$

The "interim" data surfaces of a Monte Carlo simulation can be prebuilt and subsequently used to produce the final result surface (the output result set of the simulation). However, some of the interim result surfaces may be dependent on other interim surfaces being partially or completely built. For the purposes of this discussion, we will consider the building of an interim data surface as a "batch" process even though in reality it is a completely parallelized grid process. There exists an interdependency of interim result surfaces that prevents all interim result surfaces from being simultaneously generated independently of each other. Figure 14.4 represents an example of such interim dependency.

Therefore, when there is an interdependency of two surfaces as illustrated in Figure 14.4 for A and B, where B is contingent on A being in place before the building of B can start, then coordination of the two processes is necessary. Further complexities will exist when only parts of a surface are dependent on parts of another surface. For instance, to continue with our example illustrated in Figure 14.4, not all of surface A must be completely built before construction of surface B can start. Therefore, this will result in creation of a partial dependency between the two surfaces. In both instances there exists an element of synchronization of processing with regard to start building interim surface B only when "*X%*" of surface A is complete, where the internal processing of both A and B are completely atomic and

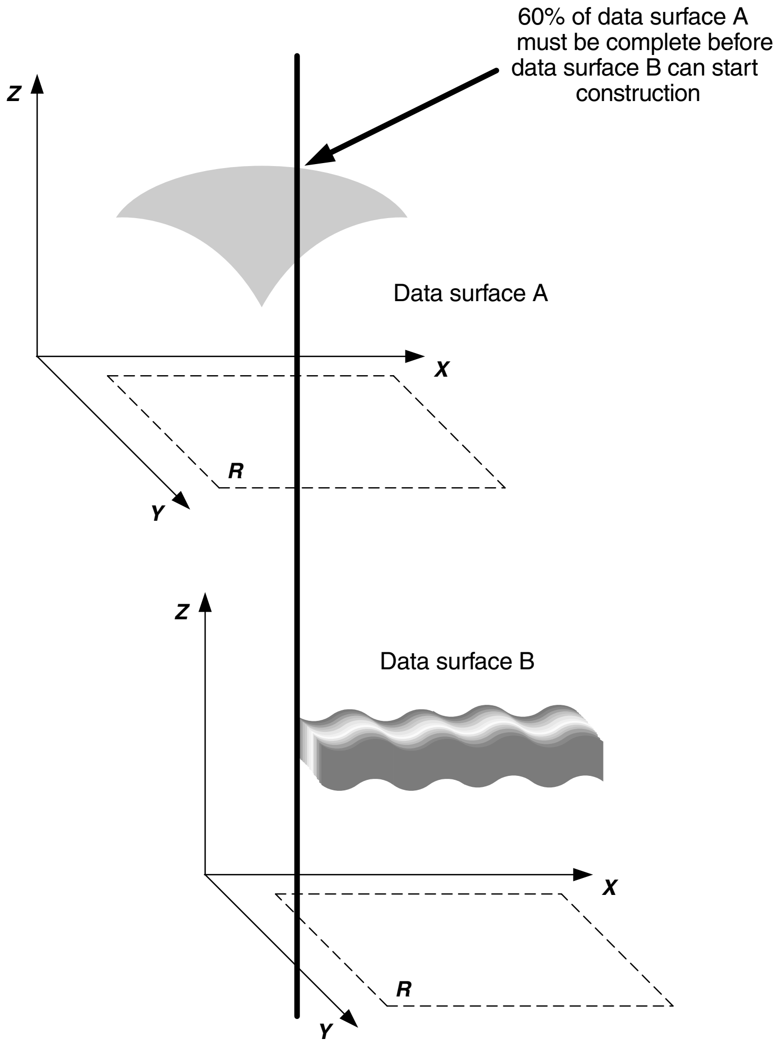

60% of data surface A must be complete before data surface B can start construction
Z
Data surface A
X
R
Y
Z
Data surface B
X
R
Y

Figure 14.4. Inter data surface dependencies.

nonsynchronous. In this case *X%* represents some variable of completeness. From the Monte Carlo expressions in the earlier chapter, the Monte Carlo simulation was represented as

$$MonteCarloSimulation\begin{pmatrix} Work(W\_T1(),W\_group1(W\_T2(),W\_T3())\ldots,W\_Tn()), \\ Data\_input(),Data\_output(),Data\_S1(),\ldots Data\_Sk(), \\ Time(), \\ Geography(), \\ Query() \end{pmatrix}$$

where

> *"W_Tx" represents a task that has no dependencies on another task;*
> *"W_group" represents a grouping or tasks where interdependency exists, thus an element of synchronicity.*
> $W\_T1(atomic, nonsynchronous)$
> $W\_T2(atomic, nonsynchronous)$
> $W\_T3(atomic, nonsynchronous)$
> $W\_Tn(atomic, nonsynchronous)$
> $W\_group1(batch, synchronous)$

The input and output data surfaces are small in comparison to the interim data surfaces that will be necessary to derive the final output data surface.

$$Data\_input(1kbits, 100bits, nontransactional, transient, nonqueryable)$$
$$Data\_output(1kbits, 100bits, transactional, transient, nonqueryable)$$
$$Data\_S1(3Gbits, 100bits, nontransactional, transient, queryable)$$
$$Data\_Sn(3Gbits, 100bits, nontransactional, nontransient, queryable)$$

The ability to run simulations in near real time enables business decisions to be made with accurate and timely data. This simulation is to run in the confines of a single data center and the applications requirement to analyze (complex queries) any of the data sets is not essential to the business.

$$Time(Near\text{-}Real\text{-}Time)$$
$$Geography(DataCenter, 1GbitEthernet)$$
$$Query(basic)$$

The data management policies that need to be imposed are represented as

$$DataDistributionPolicy = DDP\left(\begin{array}{l} MonteCarlo\_DDP, \\ MCRegion, \\ Scope(ALL), \\ Pattern\left(Automatic, Random\left(\begin{array}{l} MCDDPPattern, \\ WhiteNoise(), \\ NULL, \\ NULL, \\ NULL, \\ NULL \end{array}\right)\right) \end{array}\right)$$

$$DataReplicationPolicy = DRP\begin{pmatrix} MonteCarlo\_DRP, \\ MCRegion, \\ 7, \\ Scope(ALL) \end{pmatrix}$$

$$SynchronizationPolicy = SP\begin{pmatrix} MonteCarlo\_SP, \\ MCRegion, \\ Scope(Boundary(\text{``}intra\text{''}), NULL), \\ Transactionality(\text{``}nontransactional\text{''}), \\ LoadStore(List(\text{``}MarketFeed\_DLP\text{''}), NULL), \\ Events(NULL) \end{pmatrix}$$

Use if Events to coordinate interdependencies between data surfaces

$$EventNotificationPolicy = ENP\begin{pmatrix} W\_Group\_SurfaceCoordination, \\ MCRegion, \\ Scope(List(W\_Group\_Atoms)), \\ StartDependentSurfaceBuilds(\,) \end{pmatrix}$$

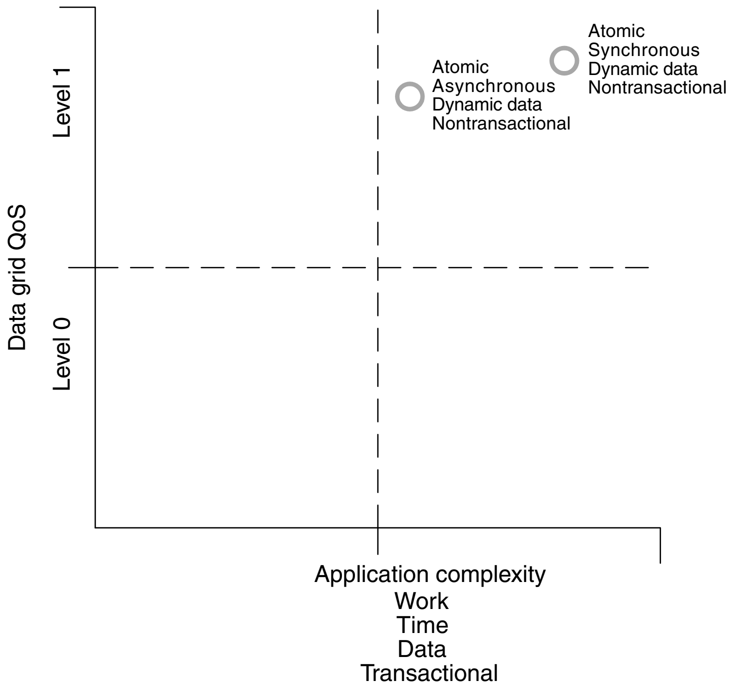


Figure 14.5. QoS–application requirement quadrant graph.

$$DataLoadPolicy = DLP\begin{pmatrix} MarketFeed\_DLP, \\ MCRegion, \\ Granularity(Grouping(1), Frequency(50)), \\ MarketDataAdapter() \end{pmatrix}$$

$$DataStorePolicy = N/A$$

The graph in Figure 14.5 shows nontransactional application characteristics; however, some, but not all, of the application's data surfaces are dependent on each other, thus creating a dual characteristic of nontransactional for data surfaces with no dependencies and transactional where the data surfaces are interdependent.