SkyServer Web Service

[1]Ioan Raicu, [1,2]Ian Foster

[1]*Computer Science Dept., The University of Chicago, iraicu@cs.uchicago.edu*
[2]*Math and Computer Science Div., Argonne National Laboratory, foster@mcs.anl.gov*

Abstract

The AstroPortal project [7] operates on the SDSS DR4 [6] dataset to perform a stacking operation on an arbitrary set of images from the SDSS DR4 dataset. Each image is composed of multiple objects, while each object is described by meta-data information such as the {RA DEC} coordinates, the band the image was taken in, calibration information, etc… When a user specifies a stacking to be performed, the input is a list of coordinates and bands in the form {RA DEC BAND}. This tuple needs to be converted to the actual location of the data where the image can be found. We have had a solution to this problem for quite some time, namely the SkyServer [1], but we wanted to explore some different alternatives and compare the results. We used a KDTree implementation to allow efficient search of a 2 dimensional space for both the nearest neighbor and for all objects within a rectangular range. This short report describes our motivation, implementation, interface, and performance.

1 Motivation

Originally, we used the SkyServer [1] to pose queries based on this tuple and in return, we would get back the position of the data. For example, posing the query http://cas.sdss.org/dr4/en/tools/search/x_sql.asp?format=csv&cmd=select+dbo.fGetUrlFitsCFrame(fieldId,'r')+as+r+from+dbo.fGetNearestFrameEq(180.234, 1.011, 0), we would get back http://das.sdss.org/DR4/data/imaging/752/40/corr/6/fpC-000752-r6-0245.fit.gz.

We had several motivations to seek some alternate means to do this translation:

1) the SkyServer is a shared system so the performance will vary
2) the interface into the SkyServer via an HTTP request with only 1 translation at a time seemed very inefficient
3) the production SkyServer has a limit of 60 HTTP queries per minute, much too small for utilizing the AstroPortal resources efficiently.

We are also pursuing obtaining a copy of the SkyServer and bringing it up online locally on a dedicated resource, at which time we hope that at least limitation #1 and #3 from above will be solved. There is also an online form at [2] that allows the translation of multiple tuples at the same time which would solve limitation #2, but this form has a limit of 80K input size, which limits the number of translations to about 2500 at a time. Furthermore, we need to find out how to use this online form in a systematic programming friendly way.

2 KDTree Implementations and Usage

We used a KDTree data structure to hold the SDSS meta-data information necessary to answer two types of queries: 1) nearest neighbor query and 2) range query. The meta-data information consisted of the {RA DEC} coordinates of the center of each image in the SDSS dataset along with a unique name of the image; we could envision having other meta-data information maintained in the KDTree, such as calibration information needed to be applied to the SDSS images. The unique name is similar to the actual file name as retrieved by the original SkyServer [1], but with several differences. For example, the file http://das.sdss.org/DR4/data/imaging/2662/40/corr/6/fpC-002662-u6-

0103.fit.gz as it would have been retrieved by the SkyServer would be P/2662/40/corr/6/fpC-002662-B6-0103.fit as it is stored in the internal state of the KDTree. This would allow easy substitution of a different path instead of the "P" and one of the 5 bands instead of "B".

We also used a tool called "funsky" [4] to find the centers (1024,744.5) of each image in the SDSS dataset, which was important for the initial creation of the KDTree. We experimented with the longest Euclidian distance (in the sky coordinates) any edge pixel (in the image) could have from the center of the image, and we found it to be always less than 0.14. We use this constant value to determine if the nearest neighbor found is indeed close enough to the query point. This method might produce a few false positives, but we can always check on the actual image file if the query point is within the boundaries of the actual image right before we do the stacking.

The KDTree implementation is done both in JAVA using the edu.wlu.cs.levy.CG.KDTree package [5], and in C++ using the ANN libraries [3]. The JAVA implementation has the advantage of being platform independent and easier integration into a JAVA Web Service, but at the cost of performance. The C++ implementation gives us better performance for the tuple translation, but we loose some functionality that we had in the JAVA implementation, namely the range query; note that we do not use the range query functionality in the AstroPortal, but it is likely that projects like Montage to use the range query extensively.

The KDTree implementation can be found in "SkyServerWS.0.4/kd.jar", while the usage of the KDTree implementation is in "SkyServerWS.0.4/KDTree/KDTreeIndex.java".

3 SkyServer Web Service

We used GT4.0.1 to wrap both KDTree implementations into a web service for ease of deployment and accessibility. We implemented the WS in JAVA, which made the use of the KDTree JAVA implementation trivial. For the C++ KDTree implementation, we had to use JNI to access native code (C++) from within a JAVA VM. We also implemented two sample clients in JAVA, one that performs nearest neighbor queries, and one that performs range queries. The web service interface is described in "SkyServerWS.0.4/schema/SkyServerWS/SkyServerService_instance/SkyServer.wsdl". The web service implementation can be found at "SkyServerWS.0.4/org/globus/SkyServerWS/services/core/SkyServer/impl/" in several files. The web service we have implemented is very basic, and hence only 2 files have specific code that we needed for our web service. The "SkyServerResource.java" has the initialization code to load the KDTree into memory from disk. The "SkyServerService.java" has the interface implementation, namely 3 functions that can be accessed from clients:

1) init(): must be called to ensure that the KDTree has been initialized; returns true if it has, or false if the load failed
2) query(): performs a nearest neighbor search for a specified list of {RA DEC BAND} tuples, and returns their full data paths
3) range(): performs a range query based on a rectangular specified by two {RA DEC} coordinates and a BAND; the output is a list of images with their full data paths and center coordinates in {RA DEC} format; if the center coordinates aren't needed, they can be omiteed, but we figured that an application such as Montage could use the centers of each image to figure out the order of images and how they are supposed to be stitched together.

Two sample clients have been implemented that can be found at "SkyServerWS.0.4/org/globus/SkyServerWS/clients/SkyServerService_instance/":

1) "nearestNeighborQuery.java": instantiates the necessary state in order to be able to use the SkyServer Web Service; it then reads a list of tuples in the form {RA DEC BAND} from disk, packages it up in a Query object, and invokes the query() from the web service with the Query object as a parameter; the result is a QueryResponse object, which contains an array of String objects specifying the location of the images corresponding with the query set. See the –help option for more details on the usage of the nearestNeighborQuery class.

2) “rangeQuery.java”: instantiates the necessary state in order to be able to use the SkyServer Web Service; it takes two {RA DEC} coordinates and a BAND from the command line arguments, packages it up in a Range object, and invokes the range() from the web service with the Range object as a parameter; the result is a RangeResponse object, which contains an array of RangeResult objects, where a RangeResult is defined as {String double double} where the String is the location of the images corresponding with the query set, and the two doubles are the {RA DEC} coordinates of the center of the image. See the –help option for more details on the usage of the rangeQuery class.

4 Performance

The DR4 SDSS dataset has over 1.2 million files, but with 5 bands (that share a common naming scheme), the number of entries in the KDTree was reduced to about 250K. The native C++ implementation supports nearestNeighborQueries at rates of about 10K~20K / second through the JNI interface. The JAVA implementation supports rates in the range of 4K~8K / second. Once we wrapped these two implementations with a WebService interface, these rates decreased somewhat, but are still at relatively high rates. It is worthwhile to note that the C++ implementation does not currently support range queries. Furthermore, the JAVA implementation is much cleaner and allows for much simpler usage code; at the same time, it allows for a more dynamic data structure as the logic needed to maintain the key/data pairs synchronized is trivial (due to the pair being tightly coupled) while complex and error prone logic is needed for the C++ implementation due to the fact that the key/data pair are maintained in completely separate data structures and are only connected by an integer pointer that must be manually updated when needed.

The following experiments were done from one client (multi-threaded) to a single GT4 container (also multi-threaded). Each machine had dual 2.4GHz Xeon CPUs with 4GB RAM and 1Gb/s network connectivity over a LAN.

In the following few figures, we use the following conventions. For the query Performance, the x-axis shows the batch size for each query as it varied from 1 to 128K in one run; each run used only 1 web service call per thread; for the range performance, the x-axis has the number of results retrieved for varying range sizes from 1 square degrees to 36864 square degrees. The y-axis on the right shows the number of threads used to get the observed throughput and latency (y-axis on the left); it should be noted that the number of queries per service call can be computed by taking the batch size and dividing by the number of threads used. All the axes are in logarithmic scale for better visualization of the results.

Figure 1 shows the query performance of the Sky Server Web Service implemented in JAVA but using the C++ KDTree implementation. We see the query throughput monotonically increase as the batch size increases; note that the number of parallel threads needed to get improved throughput with the larger batch sizes also increases. For a fixed number of threads, the throughput performance is not monotonically increasing with the batch size; the reason is that for small batch sizes, there is no point to incur extra overhead by having multiple threads as well as having multiple web service calls; on the other hand, for large batch sizes, the fewer the number of threads, the larger number of queries per web service call, and there seems to be performance degradation when 1 web service call needs to transfer large number of queries and get a response for a large number of results.

GT4 web services uses HTTP over SOAP and XML as the underlying technologies. That means that if we send 100K queries (of the form {double double char} = 9 bytes) in 1 web service call and we get back 100K results (of the form {string} ~ 50 bytes), we have about 6MB of data that needs to be serialized into XML and deserialezed from XML, which apparently is relatively expensive to do. The good part is that the cost per query/result decreases with more queries/results bunched together in a single web service call up to a certain point, after which the costs start to increase again with larger sets of queries/results. By varying the number of threads, we were able to keep the performance improving with an increasing batch size.

The best performance is obtained with batch sizes of 10K+ and 32 ~ 64 parallel threads reaching about 5K queries per second and a latency of about 200 us per query. For example, to perform a batch query of 16K queries, it would take about 2.8 seconds to complete; out of this 2.8 seconds, about 1 second would be spent querying the KDTree and getting back the results in memory, so the other 1.8 seconds is spent in sending the queries from the client to the SkyServer web service and getting back the results back to the client.

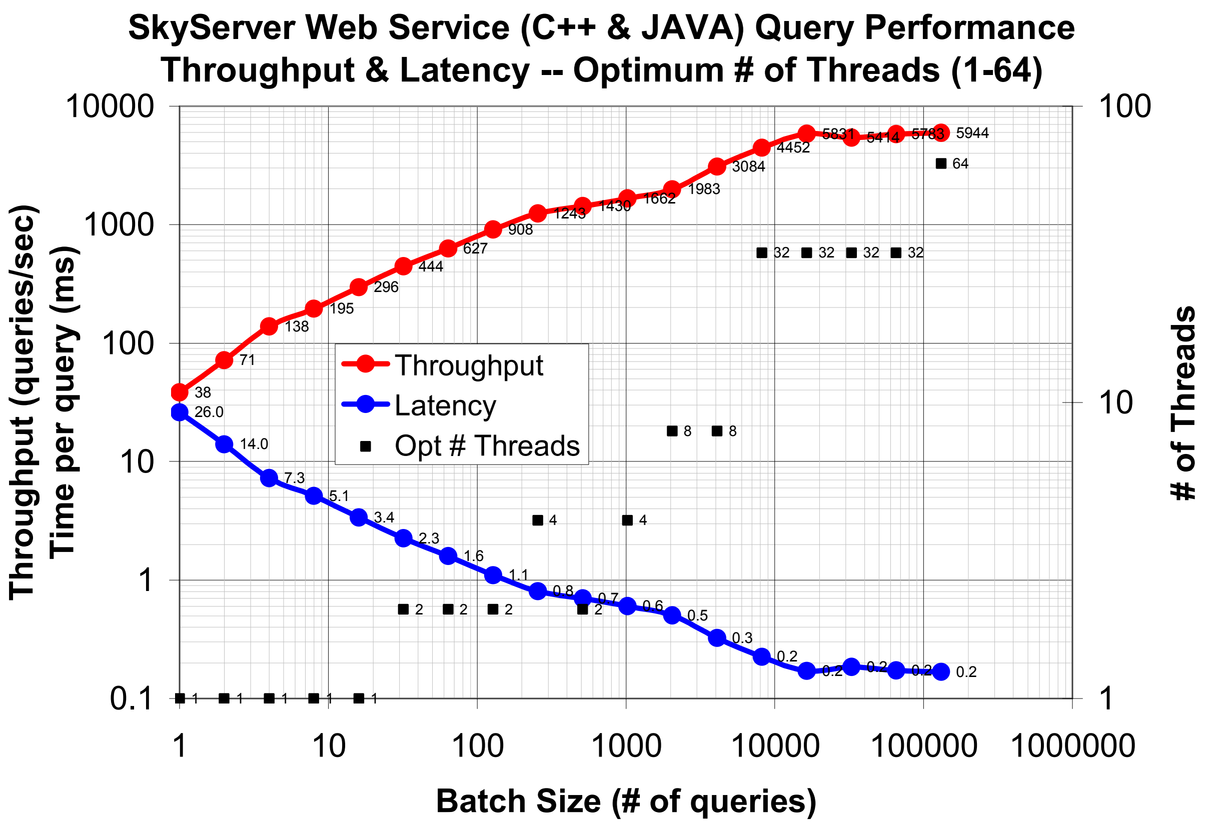

Figure 1: Query performance of the Sky Server Web Service implemented in JAVA but using the C++ KDTree implementation

Since the C++ implementation of the KDTree does not support range queries, we could perform the range query performance experiment as we did for the JAVA implementation.

Figure 2 shows the query performance of the Sky Server Web Service implemented in JAVA and the JAVA KDTree implementation. We see the query throughput monotonically increase as the batch size increases, similar to the performance of the C++ implementation. The performance is lower (almost half for the larger batch sizes), but it is probably still good enough for use in the AstroPortal to not be a bottleneck.

The best performance is obtained with batch sizes of 10K+ and 32 ~ 64 parallel threads reaching well in the middle 3K queries per second and a latency of about 300 us per query. For example, to perform a batch query of 16K queries, it would take about 5 seconds to complete; out of this 5 seconds, about 3.2 seconds would be spent querying the KDTree and getting back the results in memory, so the other 1.8 seconds is spent in sending the queries from the client to the SkyServer web service and getting back the results back to the client.

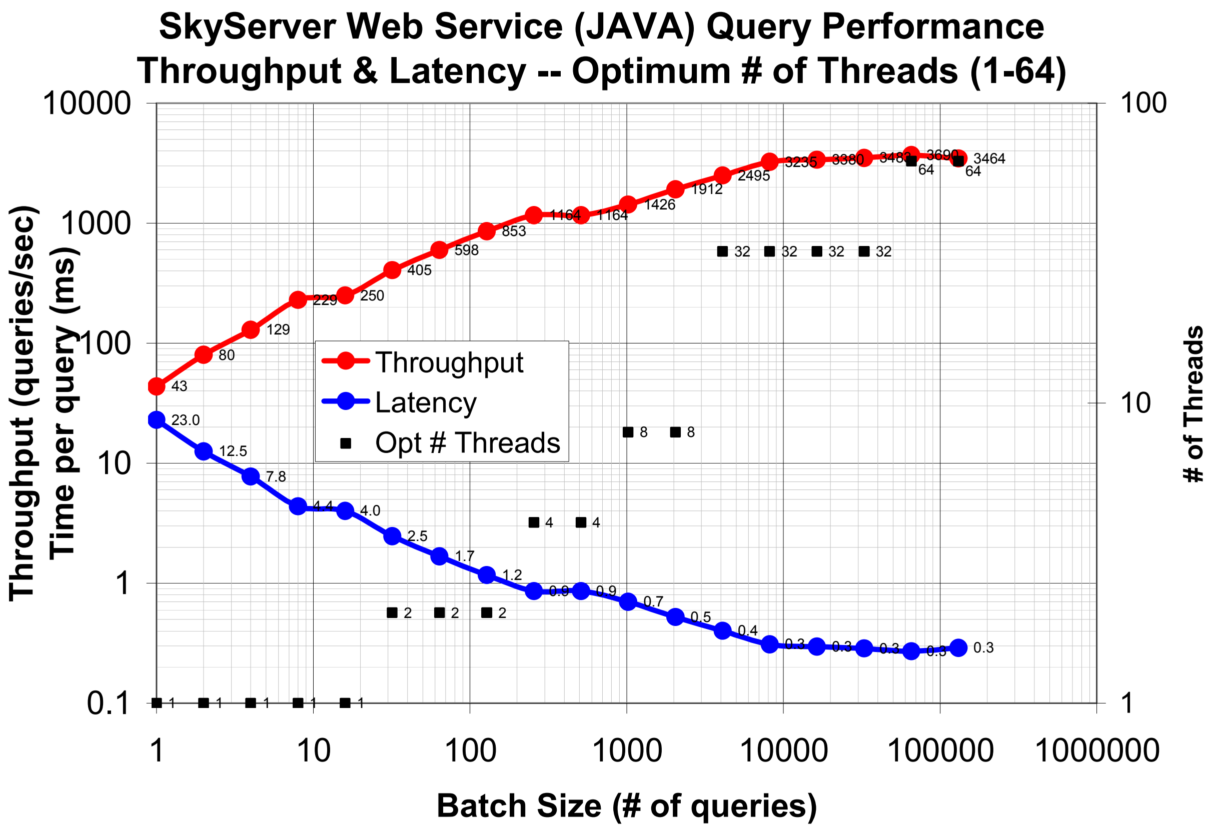

Figure 2: Query performance of the Sky Server Web Service implemented in JAVA and using the JAVA KDTree implementation

Figure 3 shows the range query performance, as it increases the range size from 1 x 1 degree (1 square degree) to 192 x 192 degrees (36864 square degrees). Since the images are not uniformly distributed over the sky (only a quarter of the sky is currently available in the DR4 dataset), the number of images found as the range size increased is not necessarily linear. For example, for a range of 32x32 (1024 square degrees) we found 3655 results, for 64x64 (4096 square degrees) we found 4916 results, and for 128x128 (16384 square degrees) we found 30662 results. Notice how we only found 30% more results in the 64x64 range over the 32x32 range despite the 400% larger surface area. Then, going up to 128x128 range, we found over 500% more results, although the range size only increased again 400%. Since the range size would not have been a good indicator to the actual performance of the range query, we decided to plot the results against the number of results retrieved, as that is the biggest indicator of the performance.

The performance seems to be better than for the simple query from Figure 2, especially for larger number of results retrieved, reaching over 5.6K results per second using 64 parallel threads. For example, to perform a range query of 128x128 which retrieves over 30K results, it would take about 6.5 seconds to perform the range query.

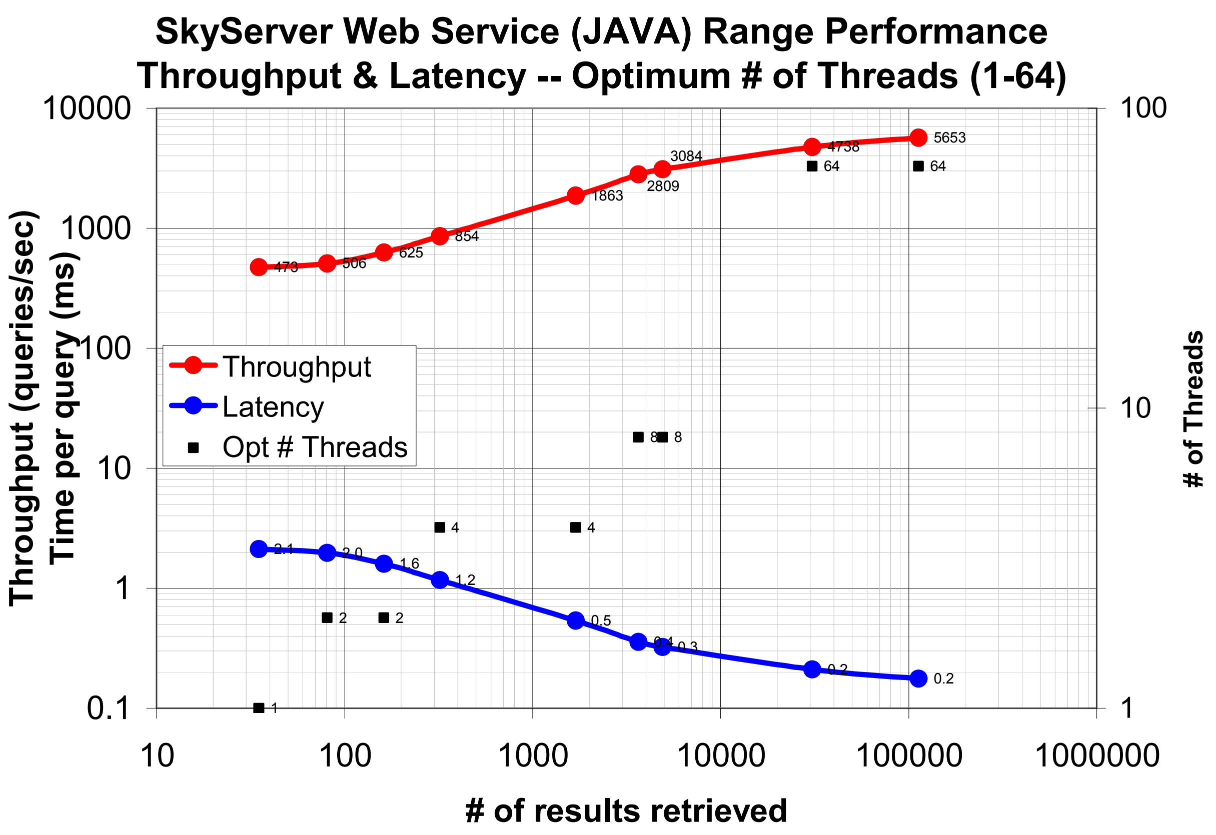

Figure 3: Range performance of the Sky Server Web Service implemented in JAVA and using the JAVA KDTree implementation

5 Project Files and Pre-requisite Software

5.1 SkyServerWS

For completeness, we discuss the important files in the SkyServerWS archive. The archive can be downloaded from "*http://people.cs.uchicago.edu/~iraicu/research/AstroPortal/SkyServerWS/SkyServerWS.0.4.tgz*". To download it, type "*wget http://people.cs.uchicago.edu/~iraicu/research/AstroPortal/SkyServerWS/SkyServerWS.0.4.tgz*"; to decompress it, type "*tar xvfz SkyServerWS.0.4.tgz*". The SkyServerWS requires the Globus Toolkit 4 to be installed and functioning properly.

The full directory listing of each file in the SkyServer web service implementation is:

- *SkyServerWS.0.4/build.mappings*
- *SkyServerWS.0.4/build.xml*
- *SkyServerWS.0.4/globus-build-service.sh*: script used to compile the SkyServer Web Service
- *SkyServerWS.0.4/kd.jar*: KDTree implementation in JAVA
- *SkyServerWS.0.4/namespace2package.mappings*
- *SkyServerWS.0.4/index-SAMPLE.txt*: sample index file with 35 entries to populate the KDTree Index
- *SkyServerWS.0.4/index-SDSS-DR4.txt*: the full SDSS DR4 index file which contains about 250K entries to populate the KDTree Index
- *SkyServerWS.0.4/queries.txt*: sample query file with 35 queries of which only 31 queries should return a result when using the index-SAMPLE.txt file to populate the KDTree Index
- *SkyServerWS.0.4/readme.SkyServerWS.pdf*: a copy of this document

- *SkyServerWS.0.4/KDTree/KDTreeIndex.java*: supporting code that uses the KDTree package
- *SkyServerWS.0.4/org/globus/SkyServerWS/clients/SkyServerService_instance/nearestNeighborQuery.java*: client that accesses the SkyServerWS and performs nearest neighbor queries
- *SkyServerWS.0.4/org/globus/SkyServerWS/clients/SkyServerService_instance/rangeQuery.java*: client that accesses the SkyServerWS and performs range queries
- *SkyServerWS.0.4/org/globus/SkyServerWS/services/core/SkyServer/deploy-jndi-config.xml*
- *SkyServerWS.0.4/org/globus/SkyServerWS/services/core/SkyServer/deploy-server.wsdd*
- *SkyServerWS.0.4/org/globus/SkyServerWS/services/core/SkyServer/impl/SkyServerConstants.java*
- *SkyServerWS.0.4/org/globus/SkyServerWS/services/core/SkyServer/impl/SkyServerResource.java*
- *SkyServerWS.0.4/org/globus/SkyServerWS/services/core/SkyServer/impl/SkyServerResourceHome.java*
- *SkyServerWS.0.4/org/globus/SkyServerWS/services/core/SkyServer/impl/SkyServerService.java*: SkyServer web service implementation
- *SkyServerWS.0.4/schema/SkyServerWS/SkyServerService_instance/SkyServer.wsdl*: SkyServer web service interface
- *SkyServerWS.0.4/scripts/make.SkyServerWS.client.sh*: script to compile the client code
- *SkyServerWS.0.4/scripts/make.SkyServerWS.ws.sh*: script to compile and deploy the SkyServer Web Service
- *SkyServerWS.0.4/scripts/run.SkyServerWS.client.sh*: script to run the client code
- *SkyServerWS.0.4/scripts/run.SkyServerWS.ws.local.sh*: script to run the SkyServer Web Service code on the local node, which really starts the GT4.0.1 container
- *SkyServerWS.0.4/scripts/run.SkyServerWS.ws.sh*: script to copy the GT4 container from another machine and start the GT4 container in order to run the SkyServer Web Service code
- *SkyServerWS.0.4/scripts/cleanup.all.sh*: script to clean up any compiled code from the SkyServerWS.0.4

Note that the last 5 scripts used to compile and run the web service and the client code are very specific to the environment we have at ANL on the TeraGrid. Some of the paths would have to be updated with the specific paths where GT4 is installed. Also, note that when running the client code on a different machine than the one where it was compiled, it might not be sufficient to copy the *.class of the client code, but you might also need some *.jar from the GT4/lib and the full array of GT variables to be set (i.e. GLOBUS_LOCATION, LD_LIBRARY_PATH, et…). Ideally, these scripts should be made more generic, but we did not have time this time around; for a more generic and easier to setup archive, see "*http://people.cs.uchicago.edu/~iraicu/research/AstroPortal/SkyServerWS/SkyClientWS.0.5.tgz"* which contains just the client code, but is very generic and requires just the Java environment to be configured properly; for more information, see the next section on the SkyClientWS.0.5. To compile the SkyServer Web Service and deploy it in the GT4 container, run the script "*SkyServerWS.0.4/scripts/make.SkyServerWS.ws.sh*". To compile the two sample clients, run the script "*SkyServerWS.0.4/scripts/make.SkyServerWS.client.sh*". To run the GT4 container (which will also start the SkyServer Web Service), run the command "*globus-start-container -nosec -p 50001*", where –p 50001 indicates the port you want to run the container on. The script "*SkyServerWS.0.4/scripts/run.SkyServerWS.ws.local.sh*" also starts up the GT4 container after it sets some environment variables (which would have to be updated with the particular environment the GT4 container will be started in). To enable DEBUG statements in the SkyServer Web Service, you have to add the following line to the "*${GLOBUS_LOCATION}/container-log4j.properties*": "*log4j.logger.org.globus.SkyServerWS.services.core.SkyServer.impl=DEBUG*"; similarly, if you just want some basic information without the full debug statements, you would use "*log4j.logger.org.globus.SkyServerWS.services.core.SkyServer.impl=DEBUG*". To run the sample client codes, run the script "*SkyServerWS.0.4/scripts/run.SkyServerWS.client.sh*" which will set up the environment variables (which need to be updated accordingly) and run the two clients one after another. Please see the –help option or more information regarding the different command line arguments to be used with these two clients.

Our environment had the following software which was used to compile and run the SkyServerWS and the client codes:

- *Globus Toolkit 4.0.1 (GT4.0.1)*
- *Apache Ant version 1.6.5*
- *JDK 1.5.0*

It is very likely that the latest version of the Globus Toolkit (GT4.0.2), slightly different variations of Ant, and JDK 1.4.2 should work without any modifications.

5.2 *SkyClientWS*

For ease of use, we have packaged the client source code along with all the supporting code. To compile and run the client code, there is no need for the Globus Toolkit 4 to be installed, as was the case for the SkyServerWS. However, the one prerequisite is to have Java JDK 1.5 (builds 1.5.0_06-b05 and later are sure to work). Normally, Java 1.4 would have been enough, however, GT4 was compiled using Java 1.5, and there are some GT4 dependencies (*.jar) included in the archive which the client code uses.

Since the client code is useless without the service code, we have set up a machine at University of Chicago which has the SkyServerWS.0.5 running. The GT4 container which has the SkyServerWS running can be found at viper.diperf.cs.uchicago.edu:50001. The full URI of the SkyServerWS is http://viper.diperf.cs.uchicago.edu:50001/wsrf/services/SkyServerWS/core/SkyServer/SkyServerService. The machine specifications are:

- **Name**: viper.diperf.cs.uchicago.edu
- **IP**: 128.135.164.107
- **CPU**: Dual Intel(R) Xeon(TM), 3.00GHz, HT, 1MBx2 cache
- **Memory**: 2GB PC2700 ECC Reg
- **Hard Drive**: 80GB PATA Seagate Baracuda 7200 RPM
- **Network Card**: 1Gb/s NIC
- **Network Connectivity**: 100Mb/s
- **OS**: SuSe Linux 10, Kernel 2.6.13-15-smp

For completeness, we discuss the important files in the SkyClientWS archive. The archive can be downloaded from "*http://people.cs.uchicago.edu/~iraicu/research/AstroPortal/SkyServerWS/SkyClientWS.0.5.tgz*". To download it, type "*wget http://people.cs.uchicago.edu/~iraicu/research/AstroPortal/SkyServerWS/SkyClientWS.0.5.tgz*"; to decompress it, type "*tar xvfz SkyClientWS.0.5.tgz*".

The full directory listing of each file in the SkyClientWS implementation is:

- *SkyClientWS.0.5/org/globus/SkyServerWS/clients/SkyServerService_instance/nearestNeighborQuery.java*: client that accesses the SkyServerWS and performs nearest neighbor queries
- *SkyClientWS.0.5/org/globus/SkyServerWS/clients/SkyServerService_instance/rangeQuery.java*: client that accesses the SkyServerWS and performs range queries
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/bindings/SkyServerPortTypeSOAPBindingStub.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/Range.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/Result.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/service/SkyServerServiceAddressingLocator.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/service/SkyServerServiceAddressing.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/service/SkyServerService.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/service/SkyServerServiceLocator.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/RangeResponse.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/QueryResponse.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/Init.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/RangeResult.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/SkyServerPortType.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/Tuple.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/Query.class*
- *SkyClientWS.0.5/org/globus/SkyServerWS/stubs/SkyServerService_instance/InitResponse.class*
- *SkyClientWS.0.5/readme.SkyServerWS.pdf*: a copy of this document
- *SkyClientWS.0.5/readme.SkyClientWS.txt*: a copy of this section 5.2 from this document
- *SkyClientWS.0.5/lib/addressing-1.0.jar*
- *SkyClientWS.0.5/lib/axis-url.jar*
- *SkyClientWS.0.5/lib/axis.jar*

- *SkyClientWS.0.5/lib/jaxrpc.jar*
- *SkyClientWS.0.5/lib/saaj.jar*
- *SkyClientWS.0.5/lib/cog-jglobus.jar*
- *SkyClientWS.0.5/lib/wsdl4j.jar*
- *SkyClientWS.0.5/lib/log4j-1.2.8.jar*
- *SkyClientWS.0.5/lib/commons-logging.jar*
- *SkyClientWS.0.5/lib/commons-discovery.jar*
- *SkyClientWS.0.5/run.SkyClientWS.sh*: script to run the client code
- *SkyClientWS.0.5/make.SkyClientWS.sh*: script to compile the client code
- *SkyClientWS.0.5/queries.txt*: sample query file with 35 queries of which only 31 queries should return a result when using the index-SAMPLE.txt file to populate the KDTree Index
- *SkyClientWS.0.5/schema/SkyServerWS/SkyServerService_instance/SkyServer.wsdl*: SkyServer web service interface
- *SkyClientWS.0.5/etc/globus-devel-env.sh*

Note that the last 2 scripts used to compile and run the client code are meant to be very general and should work unmodified as long as it is within a Linux environment and Java 1.5 is used. To compile the two sample clients, run the script "*SkyClientWS.0.5/make.SkyClientWS.sh*". To run the sample client codes, run the script "*SkyClientWS.0.5/run.SkyClientWS.sh*" which will set up the environment variables and run the two clients one after another. Please see the –help option or more information regarding the different command line arguments to be used with these two clients.

Our environment had the following software which was used to compile and run the SkyClientWS:

- *JDK 1.5.0*

6 References

[1] Sloan Digital Sky Survey / SkyServer, http://cas.sdss.org/astro/en/

[2] SDSS DR4 Tools, http://cas.sdss.org/dr4/en/tools/crossid/upload.asp

[3] KDTree Implementation in C++, http://www.cs.umd.edu/~mount/ANN/

[4] FunTools, http://hea-www.harvard.edu/RD/funtools/

[5] KDTree Implementation in JAVA, http://www.cs.wlu.edu/~levy/kd/

[6] SDSS Data Release 4 (DR4), http://www.sdss.org/dr4/

[7] Ioan Raicu, Ian Foster, Alex Szalay, Gabriela Turcu. "AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis", to appear at TeraGrid Conference 2006, June 2006.