### Document information

<table>
<thead>
<tr>
<th>Author:</th>
<th>Rajesh Ramachandran</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version:</td>
<td>1.0</td>
</tr>
<tr>
<td>Date:</td>
<td>3 December 2012</td>
</tr>
<tr>
<td>Reviewed by:</td>
<td>-</td>
</tr>
<tr>
<td>Prepared for:</td>
<td></td>
</tr>
<tr>
<td>Document filename:</td>
<td>IBM WebSphere Compute Grid Capacity Planning Report</td>
</tr>
</tbody>
</table>

### Revision History

<table>
<thead>
<tr>
<th>Revision Number</th>
<th>Revision date</th>
<th>Author</th>
<th>Reviewer</th>
<th>Summary of changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Dec 3, 2012</td>
<td>R. Ramachandran</td>
<td></td>
<td>Original document</td>
</tr>
</tbody>
</table>
# Table of contents

**IBM WebSphere Compute Grid**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM WebSphere Compute Grid</td>
<td>1</td>
</tr>
</tbody>
</table>

**Capacity Planning Methodology for Batch Workloads on z/OS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Executive Summary</td>
<td>4</td>
</tr>
<tr>
<td>1.1 Methodology</td>
<td>5</td>
</tr>
<tr>
<td>1.1.1 WebSphere Application Server for z/OS runtime</td>
<td>5</td>
</tr>
<tr>
<td>1.1.2 Estimating Java CPU based on Existing COBOL</td>
<td>5</td>
</tr>
<tr>
<td>1.1.3 Resource Usage and Scalability</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Scenario</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Results</td>
<td>6</td>
</tr>
<tr>
<td>2 Architecture</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Batch Application Architecture</td>
<td>7</td>
</tr>
<tr>
<td>2.2 System Components and WCG Topology</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Hardware Configuration</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 Software Configuration</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3 WLM Configuration</td>
<td>8</td>
</tr>
<tr>
<td>2.2.4 SMF</td>
<td>8</td>
</tr>
<tr>
<td>2.2.5 WCG Topology</td>
<td>8</td>
</tr>
<tr>
<td>2.2.6 Database Implementation</td>
<td>9</td>
</tr>
<tr>
<td>3 Test Process</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Test methodology</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Batch Test case 1 – million records Results</td>
<td>10</td>
</tr>
<tr>
<td>3.2.1 Observations</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Batch Test case 2 – 10 million records Results</td>
<td>12</td>
</tr>
<tr>
<td>3.3.1 Observations</td>
<td>13</td>
</tr>
<tr>
<td>4 Summary</td>
<td>14</td>
</tr>
</tbody>
</table>
1 Executive Summary

The IBM z/OS platform is well known for its qualities of services such as resiliency, availability and performance. Batch applications, critical to almost every business, are part of a broad set of applications that have run on z/OS batch processing environments for many decades. Much of the applications have been written in languages such as COBOL and Assembler, and executed using enterprise schedulers such as Tivoli Workload Scheduler.

As customers adopt modernization strategies, using JAVA has emerged as a viable approach to batch processing. Java skills are relatively common, Java development tooling has matured, and Java allows customers to increase the degree of reuse across the enterprise.

WebSphere Compute Grid (WCG) is IBM’s modern Java batch processing solution. It provides a batch processing platform that permits sharing applications and management processes across workloads.

As customers begin to adopt Java as a batch processing approach, a challenge that is faced is capacity planning for these new applications. This document provides a starting point for planning capacity for Java batch processing.

Based on customer experiences and benchmark testing, we are able to make the following generalizations about Java batch processing on z/OS:

- The amount of CPU consumed and the time it takes to complete a Java batch process is a function of the batch program design.
- Batch programs written in Java can compare favorably to batch programs written in other languages such as COBOL or Assembler.
- The amount of CPU consumed for Java batch is approximately 2.5 to 3 times comparable COBOL or Assembler. This includes the CPU consumed by the WebSphere Compute Grid runtime platform.
- On z/OS a large portion of the Java batch processing is offloaded to zAAP (or zAAP on zIIP) specialty engines rather than general processor (GP) engines. When compared to the total cost of batch that run exclusively on GP engines, the overall cost of Java batch is often seen as less.

The objective of this study was to come up with a methodology which can be used to help customers estimate how much capacity they would need to support their batch applications running in WCG.

---

1 The IBM WebSphere Compute Grid function is included with WebSphere Application Server Version 8.5. For those with WebSphere Application Server Version 7 or 8, WebSphere Compute Grid Version 8 may be augmented to their existing WebSphere runtime. Compute Grid Version 8 function is identical to that included with WebSphere Application Server Version 8.5.
1.1 Methodology

WebSphere Compute Grid (WCG) is a Java batch runtime platform that runs on WebSphere Application Server. Batch jobs written in Java are submitted to the WCG environment through the Job Scheduler function, and then dispatched for execution to a server endpoint.

When planning to run batch processing in WebSphere Compute Grid on z/OS, there are three capacity planning and sizing considerations to take into account:

1. How much CPU and memory does the prerequisite WebSphere Application Server for z/OS runtime environment consume?
2. How can existing COBOL batch process resource consumption be compared to the same processing being done as Java batch?
3. What is the resource usage and scalability of a Java batch process running in WebSphere Compute Grid?

This document will address the third sizing consideration. In this document we will provide the results from a controlled test of a Java batch program running in WebSphere Compute Grid on z/OS, reading from z/OS sequential data sets and performing updates against DB2 z/OS tables.

1.1.1 WebSphere Application Server for z/OS runtime

The following URL provides a document that includes a methodology to estimate the memory and CPU requirements for a WebSphere Application Server on z/OS runtime environment:


Note: WebSphere Compute Grid (WCG) is a Java batch runtime platform that runs on WebSphere Application Server. Therefore, we may use that document to derive the basic capacity requirements of a WebSphere Compute Grid environment.

1.1.2 Estimating Java CPU based on Existing COBOL

When considering existing COBOL batch processing with an eye towards estimating Java CPU usage, it is important to first determine the amount of existing CPU is associated with DB2.

- If there is no DB2 involved (that is, the existing COBOL is doing strictly file I/O), then the general rule is that Java will require between 2.5 and 3 times the CPU. That number includes the WebSphere Compute Grid infrastructure

  Note: Much of the Compute Grid infrastructure is eligible for offload to z/OS specialty engines such as zAAP or zAAP-on-zIIP. Further, if JZOS utilities are used for file access then all of the CPU associated with JZOS is eligible for offload to specialty engines.

- If DB2 is involved, then the CPU associated with DB2 will be the same whether access is from COBOL or whether from Java. Therefore, the proper approach is to identify and separate CPU associated with DB2, then multiple COBOL-only CPU by the 2.5 to 3 factor. That number includes the WebSphere Compute Grid infrastructure, and much of it is eligible for offload to specialty engines.

1.1.3 Resource Usage and Scalability

Therefore, with respect to the third sizing consideration, the general approach used in this study to size WCG applications was:

- Have a controlled environment in the form of a dedicated LPAR, a fixed number of CPU’s and a fixed amount of available memory

- Measure how many batch jobs could run in an hour with the CPU at full capacity
• Measure the overhead of the WCG infrastructure in terms of CPU used
• Measure the CPU per job

With respect to the Java batch jobs, two approaches were used:

• Run as many jobs as possible concurrently that processed one million records each and write the data to a database
• Run as many jobs as possible concurrently with the number of records processing being the variable changed for each test.

This approach would provide a framework of knowledge of CPU used based on jobs and records processed. From that it would be possible to estimate the expected CPU used for an actual batch workload based on the number of concurrent jobs and the number of records being processed.

1.2 Scenario

The batch application used simulated batch processing in a fictional stock trading company. This company receives hundreds of files from different business partners that need to be processed in a batch environment. These files are received at various times during the day and are processed at night in the batch cycle. The file sizes vary and consist of stock trading records:

1.3 Results

The environment used consisted of a single z196 LPAR with one dedicated general processor and one dedicated zAAP, with 64GB of real memory, and WebSphere Compute Grid configured in 31-bit mode with one scheduler server and one batch endpoint server.

The Java batch application was designed to read from a UNIX file and update records in DB2 on z/OS.

The results showed that we were able to process 100 million records in an hour with each job processing a million records. Running 4 jobs concurrently provided the maximum throughput.

Note: the result seen in this study was a function of this specific environment with this specific application. Your results may vary, depending on many factors that influence overall system performance.
2 Architecture

2.1 Batch Application Architecture

The batch jobs consisted of two steps. The first step creates the files, which were sequential datasets on z/OS. The second step processed the records in the generated file. These records are read one at a time.

After reading the record, the application issued a SELECT statement against the DB2 table to check if the customer ID and stock holding existed. If a record was present in DB2 the existing stock position was updated in a DB2 table holding the user's account? Otherwise, a new record was inserted indicating a new holding acquired by the customer.

WebSphere Compute Grid provides a mechanism for handling checkpoint processing, and the checkpoint interval was set at 3,000 records. Commit processing was handled by Compute Grid's batch container every 3,000 records processed by the Java batch application.

The database was accessed using the DB2 JDBC driver configured to use a Type 2 cross-memory connection from WebSphere Application Server into DB2 z/OS.

The z/OS sequential files were accessed using the APIs supplied with JZOS. For information on the JZOS toolkit provided with the z/OS operating system, see the following website:


2.2 System Components and WCG Topology

2.2.1 Hardware Configuration

The tests were run in a single dedicated LPAR on a z/196 (2817/M80). This LPAR was configured as follows:

- z/196 (2817/M80)
- 1 GCP
- 1 ZAAP
- 64GB Memory

2.2.2 Software Configuration

The software used for this exercise was as follows: ²

- WebSphere Application Server 6.1.1.33
- WebSphere Compute Grid 6.1.1.3
- DB2 V9 for z/OS
- MQ V6
- RMF
- SMF
- z/OS 1.12

² Although specific product levels are indicated, the methodology applies equally to more current versions.
2.2.3 WLM Configuration

The server address spaces were classified to run under CBSTC service class. WLM was setup with a transaction class such that individual jobs (which run as enclaves) ran under the PRDBATHI service class. The goals for each are provided here:

<table>
<thead>
<tr>
<th>Service Class</th>
<th>Importance</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBSTC (server started tasks)</td>
<td>1</td>
<td>Velocity = 90</td>
</tr>
<tr>
<td>PRDBATHI (Java batch jobs)</td>
<td>1</td>
<td>Velocity = 40</td>
</tr>
</tbody>
</table>

Note: as this test was run on a dedicated LPAR with no other work taking place, WLM did not need to prioritize access to system resources based on the goals. The goals are offered here to provide a sense for the definitions used.

2.2.4 SMF

SMF was set up to capture the following records:

SYS(TYPE(0,2,3,4,5,7,20,30,70:79,82,88,100:102,110,120))

2.2.5 WCG Topology

A network deployment configuration of WebSphere Application Server on z/OS (6.1.0.33) was built on a single LPAR. This cell consisted of one node. This node was augmented with WCG 6.1.1.3

The following figure shows the architecture used in the test:

**Figure 1 WebSphere Compute Grid Topology**

The WCG Cell used for this test was configured as follows:

- 1 Scheduler Server – server name: WT812S0
- 1 Batch container with one servant – server name: WT812A0
- Batch container max heap was set to 1GB
- GC policy was set to “gencon”
- JDBC Type 2 connectivity was used between WCG and DB2
- Jobs were submitted using a REXX which submitted specified number of JCL's
2.2.6 Database Implementation
The batch jobs were designed to run against individual tables so that contention can be avoided. 100 tables each in its own tablespace were created. Each table had a unique index. The definitions for all the tables were the same. Before each run, the database tables were emptied.
3 Test Process

3.1 Test methodology

As mentioned earlier, the methodology was to keep as many things as possible as constants and submit as many jobs as possible to run in 1 hour and measure CPU consumption. The only variation was the number of records to be processed. The things that were kept as constants are as follows:

- The number of JVMs (that is, WAS z/OS servant regions) for the batch container was set to 1
- The JVM heap was set a max value of 1GB
- The number of threads in the servant was set to match the number of concurrent jobs per each test. We ran 5 different tests for each test case, starting with running 1 job at a time for an hour to 5 jobs running concurrently at any give point of time
- GC policy was set to “gencon”
- JVM was 31-bit
- Checkpoint interval was 3000 records
- Each job wrote to a separate table

Using the above environment two sets of five tests were run: one with the jobs processing 1 million records each and the second with 10 million records each.

Before each test the cell was stopped and restarted fresh. Jobs were submitted using REXX which submitted a JCL that executed the WSGRID program. The WSGRID program passed the xJCL to CG scheduler and the jobs were executed in the end point servers. The scheduler processed all the jobs it received and dispatched all of them to the end point server. The end point server only executed as many concurrently as defined by the test case. The servers were then brought down and the SMF records were dumped into a dataset.

The same process was repeated for the 10 million records test case..

3.2 Batch Test case 1 – million records Results

The following process was followed for running the tests:

- The Database tables were emptied before each run
- The entire cell was brought up just before submitting the batch jobs
- Batch Jobs were submitted using a REXX script
- The servers were brought down
- SMF records were dumped
- The SMF 120 sub type 20 records cut by WCG were used to measure the CPU time attributed to each job
- RMF workload activity report by service class (PRDBATHI) was used to estimate %CP, %AAP and %AAPCP

The tables below give details on the average CPU time taken by each job per test case. The information in the tables is taken from SMF 120 records and RMF reports.
Table 2  SMF 120 records summary for Test Case 1 Run 1 thru 5

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Total # of jobs in an hour</th>
<th>Avg Elapsed</th>
<th>Avg CPU Time per job</th>
<th>Avg CPU time on zAAP per job</th>
<th>%CP (does not include %APPCP)</th>
<th>%AAP</th>
<th>%AAPCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 -&gt; 1</td>
<td>59</td>
<td>61</td>
<td>55</td>
<td>41</td>
<td>22.12</td>
<td>67.57</td>
<td>1.37</td>
</tr>
<tr>
<td>2 -&gt; 2</td>
<td>84</td>
<td>86</td>
<td>58</td>
<td>35</td>
<td>31.62</td>
<td>81.63</td>
<td>20.55</td>
</tr>
<tr>
<td>3 -&gt; 3</td>
<td>99</td>
<td>111</td>
<td>59</td>
<td>32</td>
<td>38.65</td>
<td>86.33</td>
<td>33.99</td>
</tr>
<tr>
<td>4 -&gt; 4</td>
<td>100</td>
<td>146</td>
<td>61</td>
<td>33</td>
<td>40.36</td>
<td>88.79</td>
<td>38.36</td>
</tr>
<tr>
<td>5 -&gt; 5</td>
<td>95</td>
<td>194</td>
<td>66</td>
<td>34</td>
<td>42.01</td>
<td>89.71</td>
<td>40.73</td>
</tr>
</tbody>
</table>

The %AAP column indicates the percentage of CPU time spent on the one zAAP processor configured for this LPAR. The %AAPCP column indicates the percent of CPU eligible for zAAP but executed on the GP because the zAAP was fully consumed. Recall that the dedicated LPAR had one GCP and one zAAP. The %CP numbers include time spent in DB2 – since we used a JDBC type 2 connection – and the native code in WAS on z/OS.

This indicates that for this workload a second specialty engine would have been beneficial for test cases 2 through 5.

3.2.1 Observations

Figure 2  Graph shows average elapsed time/job

The above figure clearly shows that as we increase the number of jobs that run concurrently, the elapsed time increases. This makes sense, as the jobs contend for CPU.
The above figure shows that running 4 jobs concurrently gives the best performance. This allowed a maximum number of 100 million records to be processed in an hour.

3.3 Batch Test case 2 – 10 million records Results

The following process was followed for running the tests:
- The Database tables were emptied before each run
- The entire cell was brought up just before submitting the batch jobs
- Jobs were submitted using a REXX script
- The servers were brought down
- SMF records were dumped
- The SMF 120 sub type 20 records cut by WCG were used to measure the CPU time attributed to each job
- RMF workload activity report by service class (PRDBATHI) was used to estimate %CP, %AAP and %AAPCP

The tables below give details on the average CPU time taken by each job per test case. The information in the tables is taken from SMF 120 records and RMF reports.
Table 3  **SMF 120 records summary for Test Case 2 Run 1 thru 5**

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Total # of jobs in an hour</th>
<th>Average Elapsed Time per job</th>
<th>Average CPU Time per job</th>
<th>Average CPU time on zAAP per job</th>
<th>%CP (does not include APPCP)</th>
<th>%AAP</th>
<th>%AAPCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 -&gt; 1</td>
<td>5</td>
<td>766</td>
<td>699</td>
<td>520</td>
<td>22.07</td>
<td>68.18</td>
<td>1.19</td>
</tr>
<tr>
<td>2 -&gt; 2</td>
<td>8</td>
<td>963</td>
<td>631</td>
<td>397</td>
<td>31.8</td>
<td>84.37</td>
<td>14.7</td>
</tr>
<tr>
<td>3 -&gt; 3</td>
<td>9</td>
<td>1290</td>
<td>646</td>
<td>369</td>
<td>37.02</td>
<td>88.39</td>
<td>27.49</td>
</tr>
<tr>
<td>4 -&gt; 4</td>
<td>8</td>
<td>1622</td>
<td>670</td>
<td>364</td>
<td>39.81</td>
<td>90.70</td>
<td>33.91</td>
</tr>
<tr>
<td>5 -&gt; 5</td>
<td>5</td>
<td>2023</td>
<td>686</td>
<td>365</td>
<td>41.26</td>
<td>91.17</td>
<td>38.72</td>
</tr>
</tbody>
</table>

The %AAP column indicates the percentage of CPU time spent on the one zAAP processor configured for this LPAR. The %AAPCP column indicates the percent of CPU eligible for zAAP but executed on the GP because the zAAP was fully consumed. Recall that the dedicated LPAR had one GCP and one zAAP.

The %CP numbers include time spent in DB2 – since we used a JDBC type 2 connection – and the native code in WAS on z/OS.

This indicates that for this workload a second specialty engine would have been beneficial for test cases 2 through 5.

### 3.3.1 Observations

The above figure clearly shows that as we increase the number of jobs that run concurrently, the elapsed time increases. This makes sense, as the jobs contend for CPU.
The above figure shows that running 3 jobs concurrently gives the best performance. This allows maximum number of 90 million records to be processed in an hour

4 Summary

The test results provided in this document illustrate several key takeaway points:

- Java batch running in WebSphere Compute Grid is capable of executing Java batch in a very scalable manner. The test LPAR was configured with two processors – one GCP and one zAAP – and with that we saw up to 100 million records processed per hour.

- The test results illustrate the relatively high degree of specialty engine (zAAP or zAAP-on-zIIP) offload possible by running batch processing in Java on Compute Grid. The offload numbers seen in this test scenario include the batch process as well as the Compute Grid and WAS z/OS runtime infrastructure. A high degree of offload contributes to a better Total Cost of Ownership (TCO).

- The test scenarios provide a framework for understanding the performance profile of Java batch applications in a Compute Grid and WAS z/OS setting. With the system configured as shown earlier – with two processors, and the application doing file reads and DB2 z/OS updates -- the Java batch application used in this test was capable of scaling to four concurrent jobs and 100 million records processed per hour.

The results you see may be different based on a number of factors. This test was conducted in a controlled environment with specific conditions present. Your environment may be different and as a result your results may be different. The results here are not a promise of similar performance results.

Re-engineering existing COBOL batch assets to run as Java with a Compute Grid environment is feasible and efficient, as this study bears out. While the amount of CPU consumed by Java compared to COBOL is in the neighborhood of 2.5 to 3 times more, much of that is eligible for specialty engine offload. Further, re-engineering to Java allows you to take advantage of powerful Java tooling environments such as IBM Rational Applicaton Developer and center your development efforts around a common programming language (Java), a common set of tooling (RAD) and a common runtime (WAS z/OS and Compute Grid).