developer.skatelescope.org Documentation

Marco Bartolini

Aug 07, 2021

HOME

The following sections provide the context and design aspects of the Imaging I/O Test prototype that is built to explore the capability of hardware and software to deal with the types of I/O loads that the SDP will have to support for full-scale operation on SKA1 (and beyond).

CHAPTER

ONE

BACKGROUND

The prototype design follows recommendations from the earlier working set memo (Wortmann, 2017). We see (de)gridding as the main computational work to be distributed. This is because this work will eventually involve heavy computation to deal with calibration and non-coplanarity. A scalable implementation requires distribution, as for SDP even "fat" nodes with multiple accelerators will likely not have enough power to shoulder all the required work on their own.

This leads to the proposed distribution by facets/subgrids, with facet data staying stationary while the distributed program walks through grid regions. This involves loosely synchronised all-to-all network communication between all participating nodes. As argued above, this characteristic is likely what will ultimately limit the performance of imaging/predict pipelines.

Finally, we have to consider raw visibility throughput. As we cannot possibly keep all visibilities in memory at all times, this data needs to be handled using mass storage technology. The achieved throughput of this system must be large enough to keep pace with the (accelerated) de/gridding work. While this only represents a comparatively predictable "base" load of order of magnitude 1 byte load per 1000 flops executed, we still need to pay attention due to the somewhat unusual amount of I/O required. This is especially significant because we will likely want to serve this data using flexible distributed storage technologies (Taylor, 2018), which introduce another set of scalability considerations.

1.1 Goals:

- Focused benchmarking of platform (especially buffer and network) hardware and software
- Verify parametric model assumptions concerning distributed performance and scaling, especially the extended analysis concerning I/O and memory from SDP memo 038

Main aspects:

- Distribution by facets/subgrids - involves all-to-all loosely synchronised communication and phase rotation work
- (De)gridding - main work that needs to be distributed. Will involve heavy computational work to deal with calibration and non-coplanarity, main motivation for requiring distribution
- Buffer I/O - needs to deal with high throughput, possibly not entirely sequential access patterns due to telescope layout

Technology choices:

- Written in plain C to minimise language environment as possible bottleneck
- Use MPI for communication - same reason
- OpenMP for parallelism - a bit limiting in terms of thread control, but seems good enough for a first pass
- HDF5 for data storage - seems flexible enough for a start. Might port to other storage back-ends at some point

Algorithmic choices:

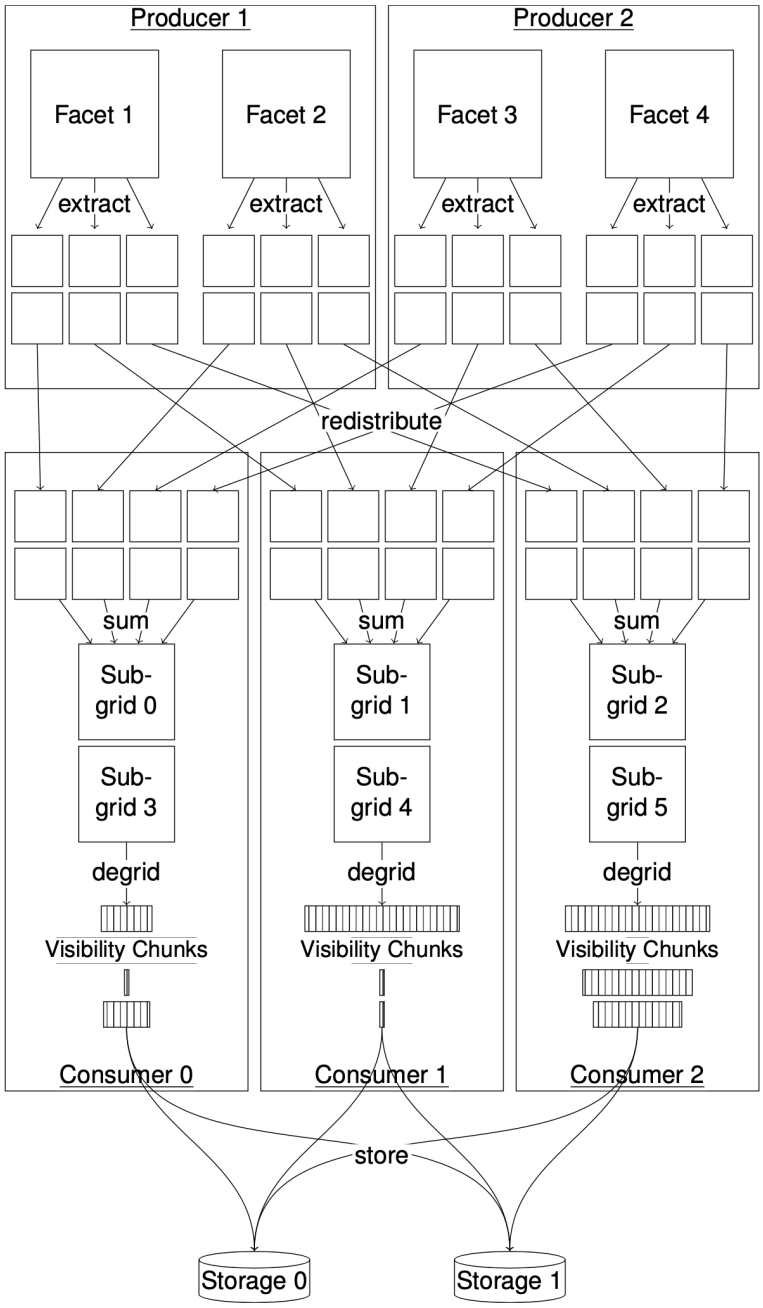
- We consider the continuum case, but with only one polarisation, taylor term, snapshot (= no reprojection). These would add code complexity, but are likely easy to scale up.
- We start with prediction (= writing visibilities). This is clearly the wrong way around, the backwards step will be added in the next version.
- Use slightly experimental gridded phase rotation ("recombination"), allowing us to skip baseline-dependent averaging while still keeping network transfer low and separating benchmark stages more cleanly.

CHAPTER

TWO

DESIGN

This prototype follows the overall structure proposed in Wortmann, (2017). We only consider the predict and not the backwards step – meaning that we go from a sky image to visibilities, but not the other way around. This means that we end up with two stages as shown below.



The idea is that we split the program into distributed processes, with both "producer" and "streamer" processes present and active on all participating throughout the run. This means that there is a continuous data re-distribution between the two steps, where we re-shuffle all relevant image data to get into the grid domain.

The way this works, producer processes hold some portion of the image data (facets). Collectively this represents a lot of data, up to a few terabytes for SKA-sized imaging cases. Instead of attempting to do a full Fourier Transform of this data to obtain the uv-grid, we instead re-construct sub-grid cut-outs sequentially until we have covered all relevant regions of the grid. This means that image data stays in place, and all producer processes walk through the grid in the same pre-determined fashion, streaming out sub-grid contributions to streamer processes.

Streamer processes collect these contributions from producer processes and assemble complete sub-grids (cut-outs of the entire grid). Each such sub-grid can then be used for de-gridding visibilities. The amount of visibility data that can be extracted from a given subgrid varies dramatically depending on the sub-grid position: A sub-grid near the centre

of the grid will both overlap more baselines, and tend to cover bigger windows in time and frequency.

2.1 Work Balance

Facet and subgrid work is assigned to producer and streamer processes at the start of the run. Facets are large memory objects, and the same amount of work needs to be done on each of them, therefore the number of nodes should be chosen so we can distribute them evenly - ideally the number of nodes should be the same as the number of facets.

On the other hand, each subgrid will be used to de-grid a different number of baselines and therefore visibilities depending on the grid area covered. This has to be balanced, keeping in mind that we want a healthy mix of subgrids to visibility chunks so the streamer doesn't run into a bottleneck on the network side of things. Right now we use a simple round-robin scheduling, splitting the work in central subgrids among nodes past a certain threshold. Parallelism

Both producer and streamer scale to many cores. The facet side of recombination / phase rotation is essentially a distributed FFT that we do separately on two axes, which leads to ample parallelism opportunities. However in order to keep memory residency manageable it is best to work on subgrid columns sequentially. Subgrid work is scheduled in rough column order to accomodate this.

The streamer employs three types of threads: One to receive data from the network and sum up subgrids, one to write visibilities as HDF5 (we serialise this for simplicity), and the rest to degrid baseline visibilities from subgrids. The network thread generally has very little work and will spawn a lot of tasks very quickly, which means that OpenMP will often have it yield to worker threads, effectively making it a worker thread.

2.2 Queues

In order to get good throughput every stage has input and output queues. We employ slightly different mechanisms depending on stage:

- The producer has only limited MPI slots per thread to send out subgrid contributions (current default: 8 subgrids worth)
- On the other end, the streamer has a limited number of MPI slots to receive facet contributions (current default: 32 subgrids worth)
- The network thread will assemble sub-grids once all contributions have been received, and create OpenMP tasks for degridding. The subgrids in question will be locked until all de-grid tasks have been finished, the queue is limited to the same number of entries as the incoming facet contribution queue (so 32 entries).
- OpenMP limits the number of degrid tasks that can be spawned, which means that we additionally have a degrid task queue with limited capacity (Seems to be around 128 for gcc). Note that a task can cover many baselines (current default is up to 256 - so roughly 32768 baselines maximum).
- Finally, once visibilities have been generated, those will have to be written out. This is done in terms of visibility chunks (size configurable, the - low - default is currently 32x32). The queue has a length of 32768 entries (roughly half a GB worth of data with default parameters).

This part deals with practical aspects of the benchmark on how to install and run on HPC machines.

CHAPTER

THREE

PREREQUISITES AND INSTALLATION

The following prerequisites must be installed:

- gcc
- cmake
- git
- git lfs
- python3
- HDF5 (doesn't need to have parallel support)
- OpenMP
- MPI (with threading support)
- FFTW3.
- Optional: Singularity (only required if building/running containerised images of the code).

On the MacOS, `homebrew` can be used to install all the listed dependencies.

The compilation process will use `mpicc`, and it has been tested with both GCC and ICC. It has also been tested on both Linux and MacOS.

To set up the repository, get data files, and compile, run the following steps:

```
git clone git@gitlab.com:ska-telescope/sdp/ska-sdp-exec-iotest.git
cd ska-sdp-exec-iotest
git lfs pull origin
mkdir build
cd build/
cmake ../
make -j$(nproc)
```

Singularity image can be pulled from GitLab registry using `oras` library.

```
singularity pull iotest.sif oras://registry.gitlab.com/ska-telescope/sdp/ska-sdp-exec-
↪iotest/iotest:latest
```

To build the image with Singularity:

```
singularity build iotest.sif ska-sdp-exec-iotest.def
```

CHAPTER

FOUR

PARAMETERISATION

4.1 Recombination parameters

They are generally given as "image-facet-subgrid" sizes. Number of facets depends on relation of facet size to image size. Total memory requirements depend on image size. The option can be used by passing argument to `--rec-set`. Available options and their facet count are as follows:

Image config (`--rec-set`)	Number of facets
256k-256k-256, 192k-192k-256, 160k-160k-256, 128k-128k-256, 96k-96k-256, 80k-80k-256, 64k-64k-256, 48k-48k-256, 40k-40k-256, 32k-32k-256, 24k-24k-256, 20k-20k-256, 16k-16k-256, 12k-12k-256, 8k-8k-256	4
256k-128k-512, 192k-96k-512, 160k-80k-512, 128k-64k-512, 96k-48k-512, 80k-40k-512, 64k-32k-512, 48k-24k-512, 40k-20k-512, 32k-16k-512, 24k-12k-512, 20k-10k-512, 16k-8k-512, 12k-6k-512, 8k-4k-512, 256k-192k-256, 192k-144k-256, 160k-120k-256, 128k-96k-256, 96k-72k-256, 80k-60k-256, 64k-48k-256, 48k-36k-256, 40k-30k-256, 32k-24k-256, 24k-18k-256, 20k-15k-256, 16k-12k-256, 12k-9k-256, 8k-6k-256, 256k-128k-256, 192k-96k-256, 160k-80k-256, 128k-64k-256, 96k-48k-256, 80k-40k-256, 64k-32k-256, 48k-24k-256, 40k-20k-256, 32k-16k-256, 24k-12k-256, 20k-10k-256, 16k-8k-256, 12k-6k-256, 8k-4k-256	9
256k-96k-512, 192k-72k-512, 160k-60k-512, 128k-48k-512, 96k-36k-512, 80k-30k-512, 64k-24k-512, 48k-18k-512, 40k-15k-512, 32k-12k-512, 24k-9k-512, 20k-7680-512, 16k-6k-512, 12k-4608-512, 8k-3k-512, 256k-96k-256, 192k-72k-256, 160k-60k-256, 128k-48k-256, 96k-36k-256, 80k-30k-256, 64k-24k-256, 48k-18k-256, 40k-15k-256, 32k-12k-256, 24k-9k-256, 20k-7680-256, 16k-6k-256, 12k-4608-256, 8k-3k-256	16
192k-64k-768, 96k-32k-768, 48k-16k-768, 24k-8k-768, 12k-4k-768, 256k-64k-1k, 192k-48k-1k, 160k-40k-1k, 128k-32k-1k, 96k-24k-1k, 80k-20k-1k, 64k-16k-1k,	36

As for the size of the image, we can estimate it using image-size * image-size * 16 Bytes. For instance, for 8k image, the total image size would be 8192x8192x16 = 1073741824 B = 1024 MiB.

Certain image configurations have alias and they are listed as below

Image config (`--rec-set`)	**alias**
512-216-256	T05_
8k-2k-512	tiny
16k-8k-512	small
32k-8k-1k	smallish
64k-16k-1k	medium
96k-12k-1k	large
128k-32k-2k	tremendous
256k-32k-2k	huge

`--rec-set` option can be passed using either sizes like `--rec-set=8k-2k-512` or alias name like `--rec-set=small`. The size of the image gives the **memory requirements** for each recombination parameter set. For instance, running the case with `--rec-set=256k-32k-2k` will require at least 1 TiB of cumulative memory on all the reserved nodes. However, the real memory requirement would be much more than 1 TiB. You can find the approximate memory required to run each of these cases in *Memory requirements*.

Note that the `--rec-set=256k-32k-2k` is not suited for `--vis-set=lowbd2`. As SKA1 LOW works with higher wavelengths, this recombination set will give an enormous Field of View (fov), which means we are approximating more curved sky into 2D image. The fov can be computed as

```
fov = fov_frac * image_size * c / (2 * max_bl * max_freq)
```

For SKA1 LOW, maximum baseline would be 60 km and maximum frequency would be 300 MHz and so using an image size of 131072 and 0.75 of field of view of image, we obatain fov = 0.81, which is 40% of sky sphere (which goes from -1 to +1, and therefore as size 2). This can be very inefficient and this recombination should be only used with `--vis-set=midr5`. In the case of SKA1 Mid, this value comes to 0.2.

By default number of MPI processes are divided equally among producer (facet worker) and subgrid workers. The recommendation is to have one facet worker per facet, **plus one**. Background is that the prototype can't properly parallelise work on different facets if a worker has to hold multiple, which means that work on the different facets will be partly sequentialised, which can lead to deadlock if the queues fill up. On the other hand, there is little reason in not having one worker per facet even if it means additional ranks: They are heavily I/O bound, and therefore rarely compete with each other or other processing on the node. But they create many unnecessary threads, which can become a problem eventually. This will be addressed in the future work of the prototype.

Furthermore, one additional worker is a good idea because this means we have a dedicated worker for work assignment. For basically any non-trivial distributed run, assigning work quickly is essential, and MPI seems to introduce big delays into such request-reply interactions while there's heavy data movement to the same rank. Usually, the last MPI rank is the assignment worker. **Important** to note here is there is a deadlock situation when not using facet workers, ie, `--facet-workers=0`. Always use at least one or more facet workers.

4.2 Visibility set

This parameter specifies the telescope layout and defines the baselines. This option can be invoked using `--vis-set`. Several configurations are available like “lowbd2”, “lowbd2-core”, “lowr3”, “midr5”, “lofar”, “vlaa”. “lowbd2” and “midr5” correspond to SKA LOW and SKA MID layouts. “vlaa” contain very few antennas and thus very fast to run. It can be used to check the running of the code. Only “lowbd2”, “midr5” and “vlaa” are extensively tested.

4.3 Time and frequency settings

These parameters help us to define the time snapshot, dump times, frequency range and number of frequency channels. These can be used as `--time=<start>:<end>/<steps>[/<chunk>]` and `--freq=<start>:<end>/<steps>[/<chunk>]`. `<start>` and `<end>` indicate ranges of time and frequency, `<step>` indicate time or frequency step. Finally `<chunk>` is both the chunk size used for writing visibilities out in HDF5, but also decides the granularity at which we handle visibilities internally: Each chunk will be generated as a whole. Note that the larger the chunks, the more likely it is that they involve data from multiple subgrids, which might require re-writing them. For instance, `--time=-460:460/1024/32 --freq=260e6:300e6/8192/32` means 15 minutes snapshot with 0.9 sec dump rate (hence 1024 steps) and 32 chunks in time. Similarly 40 MHz is divided into 8192 frequency channels with a chunk size of 32. Visibilities are nothing but complex doubles (16 bytes each) and with 32 chunks in time and frequency means 16 * 32 * 32/1024 = 16 KiB chunk.

4.4 Declination

Declination of the phase centre. Due to the position of the SKA telescopes on the southern hemisphere, values around -30 result in a phase centre closest to the zenith, and therefore lowest non-coplanarity complexity. This can be passed using `--dec=-30`. But at this point the complexity due to non-coplanarity is minimised via reprojection - this means that up to around 45 degrees, there should be little difference in terms of performance.

4.5 Source count

The default is that we generate a random image that is entirely zeroes with a number of on-grid source pixels of intensity 1 in them. Using this approach it is trivial to check the accuracy using direct Fourier transforms. Activating this option means that the benchmark will check random samples of both subgrids and visibilities and collect RMSE statistics. Number of source counts can be used by setting `--source-count=10`. This case will set 10 random point sources in cloud of zeros in the image.

4.6 Producer thread count

The number of OpenMP threads configured during the runtime is used for the subgrid workers as they do most of computational intensive tasks. On the other hand facet workers only hold the facet data in the memory while sending subgrids to subgrid workers. Hence, they do not need as many OpenMP threads as subgrid workers. The number of threads for facet workers can be configured using `--producer-threads` option.

4.7 Queue parameters

Multiple limited sized queues are used in the prototype in order to exert the back-pressure and prevent running out of memory.

- Send queue (`--send-queue`): Number of subgrids facet workers transfer out in parallel. This is per OpenMP thread!
- Subgrid queue(`--subgrid-queue`): Number of receive slots total per subgrid worker. Also decides number of subgrids that are kept.

These are roughly opposite sides of the same coin. Note that the length of the subgrid queue strongly interacts with dynamic load balancing: If the subgrid queue is long, we assign work relatively early, and might be stuck with a bad work assignment. If the subgrid queue is too short, we might lose parallelism and struggle to keep workers busy.

Task queue (`--bls-per-task`) for degridding visibilities from completed subgrids. One task (= one thread) always handles one baseline at a time (looping through all chunks overlapping the subgrid). By assigning multiple baselines per task we can make things more coarse-granular, which makes sense to keep task switching costs low and prevent generating too many tasks.

Finally, the task queue (`--task-queue`) limit is there to prevent us running into OpenMP limits around the number of tasks (something like that seems to happen around ~256). Beyond that, a shorter task queue means we proceed through subgrids in a more sequential fashion, freeing up subgrid slots more quickly. However as usual we need a queue long enough to provide enough parallelism for all threads.

4.8 Gridding options

Shouldn't really have to override these very often except if gunning for higher accuracy. This option can be used by passing arguments to `--grid` and `--grid-x0`. The x0 value determines how much of image space we actually use (and therefore have to cover with facets). Reducing it will lead to better precision and fewer facets, but also less usable field-of-view.

4.9 Non-coplanarity

We generally have two ways for dealing with this -

- w-stacking: Basically like an outer loop on the recombination. Precise no matter the distance in w, but all facets need to be re-generated and image data needs to be re-distributed. Still not as inefficient as it sounds, as we skip all unneeded grid space. Completely okay to have hundreds of w-stacking planes.
- w-towers: Subgrid workers use subgrid data to infer a stack of nearby subgrids, without needing to go back to image data. Will cause errors to creep in from the sides of the subgrid, therefore must have more subgrid margins (= place them closer together).

The w-step (`--wstep`) is (roughly) how many w-tower planes we have before we rather use w-stacking to generate the subgrid. More w-towers means that we move more FFT work from facet workers to subgrid workers. The margin (`--margin`) has (unfortunately) to be chosen by trial-and-error. Look out for messages such as

```
Subgrid x/y/z (A baselines, rmse B, @wmax:  C)
```

in the log: Comparing B and C gives an indication how badly the accuracy at the top of the "w-towers" has detoriated. Where C is larger than the required visibility accuracy, either increase the margin or reduce the wstep. Note that the margin will get rounded (up), as not all subgrid spacings are permissible.

The target error (`--target-err`) specifies maximum admissible error for determining w-tower height. This value can be provided in place of `--wstep`, where the number of w-tower planes are estimated based on the target error value. If this is not provided, an order of magnitude off the best error that we would get from recombination parameters is used to estimate w-tower planes.

4.10 Writer settings

If we want to write the output visibilities to the disk or file system, we need to pass the file path as a positional argument. It is **important** to use `%d` placeholder, like `/tmp/out%d.h5` when passing this argument. The placeholder will be replaced by a number determined by number of MPI ranks and number of writer threads. By default, the code always use 2 writer threads per streamer process to output the visibility data. This can be changed by using `--writer-count` option.

Another **important** detail here is the Imaging IO test use by default `pthreads` from streamer process as writer threads. Each writer thread writes to its own file. We are using HDF5 for writing the visibility data and there is a known issue with the HDF5 when trying to use multiple threads to write data concurrently (https://www.hdfgroup.org/2020/11/webinar-enabling-multithreading-concurrency-in-hdf5-community-discussion/). HDF5 uses some sort of global lock when using multiple threads writing concurrently **even to different files**. So, using more than 1 writer thread will not increase the throughput. Moreover, we noticed from few runs that using more than 1 writer thread is counter productive.

Fortunately, the solution is to use `--fork-writer` option, which basically forks the process to create a new writer process. The difference is that now we use a complete process to create the writer whereas without this option only a thread is created for writer. The only way to achieve concurrent writes without any global lock using HDF5 libraries is use full process rather than threads. On the MacOS `--fork-writer` **will not work** as semaphores locks used for the Darwin kernel are based on threads and not processes.

Often, OpenMPI with openib support for Infiniband will complain about forking the process. This is due to inconsistencies between OpenMPI and Openib libraries. From OpenMPI 4.0.0, ucx is the preferred support for Infiniband and forking is not an issue using these libraries.

CHAPTER

FIVE

CONFIGURATION SETTINGS

Number of MPI processes should be chosen based on the recombination parameters (`--rec-set`) and number of sockets (NUMA node) available on each node. Essentially number of MPI processes are sum of number of facet workers, number of subgrid workers and number of dynamic assignment worker. Currently, we use one worker for dynamic work assignment and this might change in the future. Number of facet workers should be equal to the number of facets in the recombination parameters, which is presented in *Recombination parameters*. Number of subgrid workers should be chosen based on the NUMA nodes on each compute node. If a compute node has two sockets, which means two NUMA nodes, we should ideally have two subgrid workers. If there are 16 physical cores on each socket with hyperthreading enabled, we can use all the available cores, both physical and logical, for the subgrid workers. Hence, we can use for this example, number of OpenMP threads would be 32. It is noticed that using all available cores gives us a marginal performance benefits for the dry runs.

```
num_mpi_process = num_facets + num_nodes * num_numa_nodes + num_dynamic_work_
↪assignement
```

where `num_nodes` is number of nodes in the reservation, `num_numa_nodes` is number of NUMA node **in each compute node**, `num_facets` is number of facets and finally, `num_dynamic_work_assignement` is number of dynamic work assignment workers which is 1.

Some of the configurations for single and multiple nodes are:

5.1 Single Node

The program can be configured using the command line. Useful configurations:

```
$ mpirun -n 2 ./iotest --rec-set=T05
```

Runs a small test-case with two local processes to ensure that everything works correctly. It should show RMSE values of about 1e-8 for all recombined subgrids and 1e-7 for visibilities.

```
$ ./iotest --rec-set=small
$ ./iotest --rec-set=large
```

Does a dry run of the "producer" end in stand-alone mode. Primarily useful to check whether enough memory is available. The former will use about 10 GB memory, the latter about 350 GB.

```
$ ./iotest --rec-set=small --vis-set=vlaa --facet-workers=0
```

Does a dry run of the "degrid" part in stand-alone mode. Good to check stability and ensure that we can degrid fast enough.

```
$ ./iotest --rec-set=small --vis-set=vlaa --facet-workers=0 /tmp/out%d.h5
```

Same as above, but actually writes data to the out the given file. By default, each subgrid worker creates two writer threads and hence `%d` placeholder is used. Data will be all zeroes, but this runs through the entire back-end without involving actual distribution. Typically quite a bit slower, as writing out data is generally the bottleneck.

```
$ mpirun -n 2 ./iotest --rec-set=small --vis-set=vlaa /tmp/out%d.h5
```

Runs the entire thing with one facet and one subgrid worker, this time producing actual-ish visibility data (for a random facet without grid correction).

```
$ mpirun -n 2 ./iotest --rec-set=small --vis-set=vlaa --time=-230:230/512/128 --
↪freq=225e6:300e6/8192/128 /tmp/out.h5
```

The "–vis-set" and "–rec-set" parameters are just default parameter sets that can be overridden. The command line above increases time and frequency sampling to the point where it would roughly correspond to an SKA Low snapshot (7 minutes, 25% frequency range). The time and frequency specification is `<start>/<end>/<steps>/<chunk>`, so in this case 512 time steps with chunk size 128 and 8192 frequency channels with chunks size 128. This will write roughly 9 TB of data with a chunk granularity of 256 KB.

5.2 Distributed

As explained the benchmark can also be run across a number of nodes. This will distribute both the facet working set as well as the visibility write rate pretty evenly across nodes. As noted you might want at minimum a producer and a streamer process per node, and configure OpenMP such that its threads take full advantage of the machine's available cores at least for the subgrid workers. Something that would be worth testing systematically is whether facet workers might not actually be faster with fewer threads. They are likely waiting most of the time.

To distribute facet workers among all the nodes `--map-by node` argument should be used for OpenMPI. By default OpenMPI assigns the processes in blocks and without `--map-by node` argument, one or more nodes might get many facet workers. This is not what we want as facet workers are memory bound. With OpenMPI default mapping, we would end up with subgrid workers on all low ranks and facet workers on high ranks. As facet workers wait most of the time (so use little CPU), yet use the a lot of memory, that would cause the entire thing to become very unbalanced.

For example, if we use `--rec-set=small` across 8 nodes (2 NUMA nodes each and 16 cores on each socket) we want to run 10 producer processes (facet workers) and 16 streamer processes (subgrid workers), using 16 threads each:

```
export OMP_NUM_THREADS=16
mpirun --map-by node -np 26 ./iotest --facet-workers=10 --rec-set=small $options
```

This would allocate 2 streamer processes per node with 16 threads each, appropriate for a node with 32 (physical) cores available. Facet workers are typically heavily memory bound and do not interfere too much with co-existing processes outside of reserving large amounts of memory.

This configuration (`mpirun --map-by node -np 26 ./iotest --facet-workers=10 --rec-set=small`) will just do a full re-distribution of facet/subgrid data between all nodes. This serves as a network I/O test. Note that because we are operating the benchmark without a telescope configuration, the entire grid is going to get transferred - not just the pieces of it that have baselines.

Other options for distributed mode:

```
options="--vis-set=lowbd2"
```

Will only do re-distribute data that overlaps with baselines, then do degridding.

```
options="--vis-set=lowbd2 /local/out%d.h5"
```

Also write out visibilities to the given file. Note that the benchmark does not currently implement parallel HDF5, so different streamer processes will have to write separate output files. The name can be made dependent on streamer ID by putting a `%d` placeholder into it so it won't cause conflicts on shared file systems.

```
options="--vis-set=lowbd2 --fork-writer --writer-count=4 /local/out%d.h5"
```

This will create 4 writer processes for each subgrid worker and writes the data to the file system. Remember that without `--fork-writer` option, the benchmark will create only threads and HDF5 library currently do not have support for concurrent writing threads. So, it will not increase the data throughput.

5.3 SKA1 LOW and MID settings

To run the benchmark that correspond to SKA1 LOW and SKA1 MID settings following configuration can be used. These settings are provided assuming we are running on 16 compute nodes with 2 NUMA nodes and 32 cores on each compute node and `--rec-set=small`.

- SKA LOW:

```
export OMP_NUM_THREADS=16
mpirun --map-by node -np 42 ./iotest --rec-set=small --vis-set=lowbd2 --facet-
↪workers=10 --time=-460:460/1024/64 --freq=260e6:300e6/8192/64 --dec=-30 --
↪source-count=10 --send-queue=4 --subgrid-queue=16 --bls-per-task=8 --task-
↪queue=32 --target-err=1e-5 --margin=32`
```

- SKA MID:

```
export OMP_NUM_THREADS=16
mpirun --map-by node -np 42 ./iotest --rec-set=small --vis-set=midr5 --facet-
↪workers=10 --time=-290:290/4096/64 --freq=0.35e9:0.4725e9/11264/64 --dec=-30 --
↪source-count=10 --send-queue=4 --subgrid-queue=16 --bls-per-task=8 --task-
↪queue=32 --target-err=1e-5 --margin=32
```

The above configurations run the benchmark in the dry mode without writing the visibility data to the file system. If we want to write the data, we have to add `/some_scratch/out%d.h5` to the end, where `/some_scratch` is the scratch directory of the file system. SKA LOW has 131,328 baselines and with above configuration 1,101,659,111,424 (131,328x1024x8192) visibilities will be produced which correspond to roughly 14 TB of data. SKA MID will have 19,306 baselines, so 890,727,563,264 visibilities which is 17 TB of data. The amount of generated data can be effected by the chunk size used. Bigger chunk size involves more subgrids which eventually require some re-writes.

5.4 Running with singularity image

Singularity image can be pulled from GitLab registry as shown in *Prerequisites and Installation*. Currently, the singularity image supports three different entry points, which are defined using `apps` feature from SCIF. Three entry points are as follows:

- `openmpi-3.1-ibverbs`: OpenMPI 3.1.3 is built inside the container with IB verbs support for high performance interconnect.
- `openmpi-4.1-ucx`: OpenMPI 4.1.0 with UCX is build inside the container

- `openmpi-3.1-ibverbs-haswell`: This is similar to `openmpi-3.1-ibverbs`, albeit, the imaging I/O test code is compiled with `haswell` microarchitecture instruction set. This default entrypoint to the container unless otherwise specified.

To list all the apps installed in the container, we can use singularity `inspect` command as

```
singularity inspect --list-apps iotest.sif
```

Typically, singularity can be run using MPI as follows:

```
mpirun -np 2 singularity run --env OMP_NUM_THREADS=8 --bind /scratch --app ompi-3.1-
↪ibverbs iotest.sif ${ARGS}
```

The above command launches two MPI processes with 8 OpenMP threads with entry point defined in `ompi-3.1-ibverbs` of `iotest.sif` image and `${ARGS}` correspond to typical Imaging I/O test arguments presented above. If visibility data is written to non-standard directories, it is necessary to bind the directory using `--bind` option as shown in the command.

CHAPTER

SIX

OPENMP, MPI AND SLURM SETTINGS

6.1 MPI specific settings

In addition to `--map-by node` some other MPI arguments can be useful while running the benchmark. Some of them are

- `--tag-output`: This will tag each line of the stdout with the MPI rank.
- `--timestamp-output`: This will put the time stamp on the each line of stdout

MCA parameters provide more finer level tuning on the `mpirun`. For OpenMPI 3, Infiniband (IB) support is provided via openib libraries. To use only IB devices, we can use `--mca btl openib,self,vader`. To use specific port `--mca btl_openib_if_include mlx5_0:1` must be used. This says that we should use port `mlx5_0:1` only for MPI communications. **Important** to know that this option must be passed on p3-alaska to get best performance. Simply passing `--mca btl openib,self,vader` will give us warnings on p3-alaska as one of the high speed port is broken on the cluster network interface.

Another MCA parameter we should pass here is `--mca mpi_yield_when_idle 1`. This puts MPI in degraded busy wait mode. By default OpenMPI uses aggressive busy wait meaning that the processes continually poll for the messages to decrease the latency. Sometimes it can result in very high CPU usages and we can turn it off passing this argument. When there are more processes than processors, ie, when we overcommit the resources, OpenMPI automatically uses degraded busy wait mode. This gives minor performance benefits when running on fewer nodes.

Using MPI argument `--map-by node` binds a given process to a certain node. Typically OS moves these processes around the sockets based on its scheduling policies. Imagine a case where an MPI process starts in Socket 0. After sometime, if Socket 0 has too much work to do, the kernel might move this MPI process to Socket 1 to balance the work on its resources. But this is not desirable in certain cases as we will lose NUMA locality when migrating the processes. It is important to define the process bindings in the MPI applications. OpenMPI binds the processes to sockets by default and for the Imaging IO benchmark, this works well as we will not use NUMA locality.

All these options are OpenMPI specific. The equivalent options for Intel MPI will be documented in the future.

6.2 OpenMP specific settings

Similar to the process bindings we need to bind the OpenMP threads too. As processes we would like the OpenMP threads to bind to sockets as well. To do so we should export couple of environment variables `export OMP_PROC_BIND=true OMP_PLACES=sockets`. The first variable turns on the OpenMP thread binding option and second one specifies the place where the threads should bind.

6.3 SLURM specific settings

MPI startup is not standardised and thus, `mpirun` is not only way to run an MPI application. On most of the production machines `srun` is used to run the MPI applications. So some of the MPI specific arguments discussed above will not work for `srun`. If the `srun` is using OpenMPI implementation, MCA parameters can still be used by passing them via environment variables with prefix `OMPI_MCA`. For example command line option `--mca mpi_yield_when_idle 1` can also be passed as `export OMPI_MCA_mpi_yield_when_idle=1`. Process binding in SLURM `srun` is done via `--distribution` and `--cpu-bind` options. For the Imaging IO benchmark, we can use `--distribution=cyclic:cyclic:fcyclic --cpu-bind=verbose,sockets`. More details on these options can be found in SLURM documentation.

A sample SLURM script using `srun` to launch MPI that is used on JUWELS (JSC) machine is

```
#!/bin/bash

#SBATCH --time=01:00:00
#SBATCH --job-name=sdp-benchmarks
#SBATCH --account=training2022
#SBATCH --nodes=128
#SBATCH --ntasks=12288
#SBATCH --partition=batch
#SBATCH --output=/p/project/training2022/paipuri/sdp-benchmarks/io_bench/out/juwels-
↪io-scalability-bare-metal-128-small-lowbd2-256-256-2.out
#SBATCH --error=/p/project/training2022/paipuri/sdp-benchmarks/io_bench/out/juwels-io-
↪scalability-bare-metal-128-small-lowbd2-256-256-2.out
#SBATCH --mail-type=FAIL
#SBATCH --no-requeue
#SBATCH --exclusive

# GENERATED FILE

set -e

# Purge previously loaded modules
module purge
# Load GCC OpenMPI HDF5/1.10.6-serial FFTW/3.3.8 git, git-lfs modules
module load GCC/9.3.0 OpenMPI/4.1.0rc1 FFTW/3.3.8 git-lfs/2.12.0 git/2.28.0 zlib/1.2.
↪11-GCCcore-9.3.0 HDF5/1.10.6-GCCcore-9.3.0-serial

# Give a name to the benchmark
BENCH_NAME=juwels-io-scalability

# Directory where executable is
WORK_DIR=/p/project/training2022/paipuri/sdp-benchmarks/io_bench/ska-sdp-exec-iotest/
↪src

# Any machine specific environment variables that needed to be given.
# export UCX log to suppress warnings about transport on JUWELS
# (Need to check with JUWELS technical support about
# this warning)
export UCX_LOG_LEVEL=ERROR

# Any additional commands that might be specific to a machine


# Change to script directory
cd $WORK_DIR
```

(continues on next page)

(continued from previous page)

```
echo "JobID: $SLURM_JOB_ID"
echo "Job start time: `date`"
echo "Job num nodes: $SLURM_JOB_NUM_NODES"
echo "Running on master node: `hostname`"
echo "Current directory: `pwd`"

echo "Executing the command:"
CMD="export OMPI_MCA_mpi_yield_when_idle=1 OMP_PROC_BIND=true OMP_PLACES=sockets OMP_
↪NUM_THREADS=24 &&
  srun --distribution=cyclic:cyclic:fcyclic \
  --cpu-bind=verbose,sockets --overcommit --label --nodes=128 --ntasks=266 --cpus-per-
↪task=24 ./iotest \
  --facet-workers=10 --rec-set=small --vis-set=lowbd2 --time=-460:460/1024/256 --
↪freq=260e6:300e6/8192/256 --dec=-30 \
  --source-count=10 --send-queue=4 --subgrid-queue=16 --bls-per-task=8 --task-
↪queue=32 --fork-writer --wstep=64 \
  --margin=32 --writer-count=8 \
  /p/scratch/training2022/paipuri/sdp-benchmarks-scratch/juwels-io-scalability-bare-
↪metal-128-small-lowbd2-256-256-2/out%d.h5\
   && rm -rf \
  /p/scratch/training2022/paipuri/sdp-benchmarks-scratch/juwels-io-scalability-bare-
↪metal-128-small-lowbd2-256-256-2/out*.h5"

echo $CMD

eval $CMD

echo "Job finish time: `date`"
```

The equivalent of `--tag-output` is `--label` with `srun`. It is better to reserve all the nodes and cores at the `#SBATCH` level and use these resources in the `srun` command. A standard compute node on JUWELS has 48 physical (96 with logical included) cores and if we are using 128 nodes, the total number of available cores are 128x96=12288 cores. By looking at the SLURM script, we see that we are asking for 12288 tasks, which means SLURM will reserve all the cores for the job. When we launch the application using `srun`, we can use these reserved resources that is best suited for our application.

In the case where we want to overcommit the resources, passing `--ntasks` and `--cpus-per-task` at the `#SBATCH` will not work as SLURM complains that it does not have enough resources to reserve. For example, using the same example of JUWELS standard node, we want 10 tasks with 24 OpenMP threads per task on one node. This will need 240 CPUs in total and a standard JUWELS compute node only offers 96 CPUs. If we ask SLURM to reserve the resources in this configuration, it will return an error. However, the same configuration can be used using `srun` with `--overcommit` flag without any issues. So, it is always safe to use `--ntasks` and `--cpus-per-task` on the `srun`. On JUWELS, we had to exclusively specify the `--cpu-bind` option as the default is `--cpu-bind=cores`. This is something we should keep in mind and check the default settings before running the benchmark on production machines.

CHAPTER

SEVEN

COMPUTE RESOURCE REQUIREMENTS

7.1 Memory requirements

In the section *Recombination parameters*, the table contains the image sizes of various possible inputs. We should have a cumulative memory on all compute nodes of at least the size of the image. We are also using limited sized *Queue parameters* and buffers in the benchmark. These queue sizes are configurable and therefore, we should pay attention to the memory available before altering these queue sizes.

The following table gives the average and maximum memory used for different image sizes.

Antenna config	Image	Avg. cumulative memory used (GB)	Max. cumulative memory used (GB)
`lowdb2`	16k-8k-512	82	90
	32k-8k-1k	120	160
	64k-16k-1k	193	276
	96k-12k-1k	380	462
	128k-32k-2k	707	913
	256k-32k-2k	2626	2925
`midr5`	16k-8k-512	87	90
	32k-8k-1k	115	156
	64k-16k-1k	176	266
	96k-12k-1k	360	440
	128k-32k-2k	560	914
	256k-32k-2k	2437	2612

These tests are made using **30 nodes** with the following hardware on each compute node:

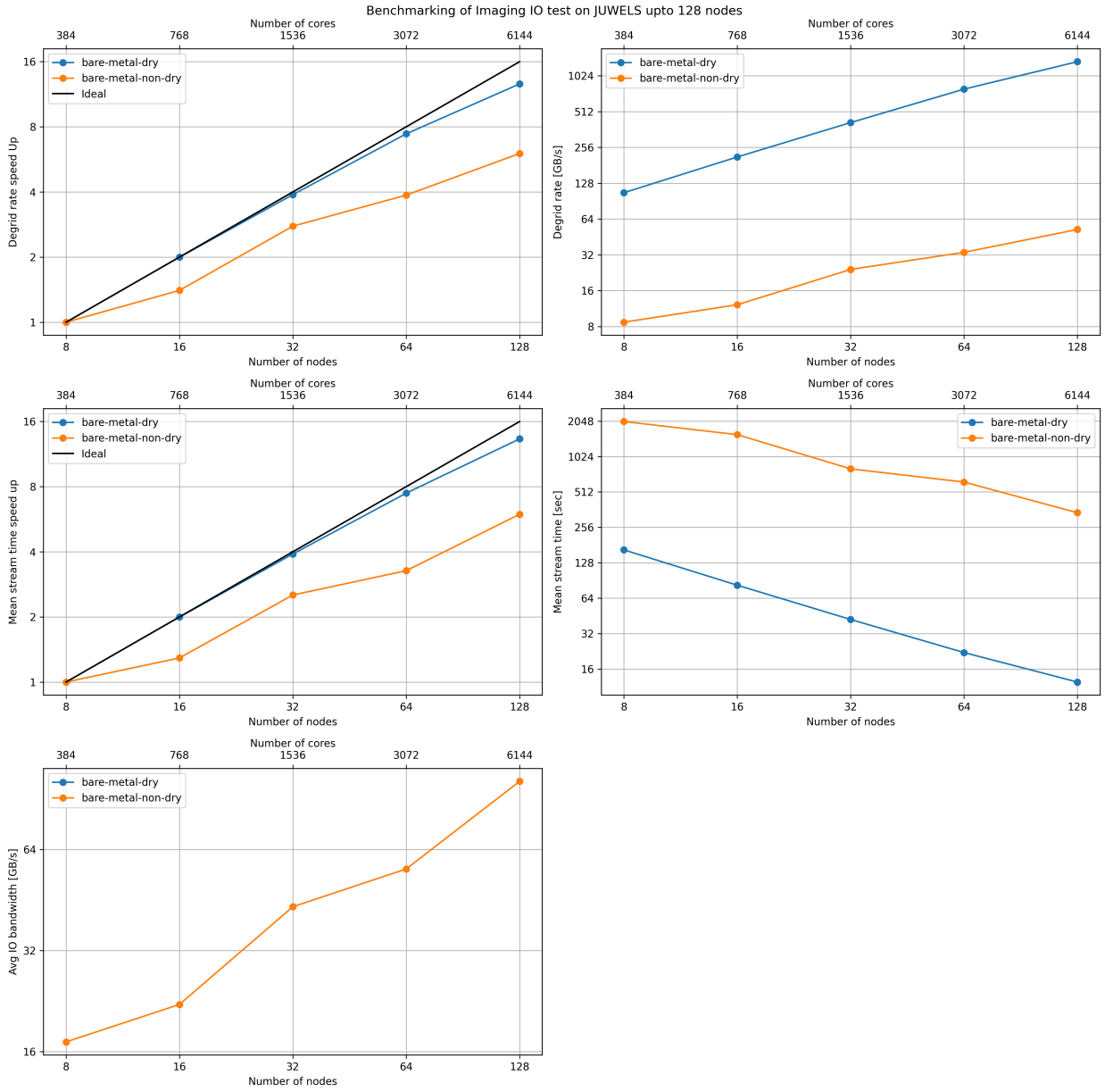
- 2 Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, 16 cores/CPU with hyperthreading enabled
- 192 GB RAM
- 1 x 10 Gb Ethernet, 1 x 100 Gb Omni-Path

Each run follows the configuration given in *SKA1 LOW and MID settings* for `lowbd2` and `midr5` settings. Number of facet workers and eventually, number of MPI processes are chosen according to the number of facets for each image that can be found in *Recombination parameters*.

These numbers are only approximate as they include the memory usage by the system resources as well. But this gives an idea of the memory requirements for different image sizes. We **only used dry runs**, *i.e.*, not writing visibility data to the disk, to obtain these numbers. In the non-dry runs we should take into account the visibility queues that will require additional memory. **Note** that as stated in *Recombination parameters*, `256k-32k-2k` is not suited to run for `lowbd2` configuration. The memory requirements are provided here only for the reference purposes.

7.2 Run times

The runtimes obtained from both dry and non-dry runs from the benchmarking tests on JUWELS are shown below.



This gives a reference run times of the benchmark code using 384 to 6144 cores. The actual run time is in the order of the mean stream time plus MPI start up and pre configuration overheads. No sever load balancing issues were observed for the runs. The above runtime values for the dry runs can be considered as a good reference for running the prototype.

Notice that the case of runtimes for non-dry runs heavily depend on the I/O bandwidth offered by the underlying parallel file system. **Care should be taken** when launching such runs as they can overload the file system cluster. In the case shown, we obtained an I/O bandwidth around 100 GB/s, where the prototype generated more than 32 TB of data. When running on the clusters that offers inferior throughputs, reservation time should be estimated accordingly based on the amount of data the prototype will generate and available I/O bandwidth. The approximate amount of data

that would be generated for different configurations are presented in *SKA1 LOW and MID settings*. It is also worth noting that the bigger chunk sizes result in more data. For instance, for the configuration used for JUWELS runs, we should expect a visibility data around 17 TB. But we ended up writing more than 32 TB of data because we used relatively bigger chunks of 1 MiB for these runs. The exact amount of data produced cannot be estimated *a priori*, but for the chunk size of 1 MiB, a factor of 2 seems to be a good estimation.

CHAPTER

EIGHT

KNOWN ISSUES

As already discussed throughout the documentation, there are some known issues in the benchmark. The important ones are

- It is imperative to use `--fork-writer` option to achieve high data throughput with multiple writers. This limitation comes from the global lock of the HDF5 libraries.
- The option `--fork-writer` does not work on Mac as `dispatch_semaphore` library used for the implementing locks on writers supports only threads. Typically, the writer processes hangs waiting for the lock to release. One possible solution to this is to use named semaphores from POSIX (MacOS supports only named Semaphores). But using named Semaphores gives us another limitation as typically the number of opened file descriptors are limited. To get around this issue we can increase the `ulimit -n` (till the value the kernel accepts) and/or decrease visibility queue size for the benchmark `--visibility-queue`.
- Often when running on MacOS, OpenMPI that is build with `homebrew` might throw some silent errors concerning vader BTL component. They might not be critical but to get rid of them include only `tcp` and `self` BTL components using `--mca btl tcp,self` argument.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- search