Lunar Terrain Coverage Analysis Data Delivery Workflow

Carlyn Lee* and Charles H. Lee†
NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena, US-CA

Mohsin A. Shaikh‡
KAUST Core Labs, Thuwal, KSA

Dominik L. Michels§
KAUST Visual Computing Center, Thuwal, KSA

In this work, we are developing a lunar terrain database to enable fast rendering of sun illumination and earth visibility for a proposed coverage analysis tool. This development will advance lunar mission design and formulation for current and future communications architectures, and will aid in lunar surface mission planning and communications/navigation operations. Our effort can be described in three steps: (1) we parallelize a brute force algorithm, which computes elevation masks from laser altimetry data acquired by the Lunar Reconnaissance Orbiter's (LRO) Lunar Orbiter Laser Altimeter (LOLA); (2) we investigate parallel I/O methods to store terrain mask information from step (1) into a parallel file system; and (3) we finally deliver data to the terrain coverage analysis tool.

I. Introduction

This manuscript describes the architecture of a Lunar Terrain Database developed to aid in Lunar coverage analysis and terrain visualization. This effort improves on the architecture for computing terrain masks in a sub-region of the Lunar South Pole described in [1] from laser altimetry data acquired by the Lunar Reconnaissance Orbiter's (LRO) Lunar Orbiter Laser Altimeter (LOLA) curated in [2]. In this architecture, we focus on optimizing I/O methods for storage of terrain elevation masks. These data will enable a terrain coverage analysis tool to quickly render Sun/Earth visibility from surface assets on the Moon.

The initial architecture was leveraged by Dask, a parallel computing python library which scaled our brute force algorithm to use multiple nodes in a high performance computing environment. Multiple compute nodes of Shaheen II, a Cray XC40 Supercomputer at KAUST Supercomputing Laboratory, were used to run the Dask jobs. Shaheen II provided the capacity to schedule and run these independent jobs concurrently therefore increasing the throughput. Shaheen II compute nodes are dual socket Intel Haswell nodes (6174 nodes), with 32 cores and 128GB of RAM, and are connected via high speed Cray Aries interconnect. Shaheen II uses a Simple Linux Utility for Resource Management (SLURM) for scheduling jobs. A job refers to a set of scripted steps of a workflow (a python script in our case) submitted to SLURM, which schedules and runs on requested computational resources on user's behalf.

In [1] a job array was launched using 2 nodes and 16 cores, and in 24 hours computed az-el masks for an area of 34 sq km (851x1001 pixels). The limitations to this approach arose as the number of files and scheduled jobs increased significantly toward finer resolution (i.e. number of pixels from 851x1001 to 30400x30400). The growing number of concurrent running tasks using Dask Futures was prohibitive to processing a significant region of the entire map. Initially, we attempted to process smaller sub-regions and increase the number of jobs to work within constraints posed by Dask.

In the improved architecture described here, we implement a ground up solution to treat each sub-region as independent task mapped on one of the workers. We used mpi4py, a python package which provides bindings for a well known Message Passing Interface (MPI) standard to communicate between multiple processes on the same or different machines/compute nodes [3]. Workers start synchronously, computing terrain masks for different regions, and then offload storage to dedicated write processes. Data is stored in Shaheen II's parallel file system as binary Hierarchical Data Format (HDF5), resulting in one file per job. HDF5 is compatible with most analytics platforms, like MatLab, and

*Engineering Applications Software Engineer, Telecommunications Architectures, Mail Stop 238–420.
†Telecommunications Engineer, Telecommunications Architectures, Mail Stop 238–420.
‡Computational Scientist, Supercomputing Lab.
§Principal Investigator, Computational Sciences Group.

leverages I/O methods in our approach to take advantage of Shaheen II's distributed parallel Lustre File System. This approach enables flexibility to scale a single job from one to many Shaheen II compute nodes, reduces the resulting number of files and parallel I/O methods to store into Lustre File System, and enables on-demand delivery of data to the terrain coverage analysis tool.

II. Procedure

This work expands on computation of terrain masks for a sub-region of the Lunar South Pole to describe the data storage and delivery methods. The formulation for our brute force algorithm, calc_mask, is described in [1] and is applied to data collected from [2] at -80 degrees to the Lunar South Pole. The data are summarized in figure 1, where each pixel represents the polar stereographic projection at -80 degrees to the Lunar South Pole. The pseudocode for calc_mask shown in Algorithm 1.

Algorithm 1 calc_mask takes as input LOLA image and the x & y coordinate map scaled by 20m

```
LAT_P, LON_P = cart2sph(idx_x, idx_y)
ALT_P = getAltitude(LAT_P, LON_P,img)
#for the Moon we use a radius = 1737400m
P_x, P_y, P_z = sph2cart(LON_P * pi/180, LAT_P * pi/180, radius + ALT_P)
P_hat=P/norm(P)
P_E= N × P_hat
P_E = P_E / norm(P_E)
P_N = P_hat × P_E
P_N = P_N / norm(P_N)
d =  [20, 40, 60, 80, .... 1500 meters]
ud = ones(len(d))
for each azimuth from 0 to 360:
    Q=ud·P + d · sin(deg2rad(azimuth)) * P_N + d · cos(deg2rad(azimuth)) * P_E
    LAT_Q LON_Q= [cart2sph(q_x, q_y, q_z) for all q in Q]
    for lat_q,lon_q in (LAT_Q,LON_Q):
        ALT_Q.append(getAltitude(lat_q * 180/pi, lon_q * 180/pi,img) )
    Q=[sph2cart(lon_q,lat_q,radius + alt_q) for all in (LAT_Q,LON_Q,ALT_Q)]
    for heights 2meters and 10meters:
        P = P_hat * (radius + ALT_P + h_P)
        PQ = Q-ud · P
        PQ = PQ / sqrt(sum(PQ*PQ)) · [ 1,  1,  1]
        Beta = rad2deg( arccos(sum(PQ * ( ud · P_hat)) ) )
        observation_dist= argmax(90-Beta)
        mask = 90 - Beta[int(observation_dist) ]
```

Each pixel in the image data from fig.1 represents a 20m resolution. The terrain mask algorithm applies a series of geometric transformations to find all elevation angles at each azimuth angle around every pixel, out to 150km, where there is a line-of-sight to the horizon from an observer at heights 2m and 10m (fig. 2). The algorithm was developed in the Matlab environment, however initial attempts to produce masks for a large set of pixels overwhelmed memory available on a personal computer. In reference [1] we used Dask to parallelize computations for coordinate transformations. In this effort we migrated to MPI4Py to alleviate scaling constraints encountered using Dask, and attempted to reduce compute time for sub-regions of interest.

In [1], elevation masks for each pixel were written to over one million CSV files to the filesystem on Shaheen II. For the entire region of interest, the number of files produced quickly out grew system limitations for archiving and transfer. We worked around this by migrating to HDF5, a self describing data format which allows organizing data by pooling related variables, associating metadata to them, and describing relationships between various pools of variables [4]. HDF5 library has C/C++ and Fortran binding but the H5Py Python package provides bindings for Python.

The current effort implements the same formulation for computing terrain masks as [1], but implements parallel

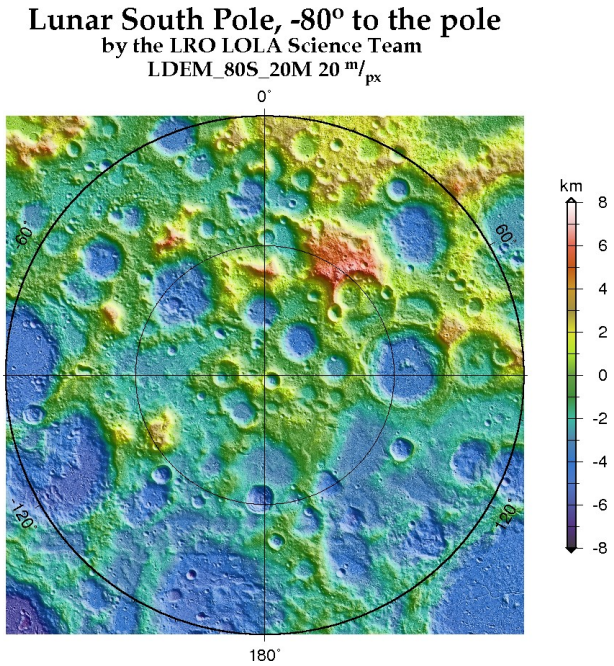

Fig. 1 We use publicly available map imagery which represents polar stereographic projections at -80 degrees to the Lunar South Pole. Data are publicly available from [2].

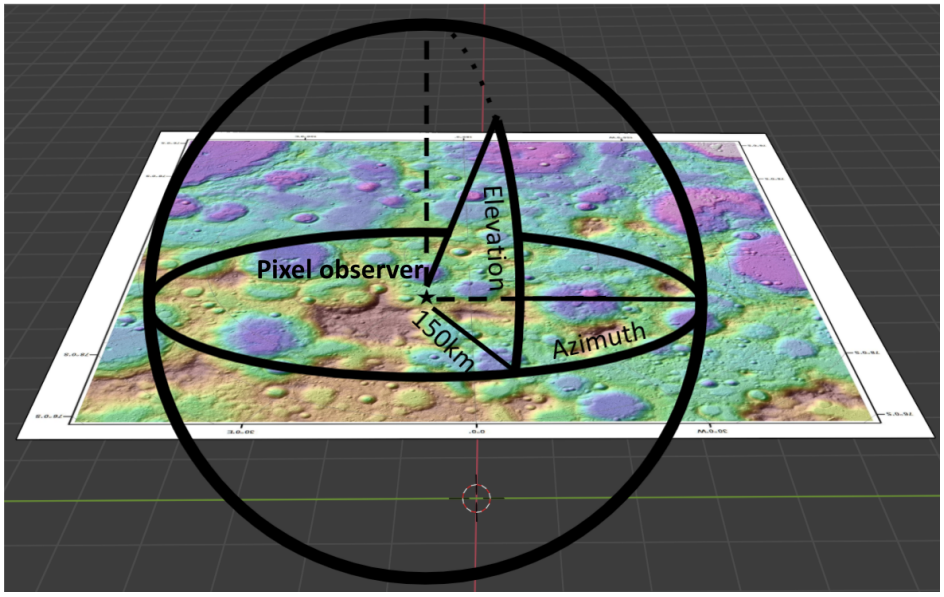

Fig. 2 For each pixel, elevation angles are computed at every azimuth angle around that pixel (0 to 360 degrees) for all lines of sight up to a 150km range.

programming with MPI to increase the scaling limit per job and orchestrates an I/O scheme which is more HPC friendly. Fig. 3 depicts the program flow.

The program starts when all MPI processes read the image file. The program is invoked with command line arguments, and each MPI process determines its chunk of work (a range of X-pixels) to process locally. These processes are assigned by MPI to become a part of the a new MPI communicator. A new communication context is created with a subset of MPI processes, using MPI process 0 of each new communicator as the writer process. For convenience we also create an additional communicator with all global IDs of all writer processes for its membership. Each MPI

process traverses through its assigned X-pixel range to locally compute calc_mask and maintains the masks locally in-memory. Each MPI process then sends these masks to its writer process.

Aggregation of data and writing are MPI collective operations, where writer processes use MPI-IO driver/API to write data to the parallel HDF5. When all the processes are complete and all the writer processes have received the data, the program blocks and does a collective write to the shared parallel HDF5 file. Each writer process writes data according to attributes of the created dataset in HDF5. Once I/O is completed, the program unblocks and moves to the next process all X-pixels associated to subsequent Y-pixel index.

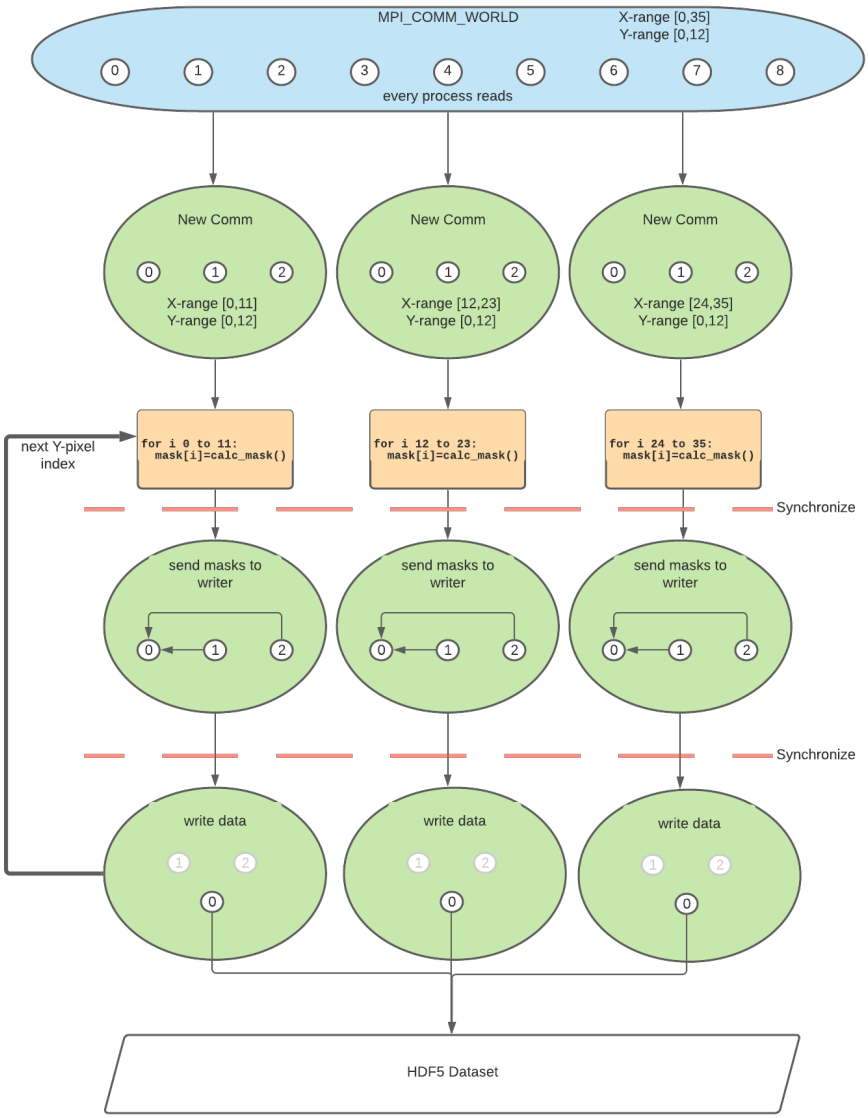

Fig. 3 The program flow uses MPI to manage decomposition of domain, local computation, aggregation of useful data and storing it in a file in parallel. The New Comm here is a sub-communicator created from MPI_COMM_WORLD. A MPI communicator is a context in which MPI processes communicate with each other. Both aggregation and writing of the data are collective MPI operations.

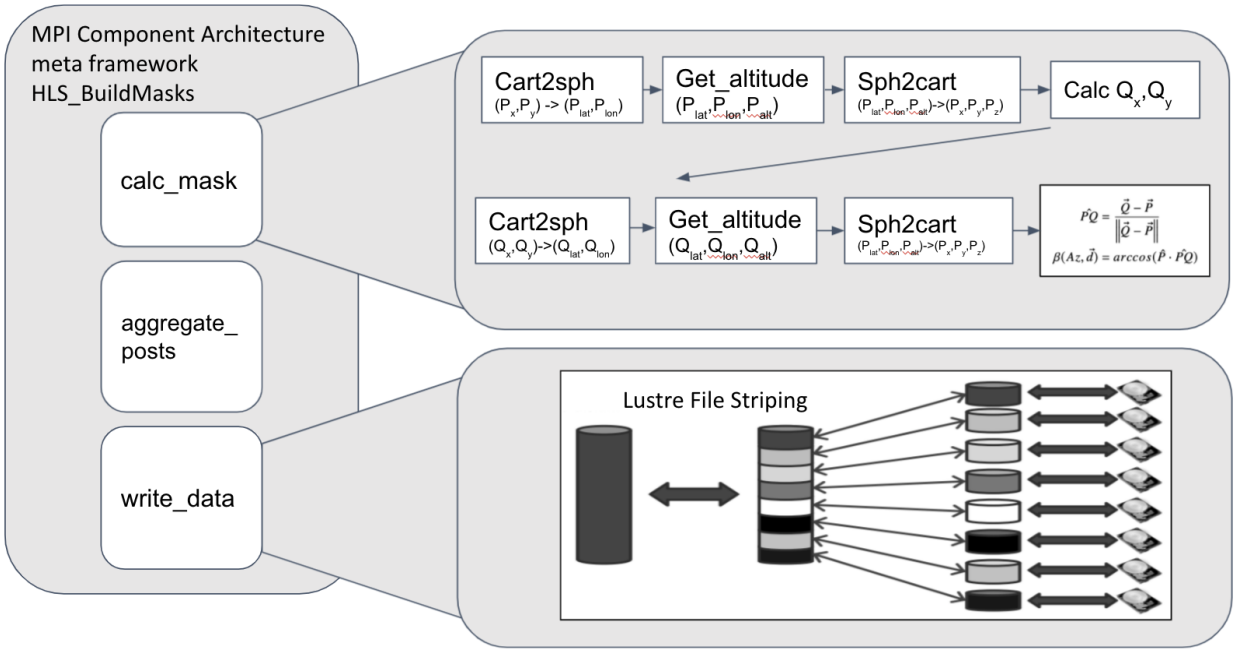

Fig. 4 The method calc_mask, described in [1], calculates terrain masks serially by applying geometric transformations all to pixel data, finding all elevation angles at each azimuth angle around every pixel where there is a line-of-sight to the horizon from an observer at varying heights.

III. Optimization

This migration to MPI4Py was motivated to increase the scale at which we can compute masks on a single job. Our goal was to process more pixels per job on many more compute nodes of Shaheen, concurrently, in order to analyze terrain for over 145 sq km. We decompose the processing in row dimension with a full array of pixels in X dimension at all columns for each pixel in the Y dimension, and process each pixel independently. Depending on the number of MPI processes, a range of X-pixels are mapped on every MPI process and completed under 650 seconds.

The other advantage of MPI4Py is leveraged by HDF5 formatting using the python API, H5Py. In the present architecture, jobs can populate binary HDF5 files across shared parallel processes. Writing parallel HDF5 files with the MPI-IO driver enables our code to take full advantage of Shaheen's parallel I/O scheme. We also introduced a scheme to agglomerate and write data by a set number of writers which is a configurable parameter at the start of the simulation. To this end, data can be aggregated and written into HDF5 files using fewer MPI processes, making 1000s of MPI processes available toward computation.

The workflow discussed in this study supports flexible on-demand simulation of various block sizes for any region. This is advantageous when the use case is vague or still in development. Furthermore, this makes the resulting HDF5 file and other derivative formats more transportable over internet to end-users. The availability of smaller file delivery sizes enables easier transfer of data product to the end user (scientist).

A. Use Case

From a survey of our the database users, it was determined that the users are a small number of university students that will request data to run in Matlab scripts. HDF5 provides utilities to query these files are directly loaded in MatLab. Still, any HDF5 reader can be used to ingest the files.

The users do not need 24/7 access to database, but might need to query for data at any time in the duration of their school term (once every few months). Users require only read access from this database, all products from their analysis would be stored on their own computers, and there are no products that will need to be put back into this database. Each query shall request a box of masks, of at most 608x608 contiguous pixels.

We aim to process sections of the raw input data of dimension 30400x30400 pixels, where the sub-sections processed will have an upper bound of 608x608. In our first time trial, we processed and delivered masks of heights 2m and 10m

MPI processes/ nodes	X range	Y range	X pixels/MPI process	Total wall time (hrs)	Elapsed time per Y pixel (s)	
					Mean	Std
7600/238	0-30400	0-1	4	0.231	479.283	72.452
7600/238	0-30400	0-80	4	14.36	643.06	116.84
152/5	0-608	0-1	4	0.133	459.075	10.685
152/5	0-608	0-80	4	9.53	428.455	4.807
152/5	0-608	0-188	4	24	454.016	7.04

Table 1 Total wall times for varying map sub-section sizes. For constant X-pixels per MPI process, the mean elapsed time for each Y-pixel processing remains tolerably similar, with a modest penalty of overhead and variability when running at large scale.

Metric	**Min**	**Max**	**Mean**	**Std**
elapsed	401.130608	1959.216399	643.057562	116.838969
calc	362.871023	447.07116	380.247298	10.21698
gather	0.000046	0.012044	0.000253	0.000768
writer	0	796.54282	20.554803	70.8676

Table 2 A breakdown for processing 30,400 x 80 px using MPI processes/nodes = 7600/238.

for a region of 608 x 160 pixels, as shown in figure 4, in a single job of 24 hours, and used three hours to completion. To process a 608x608 pixel region will take approximately 4 jobs to finish. We aim to produce 100 such data sets in total on-demand of the end-user's requested ranges.

B. Mock Processing

Data from mock processing runs are shown in tables 1 and 2 and help estimate core hour requirements. In table 1 we demonstrate the wall times for varying size maps. Note that the cost per pixel increases with the number of processes. A breakdown of wall time for the largest sized map suggests the wall time increases with more pixels due to writing the data. A breakdown of the largest run in table 1, which processes 30,400 x 80 px using MPI processes/nodes = 7600/2380, is shown in table 2. In the worst case scenario, the writer methods dominate the run-time, exceeding that of calculating masks.

Summarized in figure 4, calc_mask runs serially on each MPI process. Table 2 demonstrates that on average the cost of computing calc_mask is the most dominant part of the framework. For our experiments we typically assign 10 percent of the MPI processes for writing so more resources are allocated for computing masks. The next most expensive section of code is writing the data. On a shared parallel filesystem, Lustre in our case, write time to one disk, or Object Storage Target (OST), is susceptible to available space on the OST and number of users accessing that very OST. These variables can introduce performance hits, affecting overall wall times and predictability. This was confirmed by observing random high times reproducing the same job and can be attributed to the overall slow down to the shared file system during some different instances in the simulation. All MPI processes writing synchronously suffer this same constraint. As a possible mitigation to this issue, we have experimented with Lustre striping which chunks the data and writes to multiple disks/OSTs. It has shown promise to reduce this variability when setting stripe count to 8 and using 8 OSTs versus 1. Table 3 shows that with increasing stripe count or the number of Lustre OSTs (disks), both the overhead of writing data to parallel file and the variability decreases. We observed that the effect of striping is proportional to the aggregate file size. For small files written for a smaller assignment of X/Y-pixels (e.g. 608x80), the resulting aggregate file size was a 130MB and the benefit of Lustre stripping was not noticeable. Comparing it with processing larger number of X/Y-pixels (e.g. 30400x80) resulting in 6.6G, the effect of striping was significant, and is shown in Table 3.

OSTs	elapsed			writer		
	Max	Mean	Std	Max	Mean	Std
1	2340.20	1355.40	131.94	1267.11	93.24	282.99
4	1182.26	765.70	43.16	439.65	34.56	104.48
8	959.52	648.66	27.76	289.93	23.03	69.53
16	772.18	582.66	17.98	195.47	16.55	49.90

Table 3 Effect of increasing the Lustre stripe count (# of OSTs) on the overhead of writing data to disk and consequently on overall elapsed time.

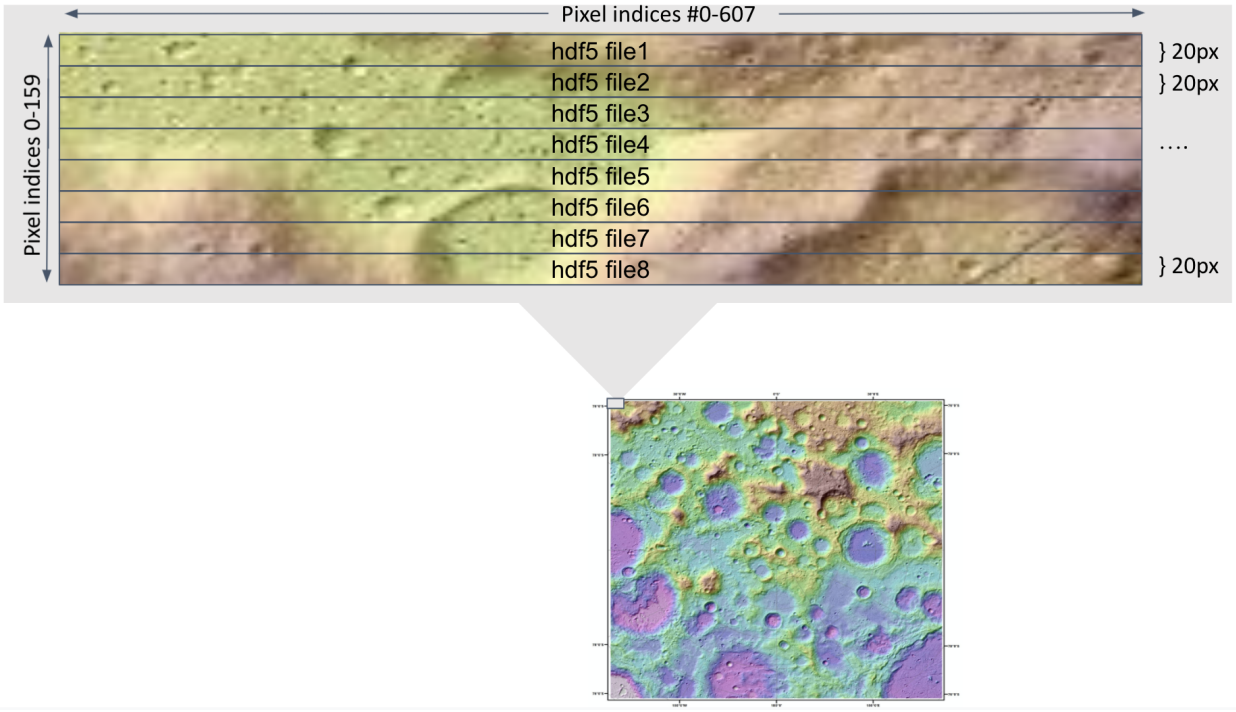

Fig. 5 In our first time trial, we processed and delivered masks for a region of 608 x 160 pixels.

IV. Conclusion

This effort supports NASA's vision to establish a permanent human presence on the Moon and advance Lunar mission design and formulation. Specifically, the resulting Lunar terrain database from this work will enable communication and surface exploration for assets positioned on the Lunar South Pole. Mainly, we demonstrate how to fully leverage a high performance computing environment to enable Lunar coverage analysis and terrain visualization.

The OpenSource software used in the code makes portability between HPC systems possible and can be installed on any HPC system/Supercomputer, cloud resource, and even on a workstation/laptop. Migrating from Dask to MPI has enabled us to scale out each batch job to greater number of cores, thus using the full capacity of a Supercomputer and reducing processing time by up to a factor of 10. The current framework is configurable to run a prescribed number of XY Pixels per batch job and sub-regions with no change to the code. By profiling availability and peak times of the compute resources, terrain masks at 20m resolution for the entire Lunar South Pole data set can be processed in reasonable amount of time. A conservative upper bound for completion is approximately one month on Shaheen II.

The HDF5 format has helped make the data product more manageable both to produce and consume by the end-user. While processing each job, we can now write to a single parallel file in binary format thus reducing the file size and the number of files generated by one batch job. Reduction in the number of output files for each batch job, compared to our Dask version, has allowed us to produce data within the limits of allowed on Shaheen II's Lustre filesystem. Each batch job produces one HDF5 file which, depending on the number of pixels (X Y), can be between 130 MB to approximately

7GB (for 30400 x 80 pixels processed). This is a manageable size for the end user to copy data via simple `scp` or `rsync` command or via Globus or gridFTP over the Internet.

Enabling serial Python code to leverge Dask Futures [1] was a straight-forward task with the boiler plate code used in the framework. However, the limitations we encountered when processing larger image sizes required a major refactor toward migrating to MPI4py and the effort involved was nontrival. The benefit however, has been worth the effort as it has provided flexibility and easier integration with traditional HPC libraries. For large scale data processing problems requiring high-throughput, MPI and feature rich I/O capabilities described in this work are enabling choices.

Acknowledgments

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (NASA), together with the Computational Sciences Group (KAUST Visual Computing Center) and the Supercomputing Laboratory (KAUST Core Labs).

References

[1] Lee, C.-A., Xie, H., Lee, C. H., Lyakhov, D., and Michels, D., "In Silico Methods for Space System Analysis: Optical Link Coding Performance and Lunar Terrain Masks," *AIAA Journal*, 2020. https://doi.org/10.2514/6.2020-4006.

[2] Smith, D. E., Zuber, M. T., Jackson, G. B., Cavanaugh, J. F., Neumann, G. A., Riris, H., Sun, X., Zellar, R. S., Coltharp, C., Connelly, J., Katz, R. B., Kleyner, I., Liiva, P., Matuszeski, A., Mazarico, E. M., McGarry, J. F., Novo-Gradac, A.-M., Ott, M. N., Peters, C., Ramos-Izquierdo, L. A., Ramsey, L., Rowlands, D. D., Schmidt, S., Scott, V. S., Shaw, G. B., Smith, J. C., Swinski, J.-P., Torrence, M. H., Unger, G., Yu, A. W., and Zagwodzki, T. W., "The Lunar Orbiter Laser Altimeter Investigation on the Lunar Reconnaissance Orbiter Mission," *Space Science Reviews*, Vol. 150, No. 1-4, 2010, pp. 209–241. https://doi.org/10.1007/s11214-009-9512-y.

[3] Dalcin, L., and Fang, Y.-L. L., "mpi4py: Status Update After 12 Years of Development," *Computing in Science Engineering*, Vol. 23, No. 4, 2021, pp. 47–54. https://doi.org/10.1109/MCSE.2021.3083216.

[4] Folk, M., Heber, G., Koziol, Q., Pourmal, E., and Robinson, D., "An Overview of the HDF5 Technology Suite and Its Applications," *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, Association for Computing Machinery, New York, NY, USA, 2011, p. 36–47. https://doi.org/10.1145/1966895.1966900, URL https://doi.org/10.1145/1966895.1966900.