SDP Memo 102:
Distributed Predict I/O Prototype

Document Number SKA-TEL-SDP-0000203
Document Type ..... MEMO
Revision 1
Author ..... Peter Wortmann
Release Date 2019-03-29
Document Classification ..... Unrestricted
Status ..... Released

Lead Author	Designation	Affiliation
Peter Wortmann	Research Associate	University of Cambridge
Signature & Date: Peter Wortmann Peter Wortmann (Mar 29, 2019)		

Revision	Date of issue	Prepared by	Comments
1	2019-03-29	Peter Wortmann	First version: prepared to address OAR SDPCDR-79

SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

Table of Contents

1 Introduction **3**

2 Background **3**
- 2.1 Simplifications . . . 3
- 2.2 Technology . . . 4
- 2.3 Test Hardware . . . 4
- 2.4 Design . . . 5

3 Parametrisation **5**
- 3.1 Visibilities . . . 8
- 3.2 Image & Grid . . . 8
- 3.3 Work Assignment . . . 9
- 3.4 Queues, Processes and Threading . . . 10

4 Results **11**
- 4.1 Dry Rate . . . 11
- 4.2 Transfer Amounts . . . 12
- 4.3 Storage Rate . . . 12
- 4.4 Chunk Size . . . 13
- 4.5 Phases . . . 15
- 4.6 Accuracy . . . 17

5 Conclusions **17**

6 Future Work **17**
- 6.1 Scale Up . . . 17
- 6.2 Refactor . . . 19
- 6.3 Add Features . . . 19

List of Figures **20**

List of Tables **20**

References **20**

1 Introduction

The SKA Science Data Processor will involve high performance computing with particularly intense requirements in terms of internal data throughput. There are good reasons for this: at core, interferometry boils down to a Fourier Transform, with every visibility making a certain contribution to every output pixel. Both the number of visibilities and pixels grow quadratically with the size of the instrument and therefore achieved resolution. At the end of the day, this means that even with the most efficient algorithms, the size of an interferometer is likely limited by the capability of the system to support the data streams required to traverse from one domain to the other.

The software prototype developed for this memo is meant to evaluate hardware and software designs in the light of this assumed breaking point. The idea is to find out how well we can manage these challenges using software and hardware we can access today. Note that this prototype is still heavy work-in-progress at the time of writing, so this memo just represents a status report.

2 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.

2.1 Simplifications

In line with SDP assumptions about the most prominent workloads (Bolton et al., 2016) we will focus on time and frequency synthesis, i.e. de/grid data from different frequency channels and time stamps together. However, we will only consider one polarisation, Taylor term and snapshot (~ 8 minutes observation time). We especially ignore all correction terms for instrumental effects such as non-coplanarity, the beam pattern or gains, as well as environmental effects such as the ionosphere. This reduces the computational load significantly, and should allow us to simulate the I/O pressure exerted by full-scale "fat" SDP nodes using more readily available "thin" nodes.

We also focus on prediction – which in this context means generating expected visibilities from a sky image. Consequently, for this test buffer I/O will be about writing visibilities to

storage. This does not actually match the I/O patterns we would expect for the SDP, as only ingest components would typically write visibilities to storage (at a relatively slow rate). Imaging pipelines would then read the ingested visibilities back while subtracting the predicted visibilities in-memory. We believe that this does not reduce the meaningfulness of the test, as degridding is entirely dual to gridding. So we simply would have to "invert" all steps to obtain an imaging pipeline, with basically the same I/O characteristics.

Focusing on degridding brings other conveniences, for example it can be tested easily for correctness by comparing against direct Fourier Transform (DFT) results. Also a "read" benchmark would require another set-up program to generate input data, which we can skip for a "write" benchmark. Finally, note that writing visibilities to disk is strictly harder than reading, therefore any buffer I/O results obtained here are likely going to be strictly better in a real test. This is especially true because re-writing a chunk is considerably more expensive than reading the same chunk twice (as noted later in subsection 4.2).

Finally we will use a somewhat experimental recombination method that directly reconstructs high quality sub-grid approximations from image data. This allows us to skip phase rotation and makes baseline-dependent averaging optional, while minimising network traffic.

2.2 Technology

The point of this prototype is to figure out fundamental scaling limits. Therefore we will *not* use high-level execution frameworks for the moment. Instead, everything is built by hand using the most standard technologies available. The narrow focus explained in the last section makes software complexity manageable.

The prototype is written in plain C to minimise language environment as possible bottleneck. This should also make it easy to interface with low-level APIs, while also maintaining the flexibility of transplanting prototyped code into more high-level frameworks in future. For the moment we will use straight MPI for communication, as this provides high-throughput network communication out of the box at most HPC facilities. The prototype uses OpenMP for parallelism, with storage I/O getting handled by separate threads (pthread). HDF5 is used for data storage, using standard file system back-ends. This was the easiest option for getting started, but we might want to investigate other storage back-end options as well at some point.

Finally, we will use double-precision throughout. Use of single-precision generally improves performance by roughly a factor of 2, but it is unclear at this point in what places we can get away with it. So we stick with double precision to make things simpler for the moment.

2.3 Test Hardware

Most tests were performed on 8 nodes of the Performance Prototype Platform P3 (Taylor, 2018). The installed system provides us a small SLURM scheduler installation for executing the pipeline, as well as monitoring infrastructure based on Monasca, Apache Kafka and Grafana (Taylor and Szumski, 2018).

Each of the 8 compute node has 2 Intel Xeon E5-2683 v4 processors for 32 cores each (256 cores total), with 128 GiB RAM available (1 TiB total) and 100 Gb/s Infiniband connectivity. See Appendix 2 of Taylor (2018) for the exact hardware specifications. The system has 3 storage back-end setups, all realised using BeeGFS (Heichler, 2014):

1. Single storage server, using 4 NVME drives (12 TB total)
2. Distributed across two storage servers, using 24 SATA connected SSD drives each (67 TB usable).

3. Distributed storage of 4 SATA connected SSDs per worker node (8.8 TB usable)

We have configured all storage back-ends to use the maximum amount of striping, as that tends to result in the best performance, and prevents problems due to imbalances between storage devices. In tests all three configurations have shown similar performance, supporting between 800 and 1000 MB/s write rate per node for reasonable chunk sizes and thread configurations (Taylor, 2018). This represents roughly a fifth of the per-node data rate assumed to be required for full-scale SDP operation (Bolton et al., 2016).

According to the Geekbench 3 benchmarking tool the CPUs achieve 166.2 Gflop/s, which at roughly 400 operations per visibility this means a node can emit 415.5 Mvis/s or 6.65 GB/s using trivial gridding. This means that with the chosen constraints we ought to be able to saturate the I/O capability of the system as long we manage distribution correctly.

2.4 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 in Figure 1.

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[1].

3 Parametrisation

Whether or not the prototype will work depends to a large part on choosing the right parameters. After all, every pipeline has different ways in which it can become expensive in terms of data consumed, data produced or computation performed. We need to make sure that we pick the right point in the parameter space that will actually tell us something about how the final SKA SDP might perform.

While this will typically involve adopting projected SKA SDP parameters – such as using actual telescope layouts and parameters from the parametric model – we will also need to make adjustments and extrapolations. This is mainly because of the restrictions in the software design and limits of the used test hardware. We will attempt to argue case-by-case for why the chosen parameters might still allow us to learn something about the full SDP system.

[1] Note that we assume no baseline-dependent averaging for the moment, which would counter-act this effect.

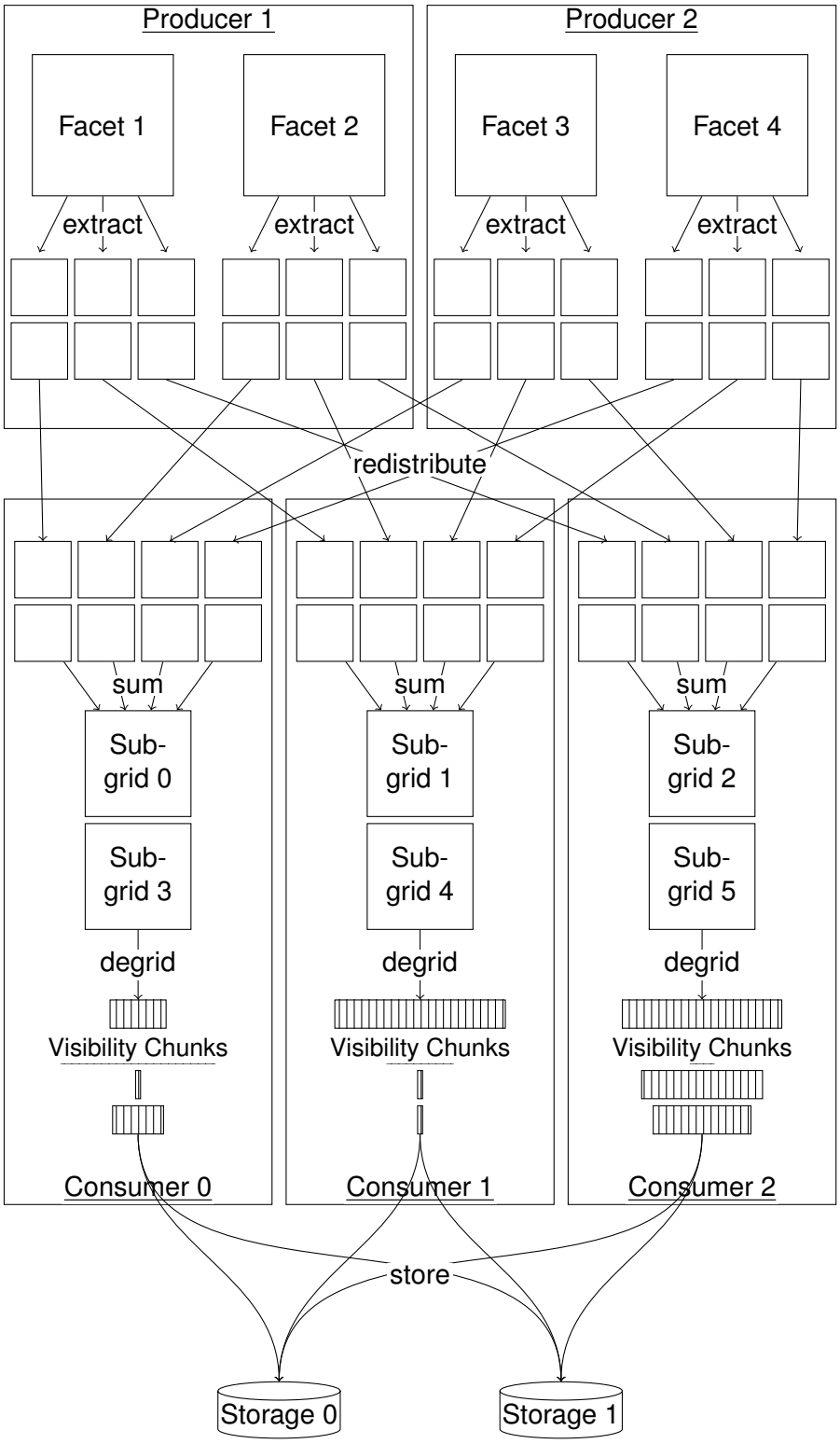

Figure 1: Prototype Design

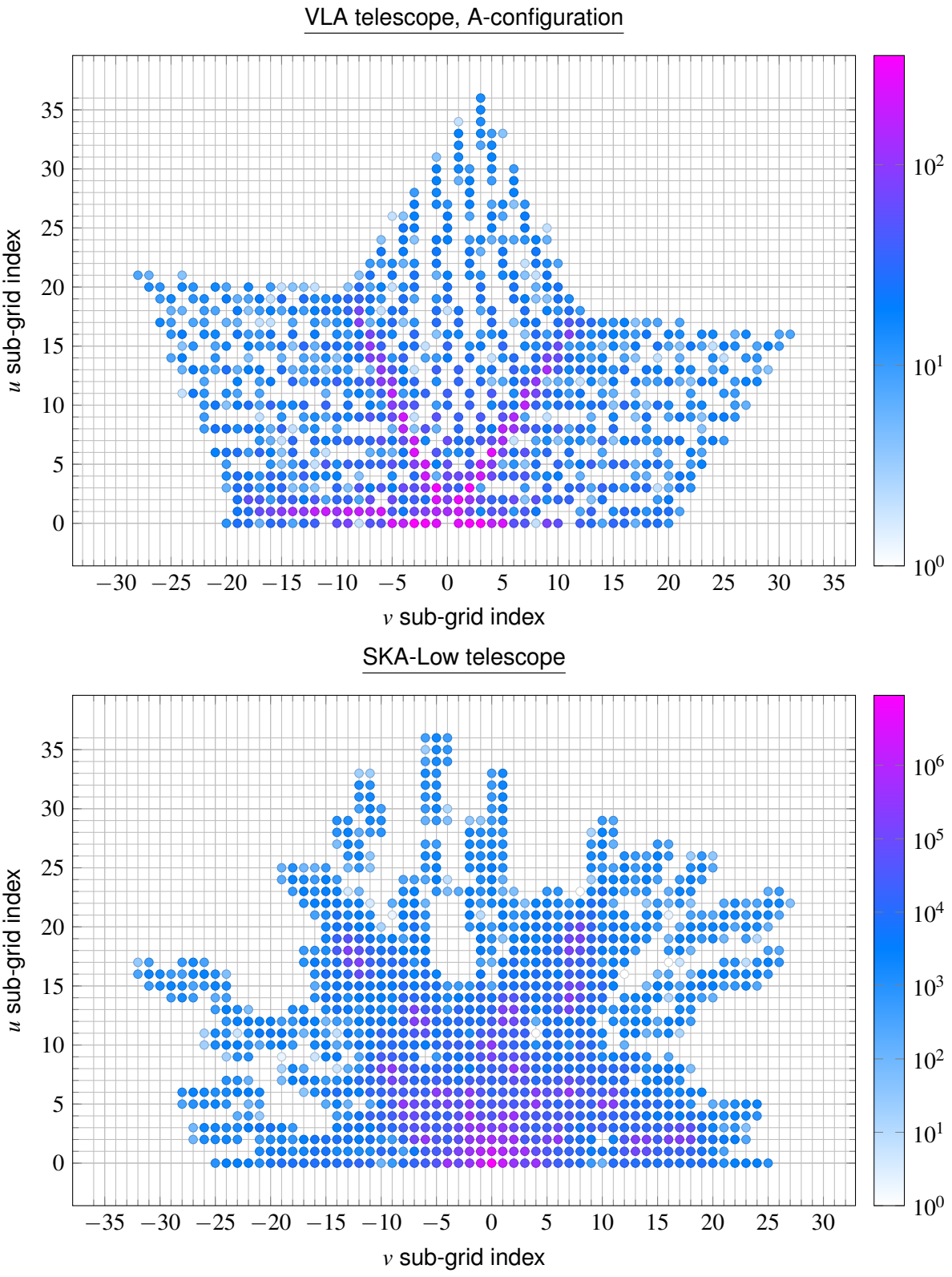

Figure 2: Baseline chunks (here ∼4096 visibilities) overlapping sub-grids

Image		Facet			Subgrid		Image	Network
Size	Usable	Size	Padded	FFT	Size	FFT	Data	Overhead
16 384	11 468	5 120	5 632	8 192	416	512	9.8 GB	35.4%
65 536	45 875	10 240	11 264	16 384	832	1024	157.8 GB	35.4%
98 304	68 812	15 360	16 896	24 576	832	1024	353.3 GB	46.7%
131 072	91 750	20 480	22 528	32 768	1856	2048	628.1 GB	21.4%
262 144	183 500	20 480	22 528	32 768	1664	2048	2512.6 GB	35.4%

Table 1: Test facet/subgrid sizings, with image data sizes (30% margin) and overhead

3.1 Visibilities

The benchmark uses actual telescope layouts as starting points. As we do not support correction for non-coplanarity, we assume that the telescope is entirely coplanar with respect to the phase centre for the entire observation time. In practical terms, this would correspond to a telescope that was "flattened" with respect to earth curvature, deployed at one of the earth's poles and looking straight towards the zenith. There are two telescope configurations provided, as shown in Figure 2: The VLA(A) configuration is implemented as a cheap test parameter set. SKA1-Low is used as the main configuration for at-scale tests. Note that the SKA1-LOW layout has a strong concentration of baselines near the centre, which must be counter-balanced by work assignment (see subsection 3.3).

By default, visibility data will be generated for a snapshot of 460 seconds (512 time steps assuming a dump rate of 0.9 s), and covering a frequency range of 30 MHz (270 to 300 MHz in 6144 channels). SKA1-Low will have 512 antennas and therefore 130 816 baselines, which means that we end up with 411.51 billion visibilities, or 6.58 TB of storage data. Compared to a full 6 hour SKA1-Low observation, this represents 2.13% of the time, and 10% of the full frequency band. So collectively this is 0.21% of a full observation, which we will run on an equivalent of 0.6% of the SDP batch compute nodes (8 instead of $\sim$1200 nodes). Therefore we have (very) roughly the right order of magnitude here to properly stress our system.

In terms of sub-grids, this yields us a coverage of about 20.9% of the grid. Note that we only cover one half of the grid, as the other is redundant due to Hermitian symmetry. The remaining 58.2% we lose due to some margins (light over-sampling of the image) as well as the "spikiness" due to the relatively short snapshot.

3.2 Image & Grid

Normally we should size the image size based on the telescope's primary beam or field of view (in relation to baseline length and frequency range). For the purpose of this scaling test, we leave it variable so that we can dynamically adjust the load. To maintain a certain precision level, we need to pad both facets and subgrids, while maintaining fast power-of-2 (or 3) sizes for Fast Fourier Transforms. This means that there is only a limited number of suitable parametrisations. In Table 1 we list configurations supported by our prototype at the 10^{-5} precision level.

While increasing the scale introduces extra work, the main limiting factor is node memory: to support continuous streaming, the total image data must fit into the collective RAM of all involved nodes. Note that we assume margins of 30% of all images to be zero to ensure precision of the grid convolution function. As the subgrid reconstruction is linear, we simply skip storing facets in this margin region, it is not part of the "image data" calculation. As the 8 nodes of our test set-up can collectively store about 1 TB of data, this means that the largest supported "usable" image size is about 91 750 pixels on a side.

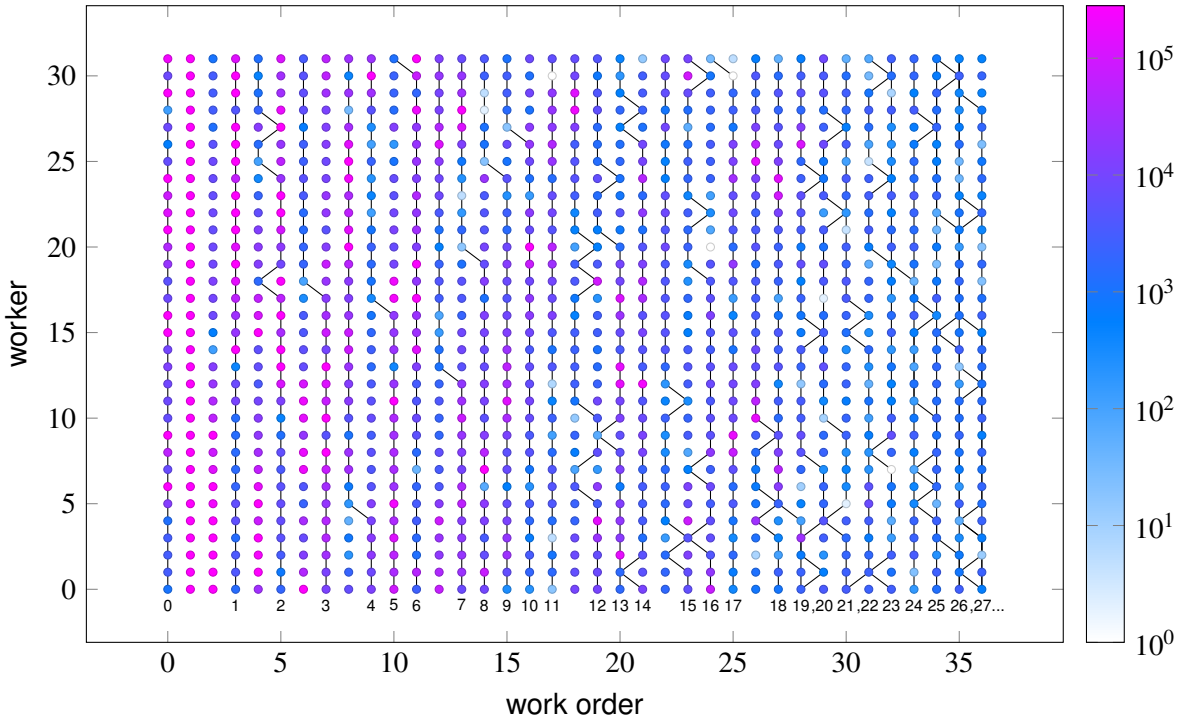

Figure 3: Work assignment for SKA1-Low, 32 workers. Lines separate subgrid columns (u).

To put this in perspective, note that for SKA1 Low (the layout of which we are using) the maximum baseline length is 65 km or 75885.8 wavelengths at 350 MHz, so this configuration is more than enough to image the entire sky ($\theta \geq 1$). The larger scales are actually more interesting for SKA1 Mid configurations, as we will be dealing with baselines of up to 150 km, which at a wavelength of 15.4 GHz (upper range of Mid5b band) works out as 7.7 million wavelengths. This means that even the largest configuration from Table 1 can only cover about 1.36 square degrees of sky (which is about the beam size to the third null). So we will effectively attempt to use the more complex SKA1 Low baseline layout (with 512 antennas) at scales approaching SKA1 Mid (with larger fields of view relative to the observed frequency range).

Finally let us quickly consider the required network communication. Padding also introduces some network overhead in this context as indicated in Table 1. Note however that we can skip roughly 49% of the image due to margins and all but 22.2% of the grid due to the layout and Hermitian symmetry, which amounts to only about 10.9% of total network transfers. We will see in the next section that to generate a fair work assignment we will replicate central sub-grids to multiple workers, but thanks to the overall data reduction this can be afforded easily (and we might at some point extend it to allow dynamic work balancing).

3.3 Work Assignment

In order for the distribution scheme to work effectively, work must be divided up between workers in such a way that:

- Streamer processes have roughly the same amount of total work, and therefore finish around the same time.
- Producers can batch subgrid data generation efficiently. We preferably want to work on "columns" of subgrids, as those can be generated from common intermediate data.

The implemented solution is to assign work to workers using a round-robin scheme walking through the grid in column-major order. Note that in Figure 2 we plot v on the y-axis for layout reasons, therefore column-major for this graph means walking horizontally first, then vertically.

To improve the balance, the prototype then "swaps" work items until all streamer processes have roughly the same amount of total work. This somewhat trivial method manages to even out the worst imbalances. For the SKA1 Low configuration this leads to the work assignment to 32 workers as shown in Figure 3: Each worker ends up with 1 208 637 to 1 208 663 visibility chunks. Note that we start with the most "expensive" column of subgrids centred around $v = 0$ in order to generate streamer work quickly right out of the gate. Also note that the middle subgrid at $u = v = 0$ has been split into 32 chunks of equal size (second column in Figure 3), as it would otherwise be basically impossible to even out the resource assignment.

This is still a pretty naïve approach. We only balance the total amount of work, and do not account for the fact that some workers might run ahead in terms of column getting worked on. Furthermore, it is quite optimistic to attempt to fix work order in advance, as small fluctuations in performance (especially of storage back-ends) almost always require some dynamic re-scheduling towards the end.

3.4 Queues, Processes and Threading

The prototype pipeline has quite a few steps running mostly in parallel:

1. Producer Process:
   (a) Subgrid extraction + MPI send
2. Streamer Process:
   (a) MPI receive + subgrid generation
   (b) Degridding
   (c) Write to storage

The epectation is that most of these will be I/O bound by the last step, and therefore must be connected by limited-size queues in order to exert back-pressure and prevent running out of memory. Queue sizes are given in Table 2 (assuming the configuration with the 128ki/32ki/2ki FFT sizes from Table 1). Note that while the buffer sizes might seem quite generous, these will only last a couple of seconds at the targetted transfer speeds.

Currently the only pipeline stage that we can not parallelise is the final step: For writing to storage we create one file per process, and can not distribute the write access with HDF5 (sadly, Parallel HDF5 does not seem to be of help here). This means that the number of processes needs to be chosen such that the storage back-end can keep up with writing visibility chunks to storage. As established in Taylor (2018), we need at least 4 threads per node in order to realise the maximum write rate (which was confirmed independently with our prototype); therefore we will run 4 "streamer" processes/ranks per node. "Producer" processes/ranks have much less computational work (but a lot more memory residency), and therefore are generally only assigned 1 or 2 ranks per node.

In terms of threads/cores, we need to reserve the maximum computational capacity for degridding work, as this is the only significant computational bottleneck. This means that for a hardware configuration with 32 available cores, we allocate 8 threads/cores to each "streamer" process/rank. Remaining functionality – which includes all threads of "producer" processes, network. storage and statistic threads – will run in secondary threads, as they are expected to be idle most of the time.

Queue	Length	Size	
MPI Send	32 sends/thread	1.01 GB	
MPI Receive	32 subgrids/process	2.06 GB	
Subgrids	32 subgrids/process	2.15 GB	
Visibility chunks	2048 chunks/process	0.54 GB	(for 128×128 chunks)

Table 2: Queue sizes used

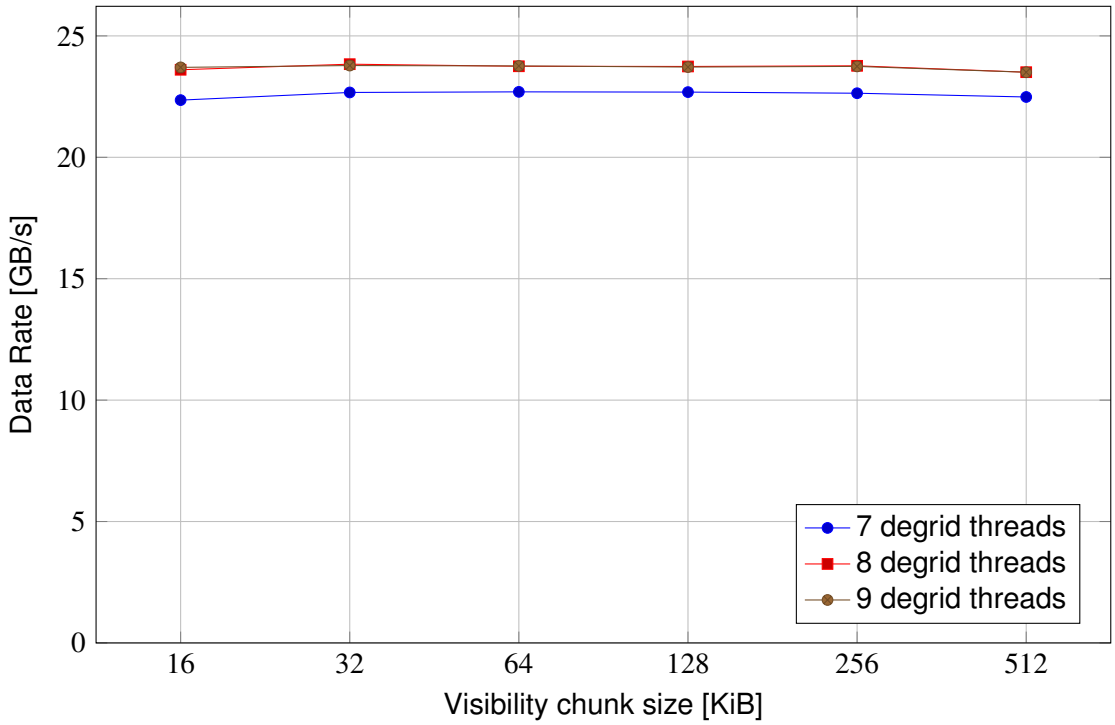

Figure 4: Visibility production rate from 8 nodes, 32 processes (dry runs)

4 Results

4.1 Dry Rate

Figure 4 shows the visibility production rate for "dry runs" without storage access, for different number of degrid worker threads and visibility chunk sizes. This was calculated by dividing the total amount of produced visibilities by the average "stream" time of streamer processes. This excludes time taken to find the work assignment, as well as the time required to populate the HDF5 file with all the baseline groups/datasets. Taking the average time across workers is also slightly optimistic – we are basically assuming that remaining work imbalances are something that we will be able to resolve eventually.

We can observe two effects: It is indeed optimal to match the number of degrid workers to the number of total available cores (with 4 processes per 32-core node this means 8 threads). Clearly having just 7 threads reduces performance less than $\frac{1}{8}$, yet it is also clear that with the chosen configuration there is not enough non-degridding work to keep an entire core busy.

Furthermore, chunk size slightly affects performance: Both very small and very large chunks cause small reductions in data rate. This is likely because for small chunks we start to see internal overhead take a toll (such as synchronisation on queues), and for very large chunks

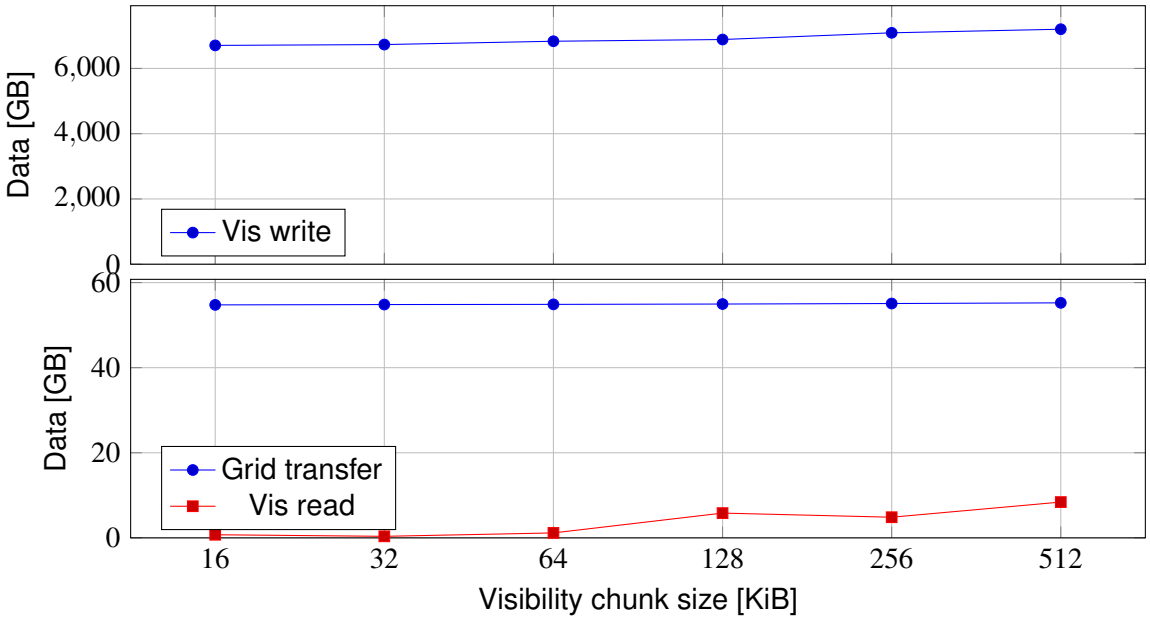

Figure 5: Data amounts transferred

the case where a chunk only partly overlaps a subgrid starts causing more overhead (see also Figure 5).

4.2 Transfer Amounts

Shown in Figure 5 is the total amount of data transfered in a "full" run involving writing to storage. This is constant irrespective of storage back-end, but changes with the chunk size as coarser visibility chunks overlap more subgrids. This can either mean that some chunks get written by multiple workers – or that a chunk gets re-written by a worker. This is why bigger chunk sizes also lead to more visibilities getting read back.

The amount of (sub-)grid data that needs to be exchanged also changes slightly due to a similar effect, but this does not really matter as is evident in Figure 5. Note that for this test $91\,750 \times 91\,750$ pixels of usable image data where exchanged, corresponding to 134 GB of data (or effectively 67 GB due to images being real / grid hermitian-symmetric). This is significantly less than the 628 GB of resident "image data" listed in Table 1 for our configuration, as that includes significant padding. There are a number of positive and negative effects competing here – as mentioned we require some padding, but can skip known-zero/uninteresting parts of the image/grid, and finally need to broadcast the central sub-grid (see subsection 3.3). In this case, we end up with approximately the same information density as we started with, which is encouraging.

4.3 Storage Rate

Effective storage write rates for different back-end configurations is illustrated in Figure 6. As previously, this is calculated by dividing the total amount of produced data by the average stream time per stream worker. It should be noted that storage rates are substantially below the "dry" production rate shown in Figure 4, so as expected the pipeline is chiefly I/O bound writing to storage.

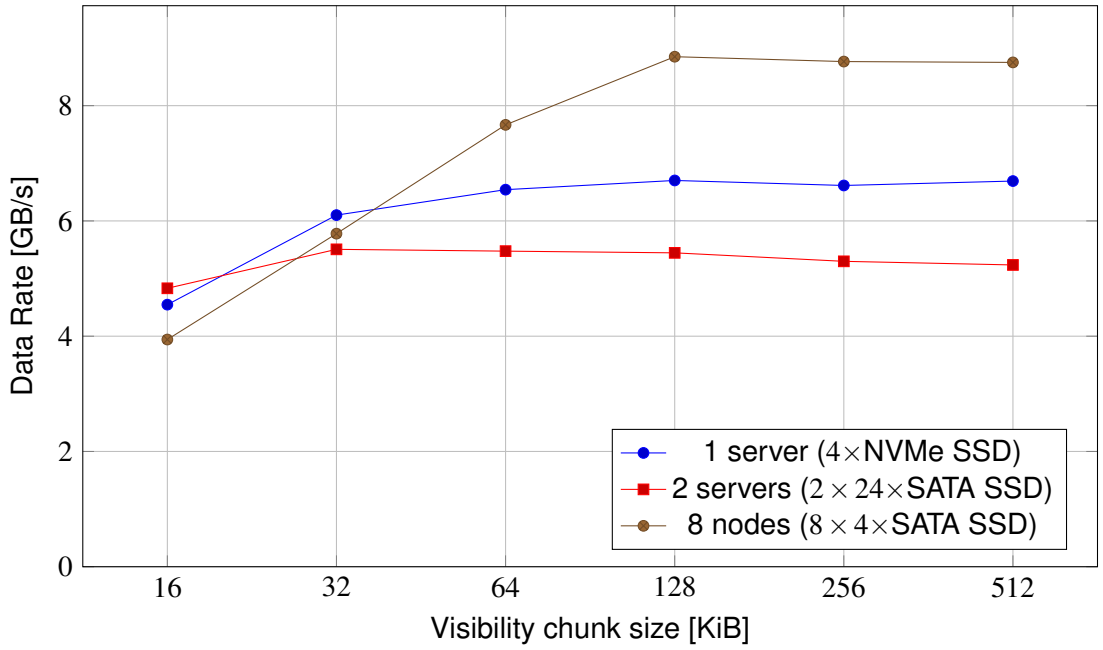

Figure 6: Effective visibility write rate per storage configuration

Consequently performance also depends strongly on the storage implementation as well as the visibility chunk size used. Note that in contrast to Taylor (2018) we find significant differences in speed between the three storage configurations. In fact, the write speed achieved here for the hyperconverged (8 nodes) case seems to be significantly above 1 GB/s/node, which is actually faster than achieved in the synthetic benchmark. This might be due to caching, even though it is unclear why this would also favour the 1-server (NVMe SSD) configuration over the 2 server (SATA SSD) configuration.

4.4 Chunk Size

Beyond the maximum throughput, different storage back-ends also seem to react somewhat differently to changing the visibility chunk size. In Figure 6 we see that small blocks lead to significant reductions in throughput, with the hyperconverged 8-nodes configuration suffering both the earliest and – eventually – the most: With 16 KiB blocks, performance of the back-end has dropped by more than half.

Note that this might simply be a CPU contention issue: Figure 7 shows the (user) CPU usage across all 8 involved worker nodes. Clearly small chunks introduce significant amount of CPU load, no matter the configuration. In fact, for most chunk sizes up to 128 KiB the CPU seems to be fully utilised – and this despite the effective visibility rate being way below the “dry” rate. It is not quite clear how this happens, given that there is only one storage writer thread per process.

On the other hand, this gives us a reasonable explanation for why the hyperconverged storage configuration falls off quickest: On some level the storage implementation will start competing with the benchmark for CPU time, resulting in heavy performance losses. A similar effect might be limiting the 1 server (NVMe) configuration, as we observed very close to 50% system CPU load on its system while running the benchmark. As we might reasonably expect small chunks to exert load both on the client *and* the server, this fits nicely into the general explanation framework.

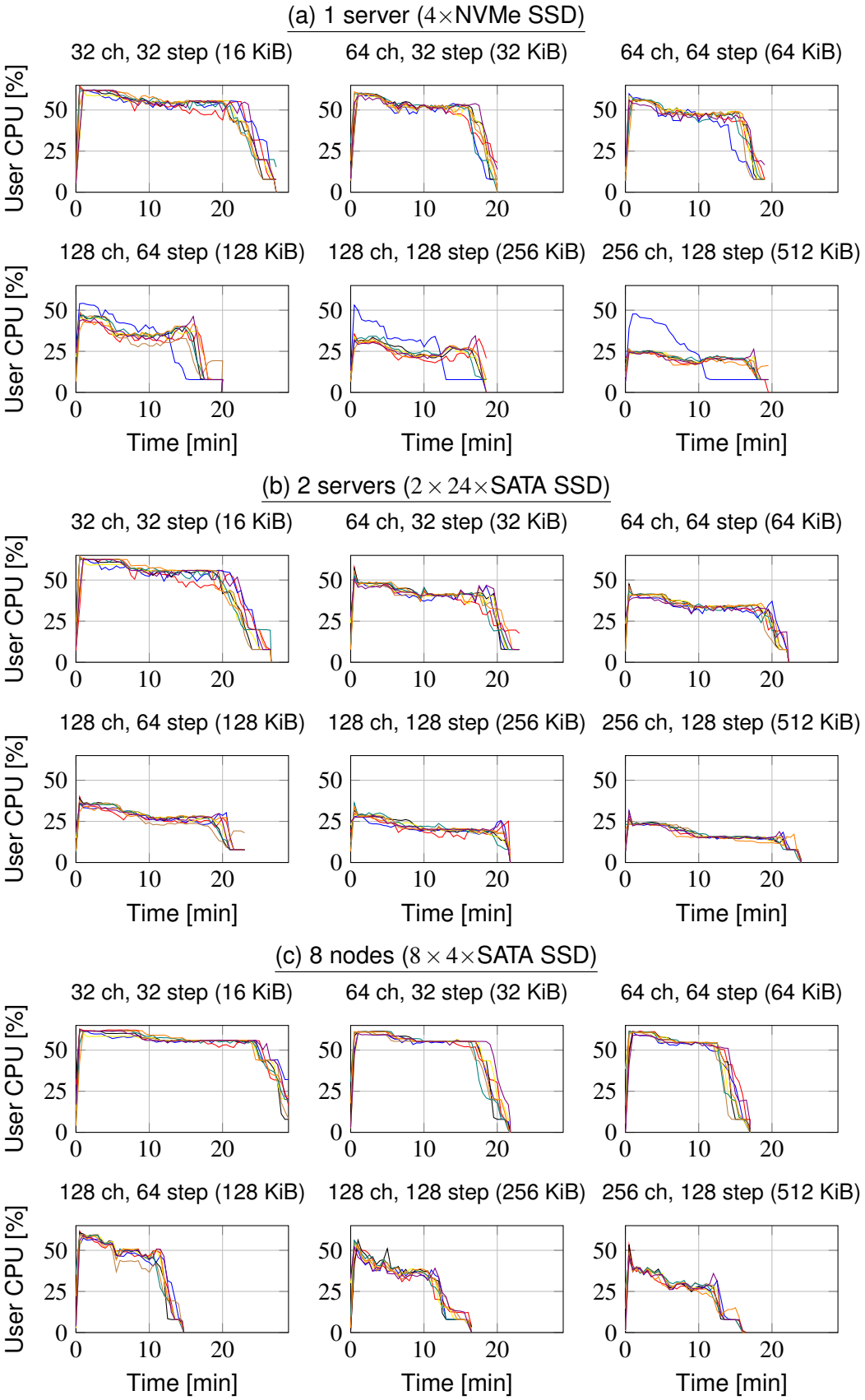

Figure 7: User CPU usage depending on visibility chunk size

Finally note that for large chunk sizes in the 1-server (NVMe) configuration, one node runs ahead and does a lot more work before finishing early. This is because this one node is more local to the storage server in terms of network topology, and therefore seems to be able to push its data to storage more easily than others. As the entire pipeline is bound by the back-pressure of storage, this translates into overall faster progress. It is not quite clear whether – or how – we should address this type of issue.

4.5 Phases

Further to Figure 7 we can also look at the storage device throughput measured on the storage nodes, shown in Figure 8. For interpreting the y-axis keep in mind that that the graph tracks:

- 8 individual ports for the 1 server ($4\times$NVMe SSD) configuration,
- 2 SSD arrays for the 2 servers ($2 \times 24\times$SATA SSD) configuration, and
- 8 SSD arrays for the 8 nodes ($8 \times 4\times$SATA SSD) configuration/

Therefore the total throughput is the average of the graph multiplied by 8, 2, and 8 respectively. Note that this is indeed consistent with Figure 6.

By looking at the data we can discern three distinct phases:

1. **Start phase**: There is always somewhat of a "hump" in CPU usage at the start of the run. This is especially distinct for the runs with small chunk size, where CPU usage almost reaches 60% in some cases. This is the time where the "producer" processes are still actively working on generating sub-grid data. Note that due to the chosen work order, the delay between initialisation and degridding work starting is mostly insignificant.

   Because of to the relatively generous queue sizes this phase currently seldom lasts longer than about 20% of the run. Note that this does not correspond to a noticeable dip in storage write speed, even for configurations that look CPU bound. We can therefore speculate that sub-grid recombination co-exists well with on-going gridding and storage work on hyper-threading CPUs.

2. **Work phase:** Once all sub-grid data has been loaded into the appropriate queues, the remaining work is mostly about doing degridding and writing to storage. The per-visibility amount of work we need to spend on each is constant, but as noted previously the balance changes depending on chunk size.

   Looking at storage throughput, for the three configurations we reach the maximum possible rate at roughly 64 KiB, 32 KiB and 128 KiB respectively. This roughly corresponds to the maximum achieved rate in Figure 6, which might suggest that the main bottleneck sits on the client side. Also note that while the disaggregated storage configurations (a) and (b) remain very steady throughout the run, the hyperconverged storage configuration (c) shows a lot more "noise". This might be due to competition for CPU resources. It is unclear how much efficiency gets lost due to this effect.

3. **End phase:** Finally, due to the static work assignment there is a significant phase at the end where some workers have run out of work while others are still writing visibilities. This results in both overall CPU usage and storage throughput to peter out towards the end. There are especially unfortunate cases like in configurations (a) and (b) for chunk sizes 128 KiB and 512 KiB, where we get a small "tail" to otherwise efficient runs. This will likely require further work.

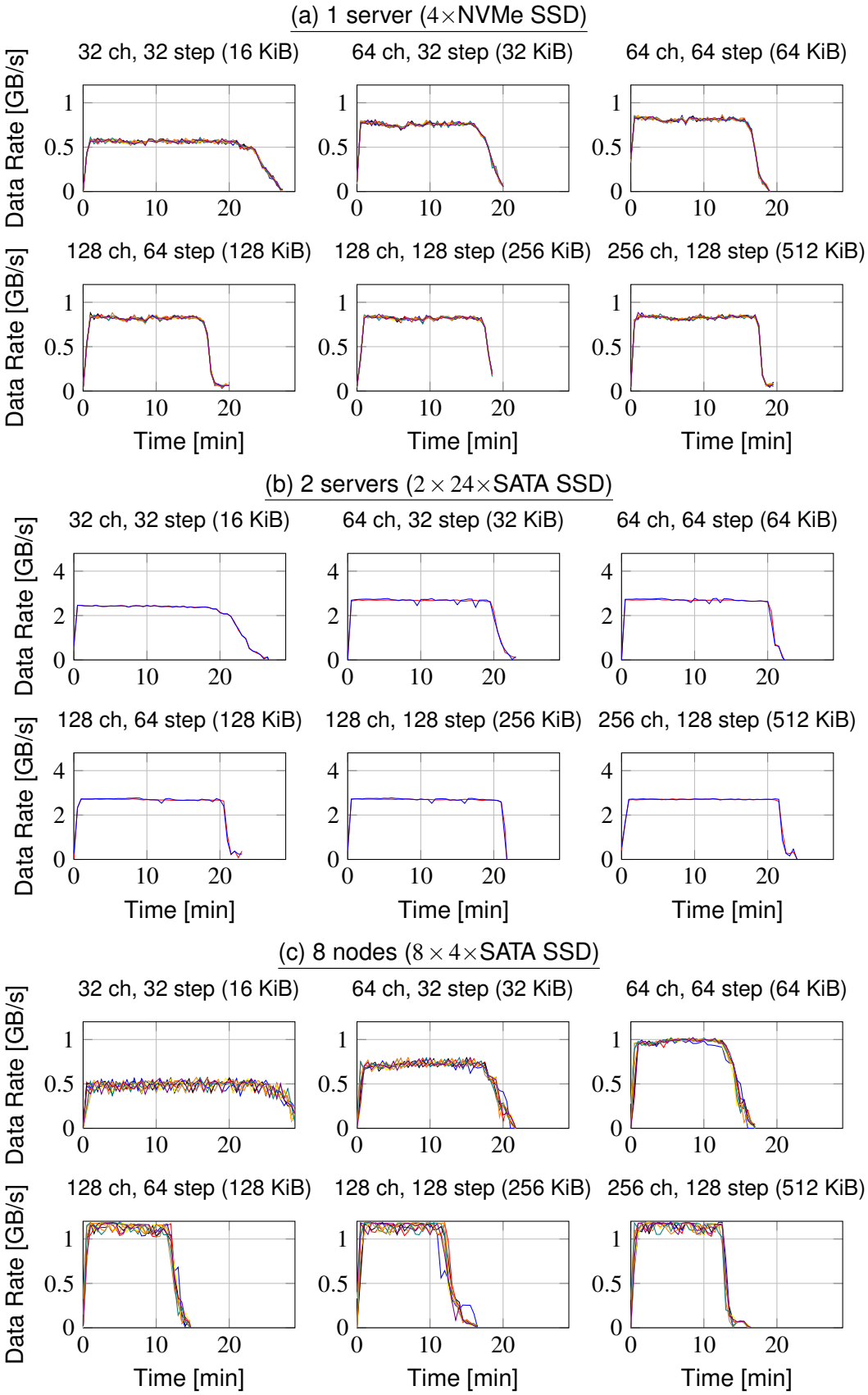

Figure 8: Storage write speed depending on visibility chunk size

4.6 Accuracy

One of the upsides of focusing on prediction is that we can easily check the pipeline for accuracy: If we use a sky image that is zero outside of a number of "source" pixels, we can directly calculate the contribution of every source to every visibility (DFT).

This method is relatively expensive, mainly because of the sine and cosine calculations involved. Therefore we only check about every 8 000th visibility (pseudo-randomly chosen). This is rare enough to make the overhead negligible, but still typically results in more than 1 000 samples taken per process per second. The error is normalised according to the maximum possible absolute value of a visibility – the sum of source intensities (all 1 in this case) divided by number of grid cells / image pixels.

For 10 randomly placed sources, the worst root mean square error (RMSE) aggregated per worker is typically about $9.6 \cdot 10^{-6}$, and the overall worst observed error around $4.2 \cdot 10^{-5}$. This accuracy loss is in line with the chosen gridding function, with sub-grid recombination not introducing much extra error. However note that we were only able to get this much accuracy easily because of restricting ourselves to the coplanar case. Introducing w-projection or Image Domain Gridding would introduce both extra cost and often more accuracy loss.

5 Conclusions

So far work on this prototype has been rather successful given its goals: we have managed to produce a rather small test ($\sim$ 6000 lines total) that stresses state-of-the-art compute platforms in a way that is arguably similar to how an SDP pipeline would work. And despite the rather pessimistic picture drawn in Wortmann (2017), it looks like with appropriate care the prediction & imaging distribution problem is tractable, possibly even way beyond SKA1 scale.

In the grand scheme of things, this is encouraging for the long-term prospects of the SKA. At the same time it must be mentioned that we have not addressed the "elephant in the room", which is calibration. In fact, the chosen distribution scheme (splitting visibilities by uv-region) will likely make it actively harder to obtain calibration solutions using conventional algorithms, which typically work with strong locality in time and/or frequency. The question of whether we can adapt this software design to calibration is still very much an open research question.

6 Future Work

As noted in the introduction, this is a work-in-progress, and there are multiple avenues we would like to explore in the near future. The following sub-sections explore three ways we could proceed with this work.

6.1 Scale Up

One of the obvious ways forward is to run the prototype on larger systems, especially ones optimised for storage throughput. As documented in Taylor (2018), such a data accelerator (DAC) system has been installed at Cambridge's CSD3 system and was recently made available for these tests thanks to excellent support by the Stack HPC and Cambridge Research Computing Services team. This system can achieve about 240 GiB/s write performance, more than an order of magnitude more than the P3 system tested so far.

As shown in Figure 9 and Figure 10, preliminary tests are encouraging: This is distributing across 32 nodes, running the maximum (SKA1-Mid-equivalent) size configuration from Table 1

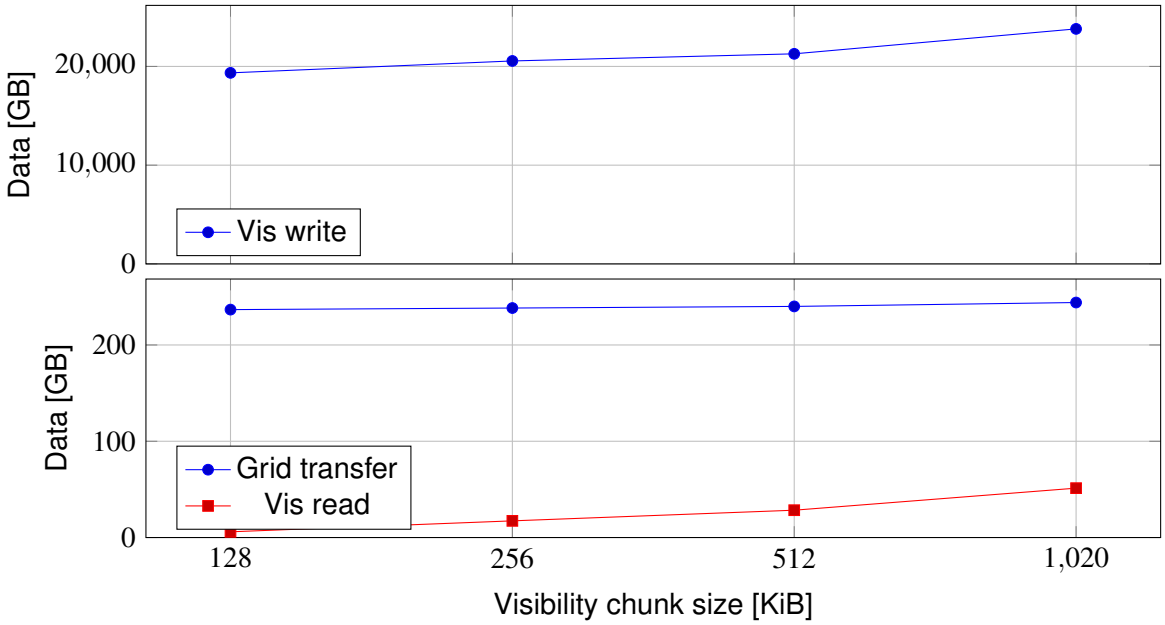

Figure 9: Data amounts transferred (CSD3/DAC)

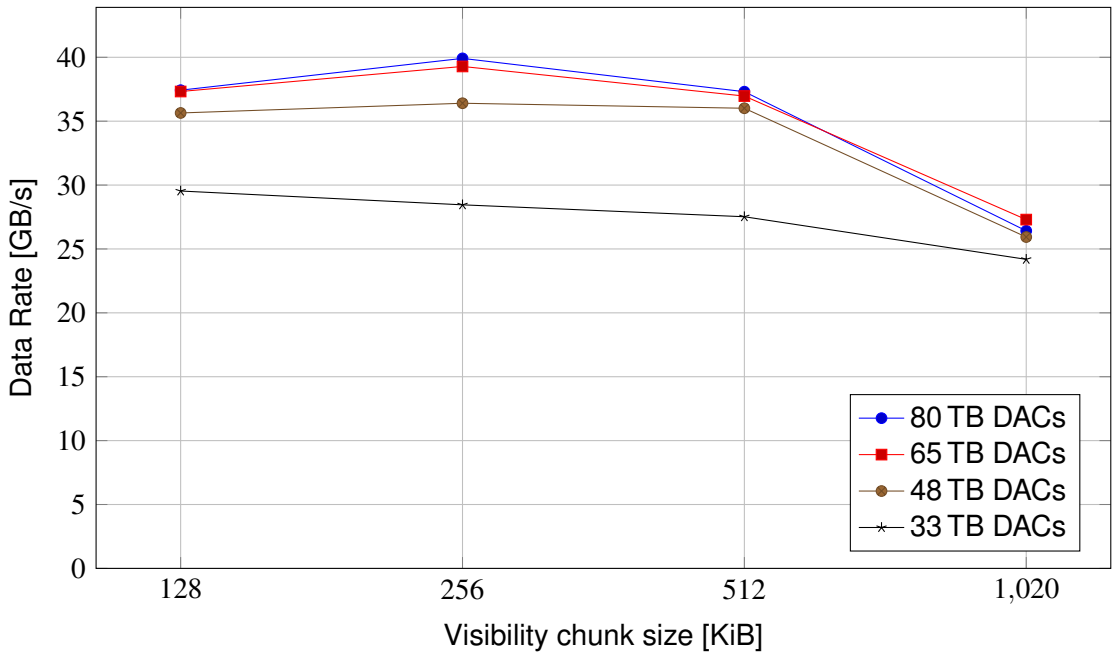

Figure 10: Effective visibility write rate per storage configuration (CSD3/DAC)

at the expected speed (dry visibility rate around 80 GB/s). We have so far managed to realise up to 40 GB/s effective write speed to the DAC storage servers. Note how adding more DAC capacity (= involving more DAC servers) increases the throughput. Interestingly enough, using 1 MiB chunks does not look like a good choice despite the Lustre configuration suggesting otherwise. On the software side, it strongly looks like we might have to adjust the prototype design to accommodate multiple writer threads per process in order to keep up with high-speed storage back-ends.

6.2 Refactor

Another high-priority goal is to bring the prototype more in line with the SDP software architecture. As noted in subsection 2.2, we are not using a dedicated execution engine here, and have also neglected to split out processing components. This is acceptable, because this was not the goal of this prototype. Nevertheless, we might want to move into that direction, and therefore away from MPI and OpenMP – not just to validate the SDP architecture, but also to ensure that we can grow the prototype's functionality further without hitting resistance.

There are multiple good candidates here. The obvious choice given the prototype's goals might be an execution framework like StarPU (Augonnet et al., 2011), which would especially allow us to easily migrate towards doing gridding work on accelerators. It is however not clear whether we could achieve similar guarantees with respect to total memory consumption without writing our own scheduler.

Furthermore, we might attempt adapting the prototype to more high-level execution frameworks in use by SDP, such as Dask, Apache Spark or DALiuGE. The main challenge is likely going to be to increase data throughput enough to achieve similar rates as the hand-coded OpenMPI solution. As noted in Li et al. (2018) for Apache Spark, there seem to be data distribution patterns that can not be implemented well in certain execution frameworks. Involving external tools such as Alluxio will likely introduce significant cost in terms of code complexity and overhead. Whether a satisfactory middle ground can be found remains to be seen.

6.3 Add Features

Clearly we would also like to work towards making the pipeline more representative. There are two major areas where functionality could be extended:

1. Dealing with non-coplanarity. As should be evident by the number of times it came up throughout the memo, removing this simplification would be a large step towards making this test more realistic. Unfortunately this would also add a lot of computational complexity to gridding, to the point where access to accelerators might be essential. Currently neither P3 nor CSD3 can reportedly offer the kind of environment where we have access both to enough accelerators and a fast storage implementation.

   We will nevertheless attempt to move into this direction. After all – even though it is likely not of immediate concern – dealing with memory transfers from and to accelerators is another I/O challenge we might want to gather experience with.

2. Implementation of the "backwards" step in order to build a proper imaging pipeline. This is clearly required if we want the prototype to actually behave like a proper SDP pipeleline in terms of inputs and outputs. However, this would add a lot of code complexity for relatively little insight, so it should not be seen as the highest priority.

Ideas we would *not* consider right now are adding in further axes such as polarisation, Taylor terms or other frequency handling, due to similar code complexity considerations.

We would also not see the point of abandoning double precision at this time, as most truly large data objects handled here (images, sub-grid contributions) are likely quite sensitive in terms of precision. It could quite sensibly be argued that (de)gridding kernels and visibilities should rather use single precision, however for current technology this would not actually move the "computational load per storage throughput" ratio much, so there seems little point to it.

List of Figures

1 Prototype Design . . . 6
2 Baseline chunks . . . 7
3 Work assignment for SKA1-Low . . . 9
4 Visibility production rate from 8 nodes, 32 processes (dry runs) . . . 11
5 Data amounts transferred . . . 12
6 Effective visibility write rate per storage configuration . . . 13
7 User CPU usage depending on visibility chunk size . . . 14
8 Storage write speed depending on visibility chunk size . . . 16
9 Data amounts transferred (CSD3/DAC) . . . 18
10 Effective visibility write rate per storage configuration (CSD3/DAC) . . . 18

List of Tables

1 Test facet/subgrid sizings, with image data sizes (30% margin) and overhead . 8
2 Queue sizes used . . . 11

References

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience, 23, 187–198, 2011.

Bolton, R. et al.: Parametric models of SDP compute requirements, Tech. Rep. SKA-TEL-SDP-0000040, SDP Consortium, dated 2015-03-24, 2016.

Heichler, J.: An introduction to BeeGFS, https://www.raidinc.com/wp-content/uploads/2016/09/Introduction-to-BeeGFS.pdf, 2014.

Li, Q., Wang, W., and Luo, Y.: Modeling and Evaluating the IO of MID1 ICAL Pipeline on Spark, Tech. Rep. SKA-TEL-SDP-0000143, SDP memo 059, SDP Consortium, revision DRAFT, 2018.

Taylor, J.: Overview of Buffer Prototyping and Modelling, Tech. Rep. SKA-TEL-SDP-0000126, SDP memo 045, SDP Consortium, revision 2, 2018.

Taylor, J. and Szumski, D.: Performance Prototype Platform (P3-ALaSKA) Monitoring & Logging, Tech. Rep. SKA-TEL-SDP-0000165, SDP memo 068, SDP Consortium, revision 01, 2018.

Wortmann, P.: Pipeline Working Sets and Communication, Tech. Rep. SKA-TEL-SDP-0000124, SDP memo 038, SDP Consortium, revision 2, 2017.