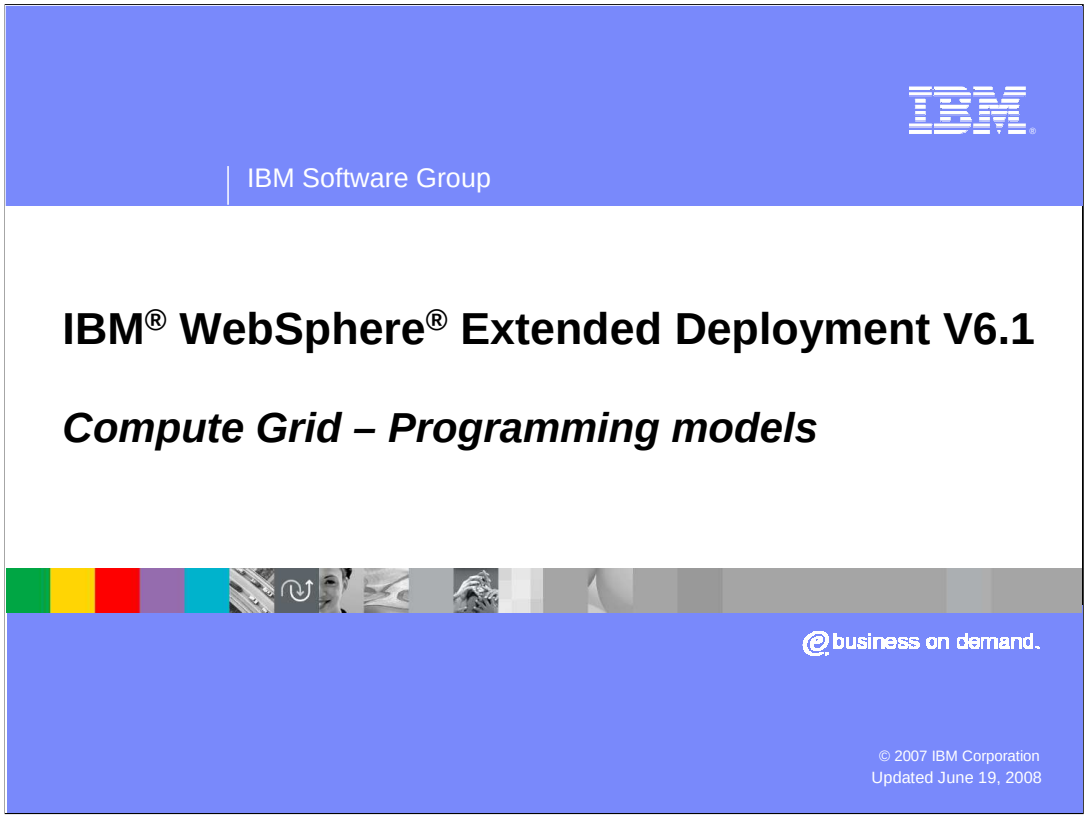


This presentation will provide an explanation of the programming models that can be used with the Compute Grid component offered in WebSphere Extended Deployment V6.1

This module references WebSphere Extended Deployment Operations Optimization, which is now called WebSphere Virtual Enterprise.

Though the module uses the previous names, the technical material covered is still accurate.

Agenda

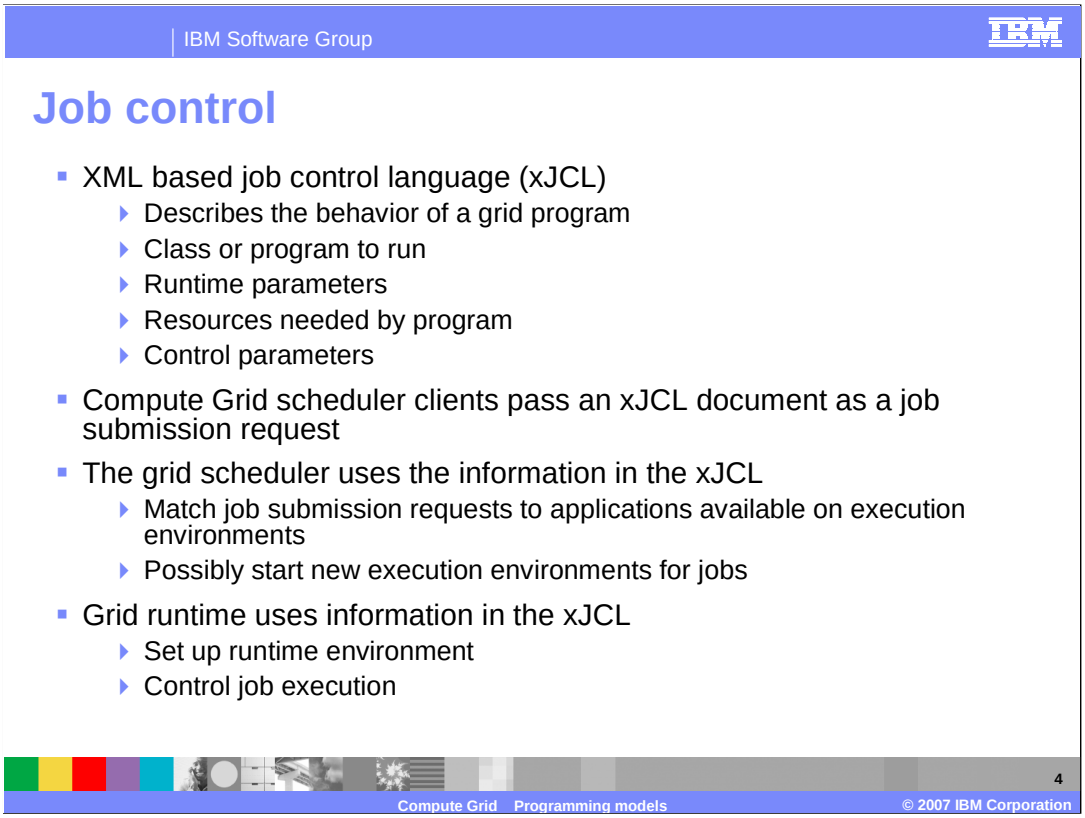
- Compute Grid programming models
  - Job control (xJCL)
  - Batch programming model
  - MVS batch
  - Job life cycle
  - Compute intensive

This presentation will introduce the job control language used to configure and control job execution, and then it will explain the batch process programming model that is supported by Compute Grid and discuss how this maps naturally to MVS batch. Next it will discuss Compute Grid integration with native z/OS batch capabilities and support for native, non-J2EE, applications. Finally, this presentation will cover the computationally intensive programming model.

Section

Job control (xJCL)

This section will explain the batch job control language used by WebSphere extended deployment.



The behavior of compute grid long-running application must be defined within an XML based Job Control Language, called xJCL. The xJCL definition of a job is not part of the application, but is constructed separately and submitted to the job scheduler.

xJCL has constructs for expressing all of the information needed for compute-intensive, batch, and native jobs; though some xJCL elements are only applicable to specific job types. The xJCL for a job specifies the “program” to run for each job step. The program could be a Java™ class for compute intensive jobs or JNDI name of a stateless session bean for batch jobs. The xJCL also specifies any runtime resources needed by the executable, such as environment variable and runtime parameters; and information used by the runtime to control the job.

A job can consist of multiple steps which run sequentially.

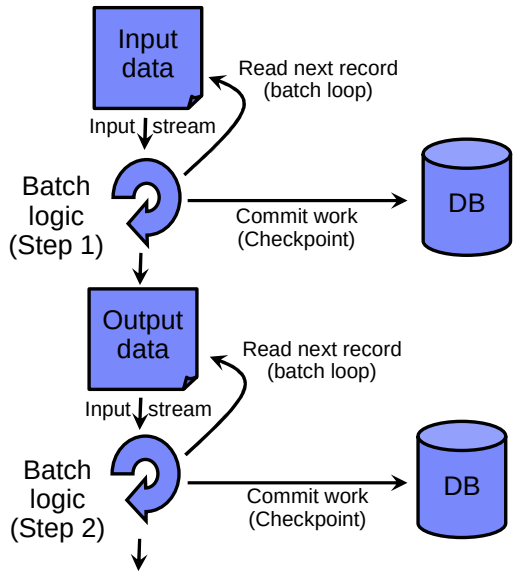
The job scheduler uses information in the xJCL to determine where and when the job should be run, matching the job to available nodes within the environment.

Section

Batch programming model

This section will explain the batch process programming model.

Flow of a typical batch program



- Typical batch process
  - Read data (1 or more records) from an input stream
  - Perform batch process logic using data
  - Write any output data if needed
  - Commit work performed by batch process to a database (checkpoint)
  - Loop to get next record
- Each batch job can be made up of multiple batch steps
- The output from one batch step can be an input to another batch step

In a typical batch process the application will read data from an input stream, perform business logic on that data, write output data if needed, commit the work to a database, and then loop to the next record to repeat the process. A batch job can be comprised of one or more batch steps, and the output from one batch step can be the input into another batch step in the process. Dividing a batch application into steps allows for separation of distinct tasks in a batch application. A batch job can be made up of any number of individual batch steps.

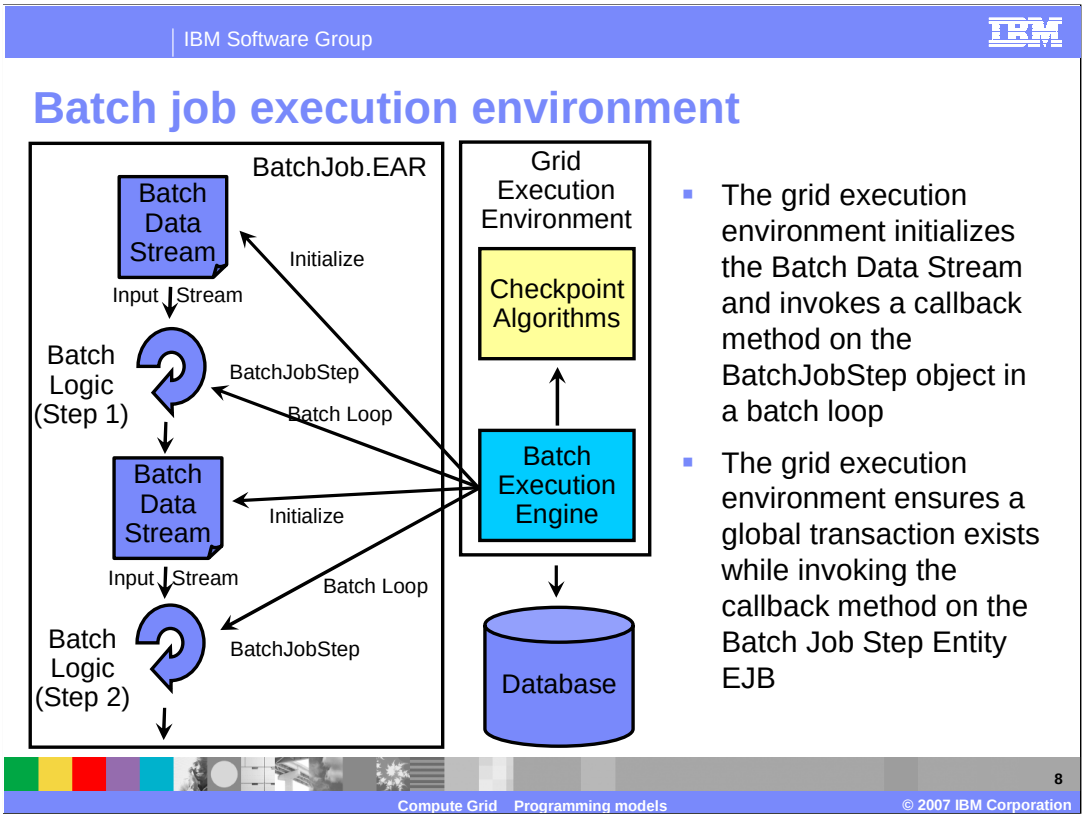
Batch processing in WebSphere

- J2EE-based batch processing programming model
- POJO programming model
- Batch data stream
- Supports the re-use of existing services in “batch mode”
- xJCL is used to describe the behavior of a grid batch program
  - xJCL: <jndi-name> element inside <job-step>

A batch application in WebSphere Extended Deployment Version 6.1 can be implemented as Enterprise Java Beans (EJB), or simple Plain Old Java Objects (POJO). Batch applications follow a few well-defined interfaces that allow the batch execution environment to manage the application for batch jobs. The batch application components are packaged as EJB modules in an Enterprise Archive (EAR) for deployment.

A batch data stream is a Java class that implements the input or output streams that contain the data processed by a batch step. Methods on the BatchDataStream interface allow the batch execution environment to manage the data stream being used by a batch step. For example, one of the methods retrieves current cursor information from the stream to keep track of how much data has been processed by the batch step.

The compute grid support for batch applications allows legacy batch applications and J2EE batch applications to be freely mixed in an MVS job. Legacy batch applications can be appropriately converted to J2EE applications that can be managed by WebSphere. xJCL is used to describe the behavior of the job to the environment.

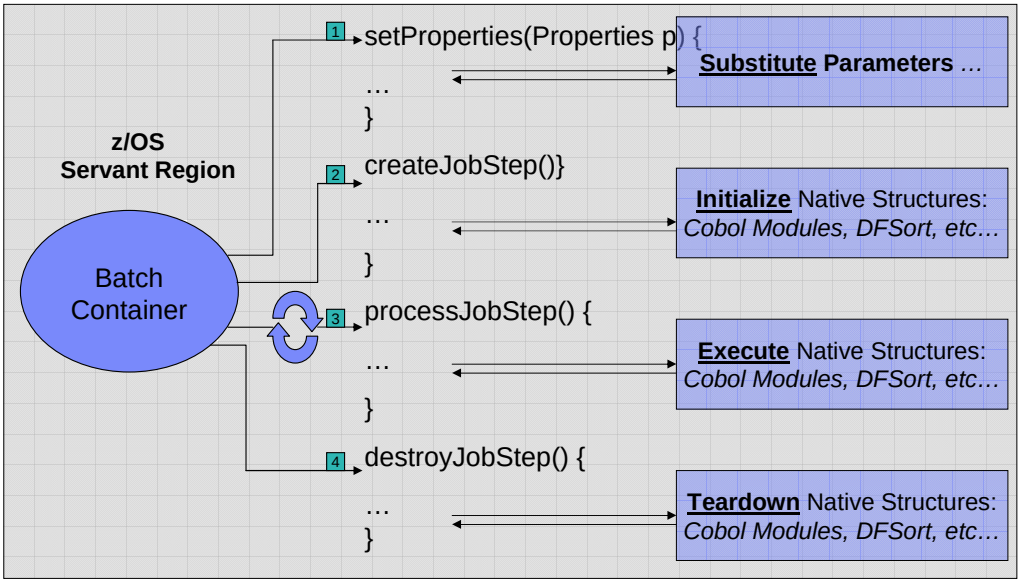


The life cycle of a MVS batch job translates naturally to WebSphere Extended Deployment. The key actions to perform a J2EE batch step are shown here. Basically it is to initialize a step, perform the step for each input record and when finished take the step down.

For each job step, the grid execution environment first instantiates and initialized the application's BatchJobStep bean as specified by the xJCL for the job step. Initialization includes setting properties and creating batch data streams defined for the job step. Once the environment is initialized the grid execution environment repeatedly invokes the BatchJobStep's processJobStep until this method indicates it has processed all of its input. This processing loop takes place in a global transaction. On each iteration of the loop the grid execution environment queries a checkpoint algorithms to decide how often to commit the global transaction. Checkpoints allow a job to resume at an intermediate point if it fails and must be restarted.

A J2EE batch application is required to declare a special stateless session bean in its deployment descriptor. This bean acts as a batch job controller and must contain enterprise beans-references to all the batch step enterprise beans being used in the batch application. The implementation of this bean is provided by WebSphere, not by the batch application; it only needs to be declared in the batch application's deployment descriptor. Only one controller bean can be defined per batch application.

J2EE batch and traditional z/OS interoperability

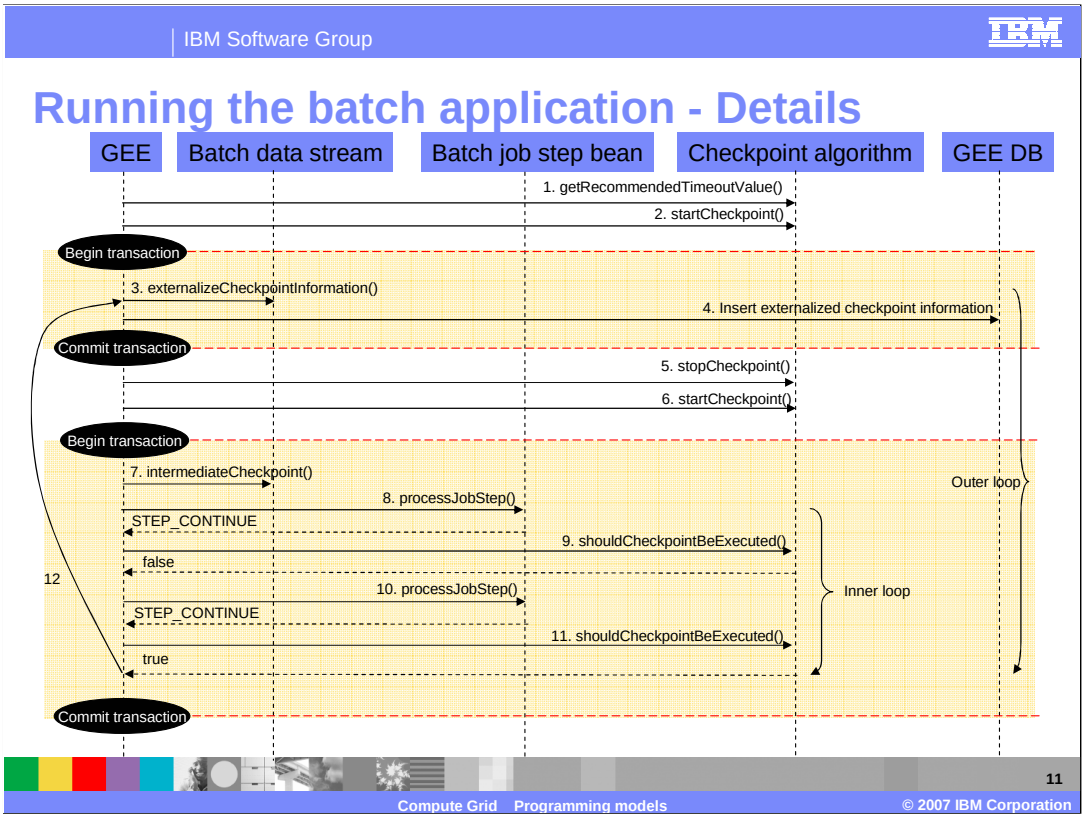


The life cycle of a batch job step translates naturally to WebSphere Extended Deployment. The basic steps in executing a traditional step of a batch job are shown on the right. For a traditional batch job these functions are performed by JES. The general flow is to perform parameter substitutions in the JCL (actually done at the start of a job), set up access to data sets, libraries and other initializations, run the program and then clean up. A J2EE batch bean performs equivalent operations as shown in the middle of this slide. In particular the batch bean calls setProperties in your application to record parameter values, createJobStep to initialize and so forth.

Checkpoint algorithms

- Checkpoint algorithms control the life cycle of the global transactions started by the grid execution environment
  - Upon committing the global transaction the grid execution environment retrieves cursor information from the batch data stream and stores it to the batch database
- Checkpoint policies are applied to a batch job when it is submitted
  - These policies determine which checkpoint algorithm to use for a particular batch job
- WebSphere Extended Deployment V6.1 contains 2 checkpoint algorithms
  - Time based
  - Record based
- An interface is also provided
  - write custom checkpoint algorithms
  - plug them into a grid execution environment through xJCL

The grid execution environment uses checkpoint algorithms to decide how often to commit global transactions under which batch steps are invoked. These algorithms are applied to a batch job through the xJCL definition. Properties specified for checkpoint algorithms in xJCL allow for checkpoint behavior, such as transaction timeouts and checkpoint intervals, to be customized for each batch step. WebSphere Extended Deployment provides time-based checkpoint and record-based algorithms. It also defines a service provider interface for building custom checkpoint algorithms. On each batch step iteration, the grid execution environment consults the checkpoint algorithm applied to that step to determine if it should commit the global transaction. Callback methods on the checkpoint algorithms allow the grid execution environment to inform the algorithm when the global transaction is committed or started.



This slide illustrates the key steps within the grid execution environment's processing loop for a job step. Notice that a global transaction is bounded by the checkpoint interval. All work done within that checkpoint interval would be rolled back in case of failure, but upon success all work is committed together. This allows a job to be restarted at the last checkpoint in case of a failure during the job processing.

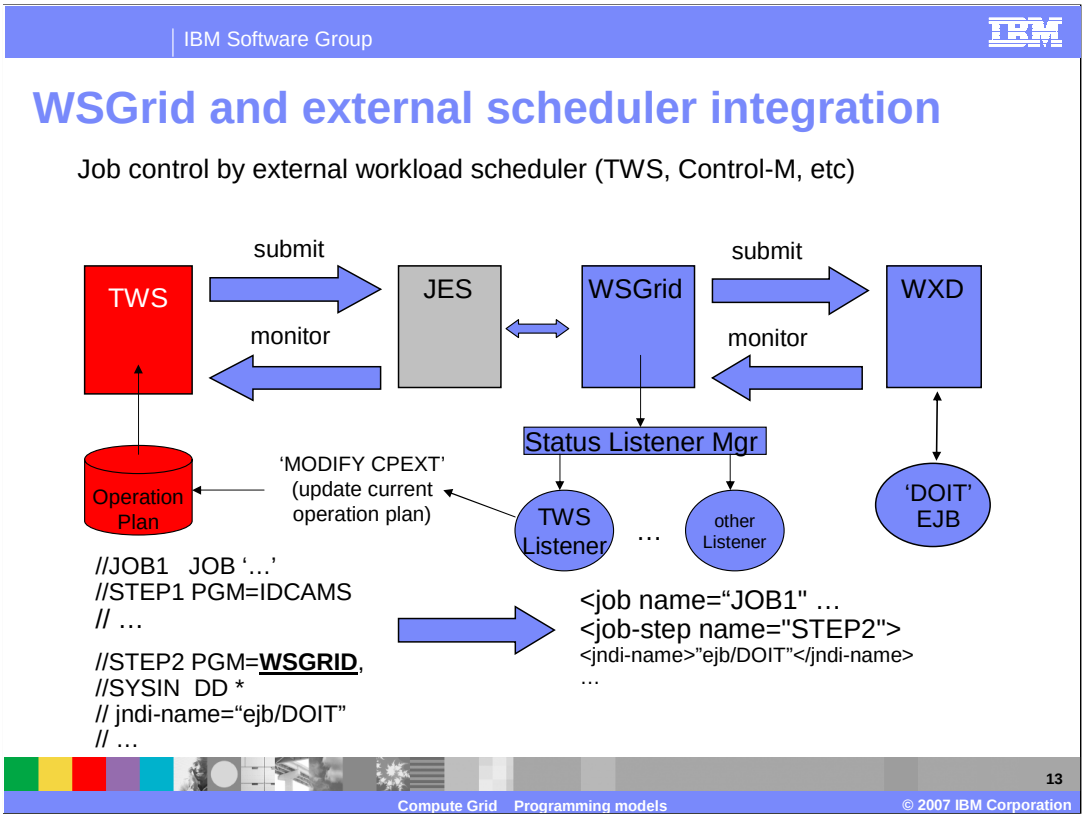
In the sequence shown, steps 2 through 5 perform an initial checkpoint when just beginning to run a J2EE batch step. First the batch bean verifies it needs to perform a checkpoint by calling the checkpoint algorithm. Next the batch bean retrieves the batch data stream checkpoint information for each batch data stream and then sends that information to the grid execution environment library to record the checkpoint. The final piece of the initialization is to notify the checkpoint algorithm the checkpoint has completed and then the start of the next checkpoint sequence in step 6. Step 7 notifies the batch data stream to set to mark the beginning of a new checkpoint. The batch bean executes the batch process, querying the checkpoint algorithm when to perform a checkpoint. This continues until the batch job step bean signals completion.

The sequence for restarting a step at a checkpoint is similar except steps 2 through 5 are replaced with retrieving the checkpoint for the execution environments data base and setting the batch data streams to the selected checkpoint. The processing then continues as shown here.

Section

MVS batch

This section will explain native execution jobs.



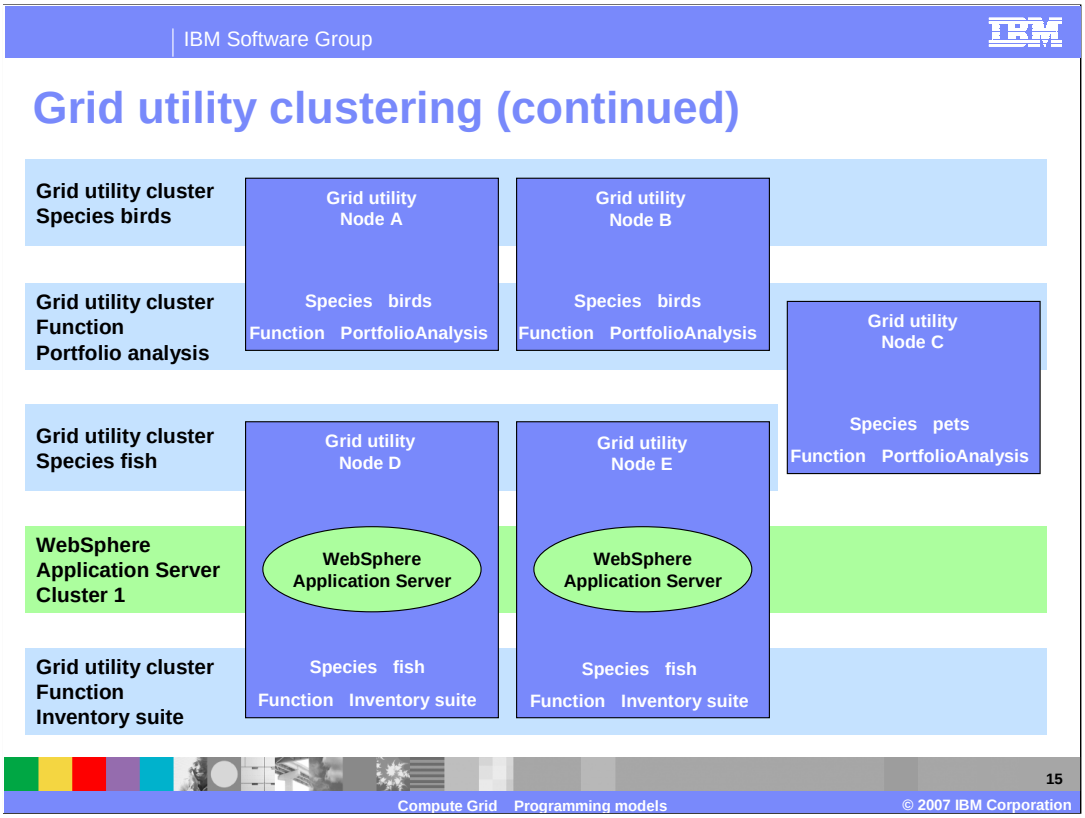
WSGrid is a z/OS JCL application and is called as shown in the lower left. To start the interaction, Tivoli Workload Scheduler (TWS) starts a z/OS task (JES). As shown here, step is traditional batch step and proceeds as always. Tivoli Workload Scheduler monitors the job until the step completes. The second step is a J2EE batch step and requires an ear file called DOIT to complete the work. This job step uses a new interface called WSGRID. When WSGRID executes in JES, it will read the xJCL from SYSIN and submits that to grid job scheduler. As resources allow, the DOIT application will run on a WebSphere node reading and writing batch data streams. To complete the picture and make the J2EE batch behave as traditional batch, J2EE batch has to monitor the progress of DOIT and report back to an external workload scheduler if present. WSGRID performs this monitoring and supports a listener framework. This picture shows a Tivoli Workload Listener plugged into the status listener framework. Here the TWS_Listener modifies the operational plan for Tivoli Workload Scheduler. While you view the Tivoli Workload Scheduler console the J2EE batch display the same as traditional batch.

Grid utility clusters for Compute Grid

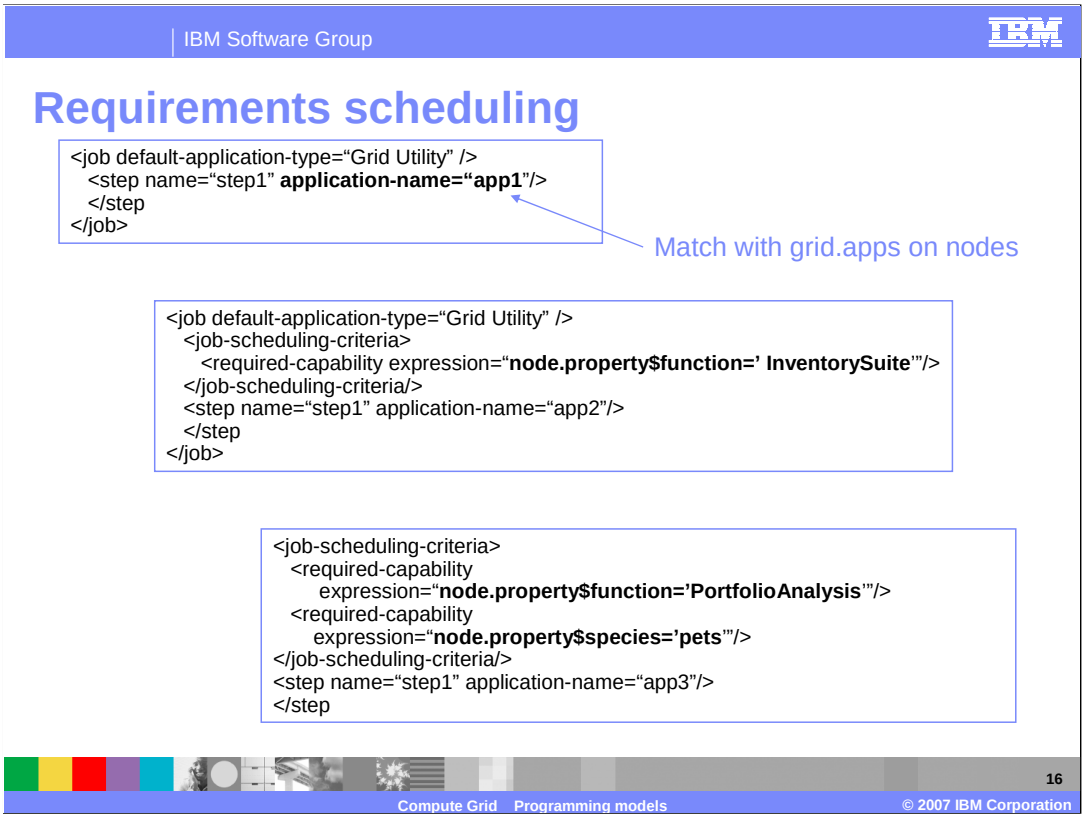
- Group of servers with similar capabilities
  - Capabilities specified as text pairs
    - Node custom properties
  - Text names and values are not pre-specified
  - Example:
    - Function='PortfolioAnalysis'

Grid utility clusters are logically formed based on the intersection of advertised capabilities. A grid utility node may simultaneously be a member of multiple clusters. Moreover, grid utility clusters may heterogeneously be comprised of both grid utility and WebSphere nodes with common advertised capabilities. Capabilities are optionally defined to the grid utility server by setting *node custom properties*. The example shown indicates that the node is capable of performing the 'portfolio analysis' function. The choice of capability names, values, and semantics is completely user defined with the 'grid.apps' and 'grid.env' custom properties being reserved names.

The Job Scheduler assigns work based on a match between a job's needed capabilities as specified in its xJCL and both grid utility and WebSphere nodes' advertised capabilities.



This picture shows four grid utility clusters formed from two capabilities ‘Species’ and ‘Function’. Each of these clusters is a virtual cluster -- their membership is determined by their claimed capabilities. In addition to the grid utility clusters there is a cluster of two WebSphere Application servers that was created by normal WebSphere administrative processes. In the case shown, all the nodes shown are each in two clusters, for example nodes ‘A’ and ‘B’ are in the grid utility clusters with ‘Species = birds’ and ‘Function = Portfolio Analysis’. In a similar way, the WebSphere Application Server nodes host the two grid utility clusters with “Species=fish” and “Function = Inventory suite”.



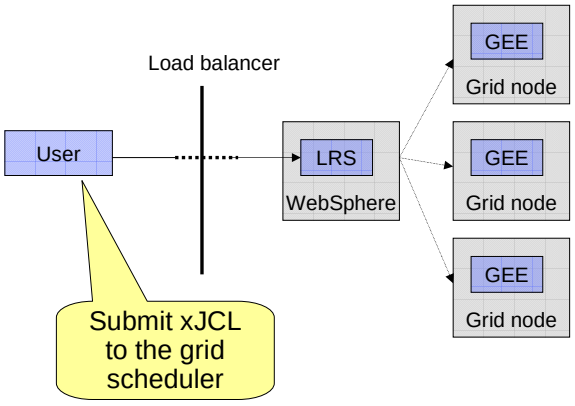
Here are a few examples of xJCL specifying capability requirements that refer to the clustering shown on the previous slide. The top one will need to be scheduled on an any node with the application app1 declared in the 'grid.apps' custom variable. The middle example will be scheduled on nodes 'D' or 'E' in the previous slide. The bottom example would be scheduled on node 'C'.

Section

Job life cycle

This section will provide an overview of the life cycle of a typical job in WebSphere Extended Deployment Compute Grid.

Submitting Extended Deployment batch jobs

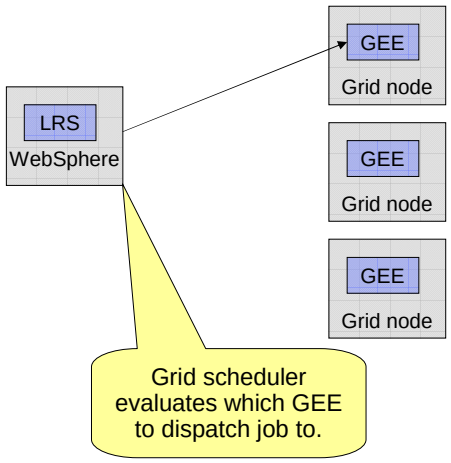


Sample xJCL

```
<job name="SampleJob">
  <jndi-name>batch.samples.sample1</jndi-name>
  <checkpoint-algorithm name="timebased">
      <classname>checkpoints.timebased</classname>
      <props>
          <name="interval" value="15"/>
       </props>
  </checkpoint-algorithm>
  <job-step name="Step1">
    <jndi-name>batch.samples.step1 </jndi-name>
    <checkpoint-algorithm-ref name="timebased"/>
    <batch-data-streams>
        <bds>
            <logical-name> mylnput </logical-name>
            <impl-class> bds.sample1 </impl-class>
             <props>
                 <prop name="FILENAME"
                       value="/tmp/input"/>
             </props>
        </bds>
    </batch-data-streams>
  </job-step>
</job>
```


To run a job, you first submit an xJCL description of the job to the grid scheduler, depicted as “LRS” in this figure. The scheduler evaluates the xJCL and, based on this information determines which WebSphere Application Server managed servers in the cell are capable of running the job, and sub-schedules the job to one of these servers. To be eligible to run a job, a server must host the target grid execution environment-enabled application specified in the xJCL.

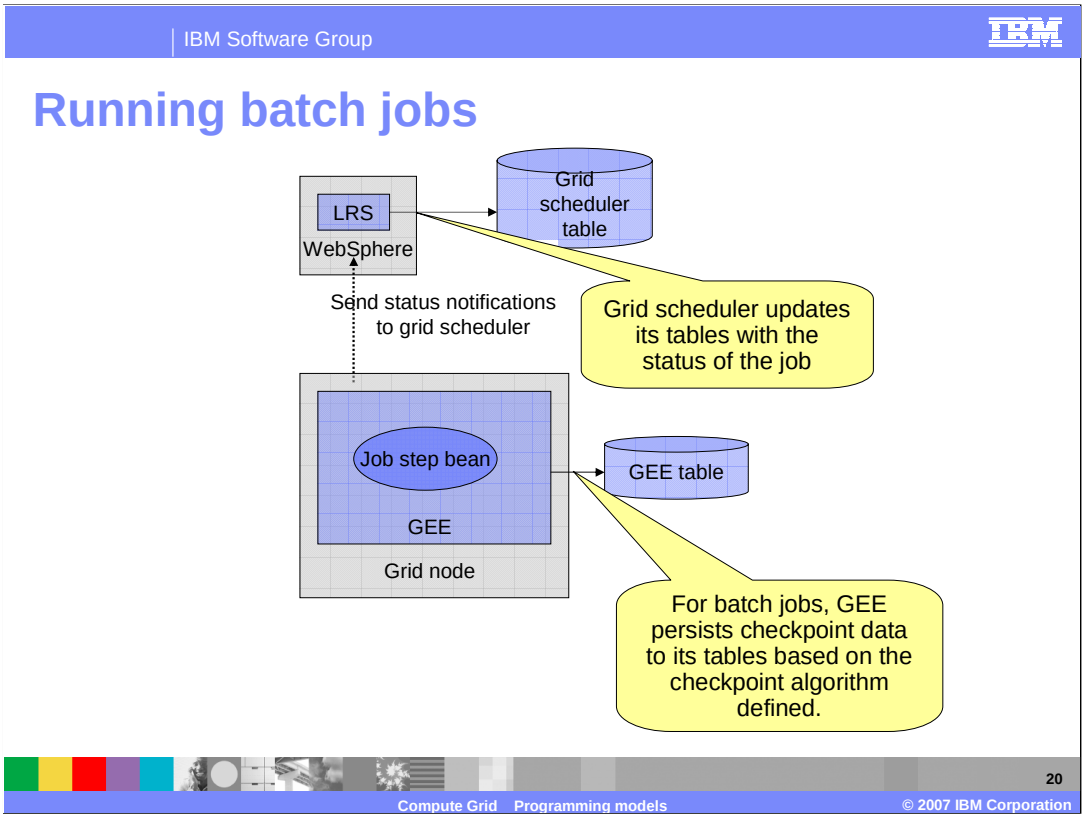
Dispatching batch job



The grid execution environment (GEE.ear) is a J2EE application that runs within a WebSphere Application Server. On z/OS the grid execution environment runs within a **Servant Region** and servant region behavior is applied to batch jobs. That is, if workload and service policies deem it necessary, new servants can be dynamically started or stopped. On distributed systems with WebSphere Extended Deployment Operations Optimization installed, the grid scheduler will work with the application placement controller to start or stop server instances as needed to run the job. The scheduler will dynamically balance the needs of long-running work against the needs of transactional applications within the cell.

WebSphere Extended Deployment 6.1, introduces several new service provider interfaces that allow more control over how a batch job is dispatched. They can be used to override the chosen grid execution environment target, force authorization to take place before accepting the job, assign a specific job class (which maps to a service policy) and schedule the job to run at a later time.

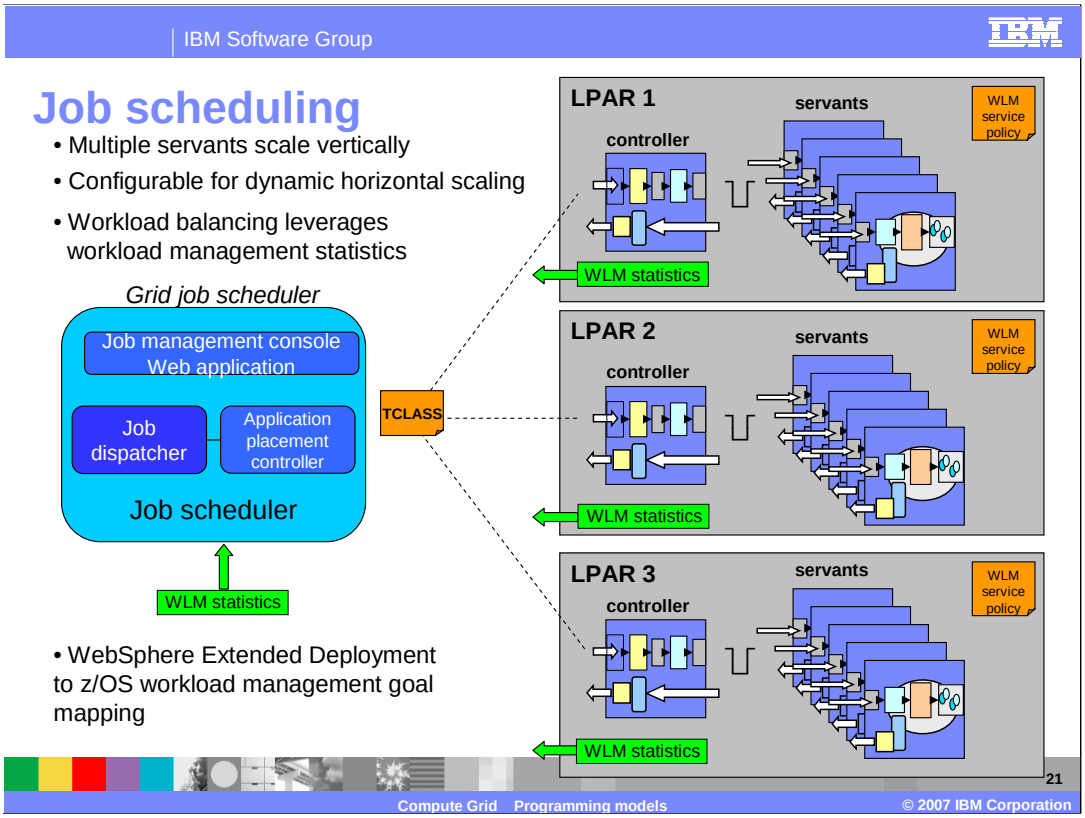
These interfaces are completely pluggable and highly customizable.



The grid execution environment runs the job step with the properties specified in the xJCL that was submitted. For batch jobs, checkpoint data is periodically persisted to the grid execution environment database for restartability. The data that is stored during checkpoint processing, such as current location within the batch data streams, is entirely provided by the application. For compute intensive jobs the grid execution environment provides a process thread and minimal runtime support. For native jobs, the grid node runs the specified program as a separate, native process.

The grid scheduler listens for updates from the execution environment, such as ‘job failed’ or ‘job run successfully’, and updates the grid scheduler database accordingly.

Starting in Extended Deployment version 6.1 the grid scheduler provides WS-Notifications so non Extended Deployment components can register as listeners for status on a particular job.



This slide points to the similarity between the job scheduler and the on-demand router when Compute Grid and Operations Optimization are both present. Both use many of the WebSphere Extended Deployment workload management tools like the application placement controller. Like the on demand router, the job scheduler interacts z/OS work load manager, provides horizontal scaling to the z/OS work load manager's vertical scaling. You read more about dynamic computing in that section.

Section

Compute intensive programming model

This section will explain the computationally intensive programming model.

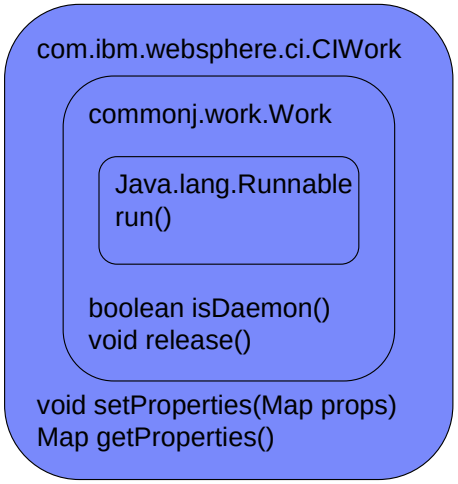
Compute intensive work

- Compute-intensive work typically requires significant amounts of processing to complete
- xJCL describes the behavior of a grid program
  - Compute grid Scheduler clients pass an xJCL document as a job submission request
  - xJCL: <class-name> element inside <job-step>
- The Grid Scheduler uses the information in the xJCL to match job submission requests to applications available on execution environments
  - Possibly starting new execution environments for jobs

A computationally intensive application is an application that requires large amounts of processing to finish. For compute intensive jobs, the xJCL must specify the fully-qualified name of the class that implements the compute intensive job and any runtime properties. Runtime properties are specified as name value pairs and are passed in to the compute intensive application in a map object that you can use to change the runtime behavior of the code. The scheduler component passes an xJCL document as part of a job submission request. The scheduler uses the information in the xJCL to match the job to available nodes within the environment.

Compute-intensive programming model

- Each job step specifies the name of a class that implements the interface com.ibm.websphere.ci.CIWork
  - A sub-interface of commonj.work.Work
- Additional constraints on classes that implement CIWork:
  - Work.isDaemon() must return true
  - Must have no-argument constructor
  - Strongly encouraged to provide robust implementation of Work.release()



A special interface is used to define the steps of a computationally intensive application. Each step is represented by a class that implements the CIWork interface which is a part of the WebSphere asynchronous bean programming model. Each step's class must have a no-argument constructor and the Boolean isDaemon() method must return true. Developers are also encouraged to provide an implementation for the release() method, which will be used to remove this step from the grid environment if a job is cancelled.

Stateless session bean facade

- Since asynchronous bean functions can only be accessed programmatically, compute-intensive jobs are also required to define a stateless session bean
- Interface and implementation classes are provided by WebSphere
  - Only the bean definition needs to be included in the application
  - Note that default class-name based JNDI names for stateless session bean will not work, as the same bean will be included in multiple applications
    - Suggested best practice is to append application name as a suffix, “ejb/com/ibm/ws/longrun/LongRunningController-myCIApp”
    - This JNDI name is specified in the xJCL

Each step of a computationally intensive program is written as an asynchronous bean. Since asynchronous bean functions can only be accessed programmatically, the applications must also define a controller bean, which is a stateless session bean defined in the compute-intensive application's deployment descriptor. The controller bean allows the execution environment to control jobs for the application. The implementation of this stateless session bean is provided by WebSphere. The application's only responsibility is to include a reference to the stateless session bean in the deployment descriptor of one of its enterprise bean modules. Exactly one controller bean must be defined for each compute-intensive application. Since the implementation of the controller bean is provided in the WebSphere runtime, application deployers should not request deployment of enterprise beans during deployment of compute-intensive applications.

Summary

- WebSphere Extended Deployment provides an environment for managing and executing batch-style and compute-intensive applications
  - Jobs are scheduled using the long-running scheduler (LongRunningScheduler.ear)
  - Jobs run in the grid execution environment (GEE.ear)
  - Jobs can run on either WebSphere or non-WebSphere nodes
- A WebSphere Extended Deployment Compute Grid will dynamically balance the needs of long-running work against the needs of transactional applications within a cell

In summary, this presentation explained the benefits of the Compute Grid provided by WebSphere Extended Deployment V6. It discussed the differences between computationally intensive and batch programs and how to create them using the different programming models.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_XD61z_ComputeGrid_Programmingmodel.ppt

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM Perform WebSphere

EJB, J2EE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.