This brief description assumes the reader is familiar with the use of Linux bash command lines and Linux pathnames, has some idea of the use of Docker™ and/or Singularity™ containers, and is familiar with basic Linux virtual machine concepts.

Lancium Compute Infrastructure (LCI) - Execution and Data Model

The Lancium Compute Infrastructure is designed for high-throughput <link to our HTC page> applications. Examples include molecular dynamics, scene rendering for movies or commercials, hyper-parameter space tuning for ML training, ensemble calculations, Monte Carlo simulations, parameter space exploration in engineering design, and many other applications.

Lancium offers three services:

- Lancium **Remote Command Line Services** provide a Linux command line service using Singularity containers that allows users to run thousands and thousands of computations remotely on LCI resources.
- Lancium **Virtual Machine Services** provides customers the ability to define, instantiate, and manage Linux KVM virtual machines on LCI resources.
- Lancium **Data Services** provide customers with a global-scale distributed file system in which data stored at Lancium sites, customer sites, and customer-partner sites can be accessed securely from anywhere with a network connection.

We begin with a description of the Lancium Compute Infrastructure, and then describe the three services. For detailed descriptions see the associated documentation and tutorials. (link to gcli docs).

Lancium Compute Infrastructure

The LCI consists of a number of *sites*, each site having a number of *subClusters*, and each subCluster consisting of a set of compute *nodes*, *a head node*, and a *subCluster* file server. Each subCluster head node has at least 20Gbs Ethernet, manages its nodes, and provides file services to the nodes. A typical subCluster consists of 2-4 racks and 1000-2000 physical cores.

The compute nodes are *compute-only* nodes or *GPU-nodes*. All compute nodes have at least 4GB of memory per physical core and at least 10Gbs Ethernet. The compute-only nodes have from 16 to 36 physical cores each. The processors are a mix of Intel 2697v2, 2680v2, 2670v3, 2698v3, or 2680v4.

The GPU nodes have either K40 or K80 GPUs and have between 16 and 28 physical cores each. The GPU nodes have either two K40s, two K80 cards (each K80 has two GPUs), or eight K80 cards (32 GPUs). From the K80 specification, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/TeslaK80-datasheet.pdf K40s have a peak single precision performance of 4.29TF and each K80 GPU has a peak of 2.8TF. For more information on resource prices and comparisons with AWS and Google Cloud see:<link>.

TECHNICAL SPECIFICATIONS	Tesla K40	Tesla K80[1]
Peak double-precision floating point performance (board)	1.43 Tflops	1.87 Tflops
Peak single-precision floating point performance (board)	4.29 Tflops	5.6 Tflops
GPU	1 x GK110B	2 x GK210
CUDA cores	2,880	4,992
Memory size per board (GDDR5)	12 GB	24 GB
Memory bandwidth for board (ECC off)[2]	288 Gbytes/sec	480 Gbytes/sec
Architecture features	SMX, Dynamic Parallelism, Hyper-Q	
System	Servers and workstations	Servers

[1] Tesla K80 specifications are shown as aggregate of two GPUs.
[2] With ECC on, 6.25% of the GPU memory is used for ECC bits. For example, 6 GB total memory yields 5.625 GB of user available memory with ECC on.

LCI Remote Command Line Service (RCLS)

The RCLS allows users to execute a Linux command line on a remote LCI resource inside of a user-defined Singularity container or a Lancium provided Singularity container. The user specifies the Singularity[1] container image to instantiate, the command line to run in the container, the resource requirements (how much memory, how many cores, type and number of GPU), the maximum time the application may run, and the set of input and output files. We call this a job. Note that "cores" refers to vCPUs, also frequently called "hardware threads".

A list of Lancium provided Singularity container images can be found here (https://portal.lancium.com/about/resources).

Jobs can be single core jobs, multi-threaded multi-core jobs, single node MPI jobs[2], GPU jobs, or some combination of the above. Each job has a jobID. When submitted from the command line for execution, the user receives the jobID for the job.

Jobs can be created (and submitted), tracked (get status), and managed (suspend, kill) using their jobID and the Lancium command line tools or Lancium Web Interface (https://portal.lancium.com/).

Once submitted for execution, the *LCI* selects a compute resource that meets or exceeds[3] the user's request. The *LCI* then moves input data as needed to file systems local to where the execution will occur, mounts requested file systems and user-defined caches, and starts the container running the command line.

---

[1] We use Singularity containers at execution time. When necessary we automatically convert Docker containers to Singularity containers.
[2] Multi-node MPI will be available if there is demand.
[3] When the resource provided exceeds the users request, the user is charged for what is requested, not what is delivered.

Command line execution is in a *bash* shell inside the container in a Job Working Directory (JWD) into which all of the job data has been copied or mounted. Processes running inside of the container can perform all of the usual file system operations on data in their JWD. Process running inside the container can also access external data resources[4] using the Lancium Data Services, *wget* or similar tools.

The example below illustrates how a job is created and submitted.

```
gcli startContainers Lancium/ubuntu18.04.simg myJob 1 --cores 4 \

       -- memory 8GB --i inp-data --o myoutput "myProgram inp-data myoutput"
```

The above command line will return a jobID and cause a new instance of the container image *myimage* to be created on a resource and given 4 cores and 8GB of memory. The local file *inp-data* will be copied into the job working directory (JWD), and the command *`myProgram inp-data myoutput`* will be executed in the container. Upon completion, the file *myoutput* will be copied back to the user's local machine.

Of course, the command to start jobs can be called thousands of times in a loop to process a large amount of data. For example, suppose that you had a set of image files in a source directory to process by *`myProgram`*.

```
#!/bin/sh
if [ $# -ne 2 ]
then
        echo "USAGE:  $0 <input-dir> <output-dir>"
        exit 1
fi
for IMAGE in `ls $1/*.ppm`
do
        IMAGE=`basename $IMAGE`
        if [ ! -e $2/${IMAGE}.out ]
        then
           gcli startContainers myimage.simg filter.$IMAGE 1 --cores 4 --memory \
             8GB --i $1/$IMAGE --o $2/$IMAGE.out "myProgram ${IMAGE} ${IMAGE}.out"
        fi
done
```

The above simple shell script checks all files in the *input-dir* to determine if they have already been processed, and the results placed in *output-dir*. Those that have not been processed are sent to the LCI for processing. This is classic High Throughput Computing. https://portal.lancium.com/about/HTC.

[4] Jobs may open TCP/IP and UDP sockets to routable, public IP addresses. External resources cannot open incoming connections to jobs. Ability to listen for socket-connections from external sources will be available in the future.

LCI Virtual Machine Service (VMS)

Lancium VMS supports the definition, instantiation, and management of Linux KVM virtual machines on Linux using qcow2 images. Users begin by downloading a qcow2 base image from our image repository (If you do not see a base image you need, please let us know.) Alternatively, one can use one of our predefined images directly.

Virtual machines are started using the gcli startVMs command. The user can specify the VM image to use, the job name and host-name for the VM, the number of instances to instantiate, the number of cores (vCPUs), the total amount of memory for each instance, and an optional *cloud-init user-data file*. For example,

```
gcli startVMs Ubuntu-18.04.qcow2 myVM 1 --cores 4 --memory 8GB \
      --user-data my-startup
```

This command will start a single VM job with the job name and hostname "myVM", using the Ubuntu-18.04.qcow2 image, with 4 cores, 8 GB of memory, and run the script "my-startup" after the network initialization on the VM has been performed. The command will return a ticket.

If you want a vector of virtual machines of say five virtual machines, change the "1" above to "5". The

```
gcli startVMs Ubuntu-18.04.qcow2 myVM 5 --cores 4 --memory 8GB \
      --user-data my-startup
```

job names and VMs will be "myVM_1", "myVM_2", .. , "myVM_5".

For more information, see the tutorials and documentation https://lancium.github.io/lancium_lci/gcli.html.

Lancium Data Services

The central feature of the Lancium data model is the Global Federated File System (GFFS) global directory system[5]. All resources can be named with path names, e.g., */home/CCC/Lancium/grimshaw/stored-data/myfile* for a file in grimshaw's Lancium home directory, or */home/CCC/Lancium/grimshaw/jobs/254908/working-dir/stdout* for the *stdout* of job *254908.*

All access to the GFFS is secure. All data transport is encrypted to ensure data integrity and confidentiality. User and group authentication is achieved via cryptographically strong authentication using standard signed SAML attribute assertions. Access control is via access control lists controlled by the data owner.

Each user (and organization) has a home directory in the GFFS, e.g., */home/CCC/Lancium/grimshaw* above. The home directories live on Lancium storage. In addition to Lancium created directories such as

[5] Grimshaw, M. Morgan, A. Kalyanaraman, "GFFS – The XSEDE Global Federated File System", Parallel Processing Letters, Vol. 23, No. 02.

the *jobs* directory above, the user can manage their home directory by: i) adding and removing directories; ii) adding and removing files; or iii) linking to directories exported from their own infrastructure. Similarly, the user can read and write files. All of these typical file operations are available via a variety of mechanisms.

The Images Directory. Each user has an *Images* directory in their home directory. The *Images* directory contains the Singularity container images and qcow2 images that they have chosen to upload. To run a container or a VM there must exist an image in either their Images directory or in the public Lancium Images directory.

GFFS Exports. Users and organizations may choose to securely *export* selected directory trees in their infrastructure into the GFFS. Once mapped into the GFFS, files and directories in the exported directory tree may be created, read, updated, and deleted *subject to access control*. Data owners control who can perform what operations on their data, not Lancium.

GFFS Access. Data in the GFFS can be accessed and manipulated via one of four mechanisms: via an API, via a CLI, via a GUI or web portal, and via a Linux FUSE mount. The API, CLI, and GUI/portal are self-explanatory. The Linux FUSE mount allows users to directly mount sub-trees of the GFFS directly into their own Linux file system, providing direct, transparent, and secure access to data in the GFFS from Linux programs, shell scripts, etc. on their system.

Data from a job's perspective. From the running job's perspective there are three types of storage: the job working directory (JWD), *directory caches*, and directories within the GFFS they mount for the job. The JWD and directory caches are on NFS mounted storage in the sub-cluster in which the job is running. (There is a sub-cluster for every 2-4 thousand cores).

The JWD. The JWD is a directory created for each running job that is temporary file system space for the job. Data may be staged (copied) into the job prior to execution of the job, and can be staged out of the JWD post job execution. Data can be staged in/out from locations in the GFFS, or using other protocols such as FTP/sFTP, scp, and http(s). During program execution, the program can copy additional information into the JWD, e.g. using *wget*, and read/write temporary files. The user can mount portions of the GFFS into the JWD, and read/write files and directories in the GFFS subject to access control.

Post execution the user can specify data to be staged out of the JWD. Once the job is complete, and the required data is staged out, the JWD is deleted.

A Directory Cache (available Q3 2020). A *directory cache* is a sub-tree in the GFFS that the user wants *READONLY* cached in the local file system of the sub-cluster. The user specifies a directory path in the GFFS directory structure to be cached using the command line option *--inputCache:<GFFS path>*. The LCI mounts the directory cache in JWD to a directory that is the *basename* of the pathname in the GFFS. Prior to job start, the system determines whether a copy of the specified directory tree has been cached locally. If there is not a local copy, the system makes a read-only copy locally and links the local cached copy into the JWD. If there is already a copy, the system links it in. Many jobs may safely use the same copy of a cache at the same time.

The directory cache is a read only data space loaded by the system when a job uses it for the first time in a sub-cluster. Typically, a directory cache is loaded with data that is both large and constant across many runs of the application. For example, a scene file for rendering, a large read-only database, or executable, scripts and libraries. The objective is to avoid repeated downloads of the same data and the time and customer bandwidth associated with the downloads. Each sub-cluster will get a copy of the directory cache on first use in the sub-cluster. It may get the data from another sub-cluster at the site if it is available.