SDP Memo 56: Hardware Scaling Co-Design Recommendations

Document number……SDP Memo 56
Document Type……MEMO
Revision……..……..…… 1
Author…… …….. TN Chan, Chris Broekema, Anna Brown, Wes Armour
Release Date……..…. 2018-08-31
Document Classification…….. Unrestricted

Lead Author	Designation	Affiliation
TN Chan	System Architect	New Zealand Alliance / Compucon New Zealand
Chris Broekema	Lead	ASTRON

Signature & Date:	 2018-08-31	 2018-08-31

Acknowledgement: All benchmark tests including the programming involved were handled by Song Huang of Compucon New Zealand.

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.

Abbreviations

ARL	Algorithm Reference Library
CIP	Continuum Imaging Pipeline
CPU	Central Processing Unit
GPU	Graphics Processing Unit
ICAL	Self-Calibration Pipeline
SDP	Science Data Processor

Table of Contents

1.0 Introduction

- This memo attempts to enhance the chance that SDP hardware will be scalable to the extent satisfying the scope and performance of the pre-defined set of high priority science pipelines in terms of technology behaviour. It supplements SDP Memo 54 on Compute Efficiency and SDP Memo 55 on Hardware Technology Landscape to provide a good view of the design issues surrounding Compute Node for construction.

- This memo does not assume if SDP science algorithm and workflow should be attached or detached. Benchmark testing of any pipeline (that is, algorithm and workflow are attached) is done for revealing scaling behaviours only.

- Hardware scalability of SDP cannot be taken for granted due to a myriad of technology causes that would bring a cluster to its knees. This memo aims to provide a better understanding of the issues involved to allow solutions to be implemented accordingly.

- The universal scaling expression proposed by Neil Gunther [RD1] provides a mathematical view of this memo.

  $$\text{Speed Up (SU)} = \gamma N / [1 + \alpha (N-1) + \beta N (N-1)] \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (1)$$

  Where
  - N is the number of hardware nodes or cores in the cluster
  - γ (gamma) is a coefficient representing the linear efficiency of scaling
  - α (alpha) is a coefficient representing the effect of the code that cannot be parallelised (ref: Amdahl's Law)
  - β (beta) is a coefficient representing all other factors causing sub-linear scaling of the hardware cluster for the algorithm

- The expression is empirical in nature and is an approximation. The 3 coefficients could be functions and are shown as approximations in the form of a scalar value. Sub-linear scaling comes from non-zero alpha and beta values. The real threat is the possible existence of negative scaling or closeness to it. Negative scaling happens when an additional hardware processor causes the cluster to be less productive.

- Scaling involves the algorithms, the hardware, dataset allocations, and the scheduler. At this stage when down-selection has not happened, this memo aims to reveal scaling behaviour patterns rather than to provide a solution with the following test set up. This investigation is a step closer to solving the scaling equation.

  - The test algorithms and test data were taken from SDP Algorithm Reference Library (ARL). Details about them are given in Appendix A. Specifically, the

Continuum Imaging Pipeline, the Self-Calibration pipeline and their components were tested.

- The scheduler is Dask as implemented in ARL for testing purposes. Notes about the Dask scheduling approach and performance are given in Appendix B.

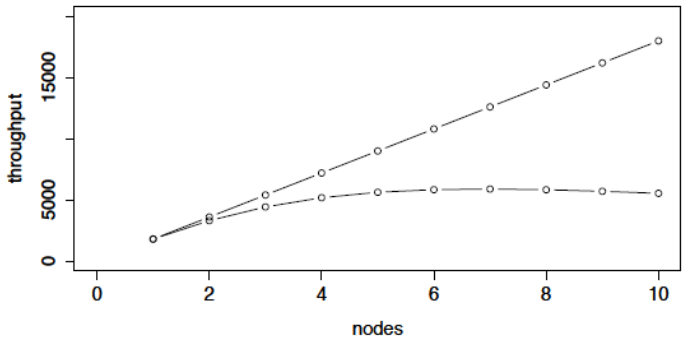
- The hardware devices are AMD EPYC 7351P (E7), AMD Ryzen 5-2400G (R5) and Intel Core i3-7100 (i3). Their features are given in Appendix C.

- Summing up the above, this investigation uses existing software and attempts to characterize the scaling properties of applications when exposed to an available scheduler. It is not intended to show how the SDP software will scale when deployed on the hardware that will be procured in the construction phase.

2.0 Scaling Coefficients of SDP Pipelines

- Ideally, each of the two SDP systems is a giant compute node, similar to the large shared memory machines in the past, that can hold many copies of a visibility grid of size 300kx300k and a raw peak processing capacity of 130PFLOPs DP (assuming for time being they have equal compute loads) to process the prescribed SDP compute load of 13.0PFLOPs DP on 10% compute efficiency. Even in this ideal situation, scaling may be sub-linear in terms of hardware cores and the same threat of negative scaling exists though in a lesser extent than a multi-node cluster.

- Scalability optimization can be a real mystery unless an accurate model of how SDP pipeline works is established. A SDP science pipeline is such defined as to be computable independently of other pipelines. Such a pipeline, say continuum imaging pipeline (CIP), can be decomposed for computation independently or individually for a range of time slices, frequency channels, sub-arrays, polarisations, and sky directions. The results of these individual computations may need to be merged for a science answer at the end but each pipeline instance is free of data dependency from other pipeline instances. That is, CIP possesses a large portfolio of freedom of computation. Each pipeline instance can be described as massively or embarrassingly parallel. Hardware scaling for this situation is normally perceived to be linear, that is, more hardware more science proportionally. This is correct but not universally correct. In the chart below, the straight line indicates linear scaling of compute nodes and the curve indicates sub-linear scaling which may happen to massive parallelism under certain conditions and is most likely associated with non-massive parallelism.



- The chart is for illustration of the concept of scaling. Refer to Equation (1). For the straight line, coefficient gamma is the slope, and coefficients alpha and beta are both zero. For the curve, all 3 coefficients are not zero.

- Each pipeline instance refers to a specific time period, set of frequency channels, sub-array, polarisation and sky direction. Within each of these pipeline instances, computations do not necessarily fall into the category of massively parallel as data dependency may be required from process to process within the pipeline. Gathering of data from individual processors is a synchronisation block as all processors of the same pipeline cannot proceed until data dependency has been resolved. In this type of situations, parallelism is restrained. How many hardware processors can be allocated to one such pipeline instance is a scaling issue. This is where the real threat exists.

    - An example of scattering and gathering could be obtained from RD2. A diagram is replicated here for illustration purposes.

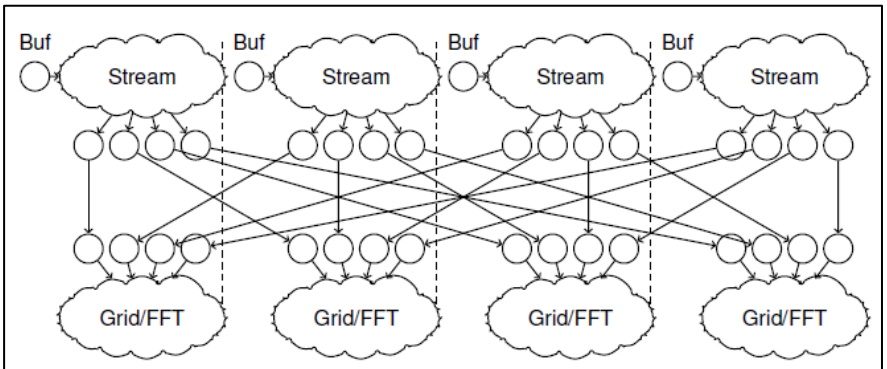
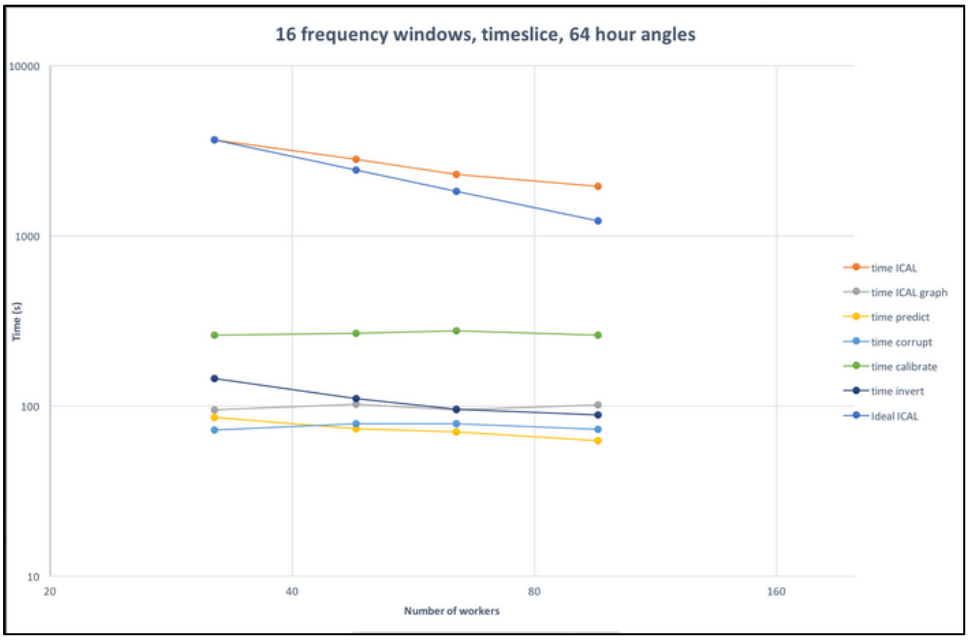


Figure 3: All-to-all Distribution Pattern of Facets, Polarisations and Taylor Terms

    - An experiment done by Tim Cornwell on 2017-11-10 shed some light on scaling of SDP applications written in ARL. The test was set up with dask.delayed for turning the ICAL code into a graph for parallel computing and dask.distributed for distributed computing on 64 physical cores of Alaska (which is known as P3) set up in Cambridge. Since the cores support hyper-threading, up to 128 workers were tested. The code was amended to remove

the data gathering step. Dataset was set for 16 frequency channels and 64 time slices (hour angles). As the test results in the chart below shows, calibrate (green) & corrupt (light blue) runtime was flat for 32 to 128 workers whereas invert (dark blue) and predict (orange) scaled reasonably well. Flat means no reduction of runtime when more workers were added. The tests showed scaling was not as smooth as desired. https://confluence.ska-sdp.org/pages/viewpage.action?pageId=235700447



- Equation (1) is empirical and the coefficients have not been expressed in terms of the causes so far. It is necessary to resort to hands-on benchmark testing to reveal scaling behaviours for specific cases as recorded in this memo.

- On top of task parallelism, SDP involves algorithms in data parallelism such as Convolution and Fast Fourier Transform. When computed in a compute node, GPU would perform better than CPU due to the design of the GPU as a huge SIMD (single instruction multiple data) engine. GPU operation may be advancing to inter-node in the near future but for simplicity its operation in a single node shall be assumed. That is, data movements involved are limited to local node grid data only. Despite the GPU is a great asset for SDP, it will not replace the CPU or eliminate the scaling problem affecting the CPU. Even if it is programmatically possible to communicate directly between GPUs, they will still have to communicate over the network and thus there will still be a bottleneck to be considered for scaling.

- The computing scenario being set up is the standard X86 accelerator programming model where a compute node consists of one or more CPU devices and each CPU device is connected with one or more GPU devices. Scaling in terms of threads, cores, sockets and nodes of CPU hardware is the context for benchmark testing in this investigation.

3.0 Test Set-ups & Results

- Benchmark tests were set up for revealing the technology causes underlying the scalability of distributed parallel computing. The following factors were believed to be involved. Ideally each of the above factors could be revealed one at a time but this was not realistic. Subsequently the tests were aimed at seeing behaviour patterns rather than resolving the values of coefficients of Equation (1).

  a) Application computation style (discussed in Section 4)
  b) Serialisation costs
  c) Compute time
  d) Data allocation
  e) Scheduler overhead
  f) Memory bandwidth
  g) Network bandwidth
  h) Disk bandwidth

- Tests were initially carried out in a single node arrangement before scaling out was attempted. Test results looked alarming but they were expected since no optimization of any sort had been carried out. These tests were solely for learning the behaviours of the tested objects.

  - Dask Overhead (see Appendix D). The runtime overhead was obtained from the runtime difference between 2 scenarios: the same application was executed in the same hardware with and without Dask Scheduling. The overhead appeared to be very substantial in a single node arrangement. The overhead was not a fixed constant value and it varied when the dataset size was increased.

  - ARL ICAL Pipeline in Single Node (see Appendix E). Many observations were made on the scaling performance of hardware devices for ICAL pipeline. Negative scaling was observed when an additional worker allocated with a physical CPU core slowed down the execution of the entire application. In terms of memory, alarm was raised when the physical memory was divided up for multiple workers and the grid size became rapidly limited. Care must be exercised to avoid triggering swapping of data to disk (SSD/HDD). In terms of runtime, a higher operating frequency and plenty of cache are desirable. Overall, the test results indicated that hardware may not play as important a role in scaling as in compute efficiency. Compute efficiency can be determined from the application and the computer hardware. Scaling involves additional factors with the scheduler being the main one.

  - ARL Invert in Single Node (see Appendix F). Similar observations were obtained for Invert as for larger pipelines and negative scaling appeared. This comment was based on 3 scenarios of the application- a long pipeline as

in ICAL in Appendix E, a shorter pipeline as in Invert and Deconvolution, and a shortest pipeline as in Invert. The scheduler and hardware were kept the same, and so they were suspected to be the cause for sub-scaling and negative scaling in all three test cases.

- Scaling of Multiple Nodes

    - The tests carried out include 2 sets for a data intensive algorithm which was the Restore component of ICAL or CIP from ARL and 1 set for a compute intensive algorithm that was hand-crafted in Python for testing purposes only. Test set up and results were recorded in Appendix G.

    - The first difference from single node tests was that multi-node tests required 3 types of nodes to be specified: the scheduler, the worker(s) and the client. E7 was specified as the client and the scheduler. E7, R5, and i3 were specified as the workers individually and as one cluster respectively. The choice of these hardware devices did not reflect any design direction at all. Similarly, the network was provided with a 100mbps switch and it was certainly not going to be SDP production hardware.

    - Dask offered 2 modes of use of compute nodes- a node is one worker or a node is a collection of workers as specified by the user. In the former case, Dask would still use the hardware resources present in the compute node in a parallel computing manner by default. This can be proven by comparing the wall clock runtime with a 'no Dask' set up and the single worker mode took less wall clock runtime than the 'no Dask' set up.

    - Negative scaling was revealed easily from the data intensive test sets, whereas it did not happen in the compute intensive test set with the limited amount of compute nodes and workers available.

    - When the application was data intensive, scheduling and network costs were so excessive that they rendered the available hardware resources beyond one node to futility. However, when the application was computing intensive, scaling was so healthy that the 22 hardware cores (over 3 compute nodes) used in the test were not enough to expose any signs of negative scaling. The low speed network switch hardware did not appear to offset the utilisation of available hardware cores at all.

4.0 Scaling Recommendations

- All benchmark testing and observations were based on Dask as the distribution scheduler and 3 specific X86 COTS hardware devices. The algorithms tested were mostly sourced from ARL, but some were amended to facilitate testing, and

one application was created from scratch for multimode testing purposes. These combinations have produced negative scaling and reminders of what to do and what not to do for SDP as far as scaling is concerned.

- There were no attempts to solve the value of coefficients in Equation (1) or to relate the coefficients to the causes in this investigation. Nevertheless, test results do confirm the validity of Equation (1).

- Priority of SDP development attention shall be in the sequence of science pipelines, data allocation, scheduler and hardware as elaborated below. The progression of development would most likely take an iterative approach to be realistic.

- Tim Cornwell has tested ARL with Dask on different hardware and has lots of experience with scaling. His observations are useful and are added below for completeness.

- The recommendations below are fit for referencing but shall not be taken as gospel due to the fact that they were based on investigations with application code and a scheduler that were not written for SDP production or performance at the outset.

4.1 SDP ARL Applications

- The science pipelines in ARL were established for reference purpose and they were written in Python. Python is known to be user friendly and flexible and is not runtime efficient. Also ARL was written in a single thread fashion (though it employed Numpy as often as the opportunities allowed). It is expected that SDP production code would be written for higher runtime efficiency.

    a. As far as distributed computing is concerned, the degree of scaling would decrease in the following sequence: best when the applications can be distributed in a massively parallel mode, not as good when there are data to be distributed, and worst when there are synchronisation blocks.

    b. A simpler view of the above scenarios can be expressed with compute intensive versus data intensive, which are in turn indicated by the operational intensity of the algorithm. Applications with a high operational intensity would scale very well.

    c. Serial portions of the code could not be distributed for computing and should be minimized. An alternate view is to increase the amount of parallel computation that can be done whilst holding the serial portion constant.

d. Tim Cornwell made the following observations that are helpful in supporting the above recommendation. Source: https://confluence.ska-sdp.org/pages/viewpage.action?pageId=235700447 updated in 2017-11. "The calculation part of any algorithm seems to scale nicely whereas the reduction part is where we are losing efficiency. We have some choice in the imaging algorithms as to how much processing we do before reduction. For example timeslice + wprojection and wstack+wprojection can be tuned to reduce the reduction at the expense of more FLOPs."

4.2 SDP Data Allocation

- Although SDP enjoys a portfolio of massively parallel computational freedom, the split of data in each pipeline incidence to member workers must be done to avoid partial splitting among 2 or more workers. For example, distributing 7 or 17 frequency channels to 16 workers would be a case of partial data splitting among workers. In this situation, a worker would not have sufficient data to complete a cycle for one frequency channel and data transfers from another worker would be required. Test results in Appendix E showed that scaling ran against the wall after (8,2) for E7 and grid size 4096 because there were only 5 frequency channels and 5 time slices of data to share. Test result from Tim Cornwell quoted in Section 2 showed better scaling of ICAL to 128 workers presumably due to having 16 frequency channels and 64 time slices for sharing.

  a. Tim Cornwell made the following observations on problem size that are helpful in supporting the above recommendation. Source: https://confluence.ska-sdp.org/pages/viewpage.action?pageId=235700447 updated in 2017-11. "For large images, the sum_psf step takes many seconds. This is due to the inter-node communication needed. For small images and many workers, the processing time get so short that the event rate becomes too high for the Dask scheduler to keep up. I am using the Dask distributed scheduler. The cure for this is to decrease the event rate by making the problem size larger. But then the sum_psf starts to dominate."

4.3 Dask Scheduler

- Dask is part of the Python ecosystem- it aligns with the high level approach of Python and therefore SDP shall expect to pay a premium in terms of runtime for the ease of use of Dask.

- The runtime overhead incurred by Dask scheduling was non-trivial as shown in both single node and multiple node tests. Only compute-intensive applications were able to hide the Dask overhead and low speed Ethernet hardware switching.

- As at 2018-08, four schedulers were known to be tested by SDP [RD3]. Two of them are open source being Dask and StarPU, and the other two are hand-

crafted being serial and MPI/OpenMP. MPI/OpenMP would be runtime efficient for being closer to the metal if the development is successful.

a. A Dask author claimed that Dask has a scheduling capacity of more than 3000 tasks of 100ms each. Event rate is one condition and the amount of data transfer is the other condition constituting a dynamic restraint on scaling from the scheduler's perspective.

b. The same author also advised that Dask scaled well up to 256 workers, and Dask liked to deal with bigger workers and bigger workload for each.

c. The same author provided an update of his views as recent as 2018-06-26. He put the scaling performance of Dask into the perspective that it scales about the same as Apache Spark but less well than a high performance system like MPI. He further commented that when users stick to standard workflows like Dask Dataframe or Dask Array, they will probably be OK but when operating with full creativity some expertise will invariably be necessary. Reference: http://matthewrocklin.com/blog/work/2018/06/26/dask-scaling-limits

4.4 Hardware

- The best hardware for scaling shall have lots of memory and large cache, high memory bandwidth, and high operating frequency. The number of cores shall be at least one more than the number of GPU devices available in the compute node. The present X86 COTS technology does not allow too many full bandwidth PCIe links for GPU installation. SDP may go for a single socket CPU such as tested in this investigation as it supports 3 GPU devices, 2TB of DDR4 max, and 8 memory controllers. However, its boosted frequency is only 2.9GHz and this is a major deficient feature. There should be pleasant developments in this hardware space over the next 4 years such as the adoption of high bandwidth memory (HBM) for CPU.

  a. Virtual cores are not as helpful as physical cores as far as scaling is concerned. Virtual cores may force the early emergence of negative scaling.

  b. Plenty of main memory per worker is helpful due to Dask starting to swap memory with SDD/HDD when the memory reaches 60% full. The threshold can be changed by users.

  c. Infiniband did not help to release the bottleneck more than Ethernet (observation made by Tim Cornwell in 2017-11)

  d. Network switching hardware is not as important as the application from a scaling perspective. If the application is compute intensive, it would

tolerate low speed network switching hardware. If the application is data intensive, high speed network switching hardware would not release the bottleneck.

5.0 References

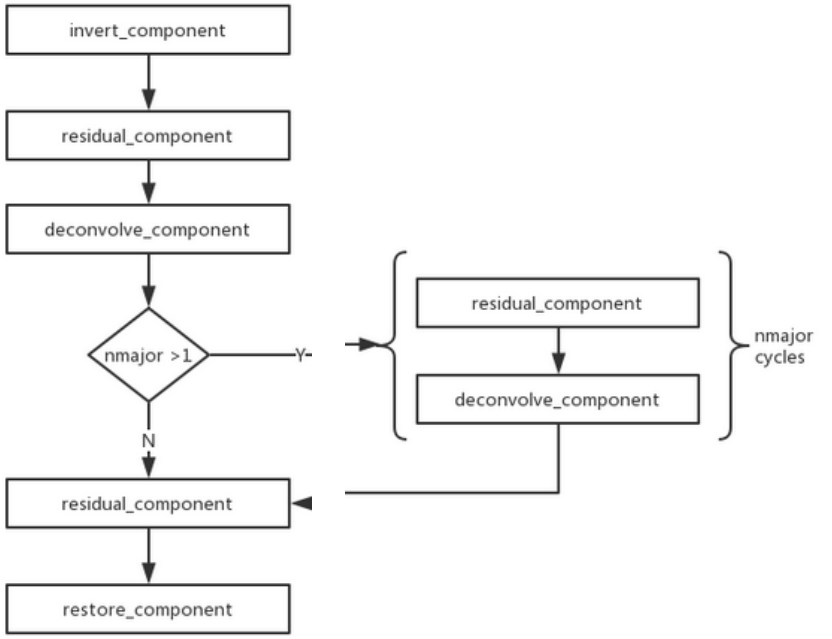
- RD1 Universal Scalability Law, http://www.perfdynamics.com/Manifesto/USLscalability.html
- RD2 SDP Memo 38 Pipeline Working Sets and Communications, Peter Wortmann, Rev 2, 2017-12-14
- RD3 ARL Integration with Workflow, Tim Cornwell, https://confluence.ska-sdp.org/display/WBS/ARL+integration+with+workflows last visited 2018-08-15

6.0 Appendices

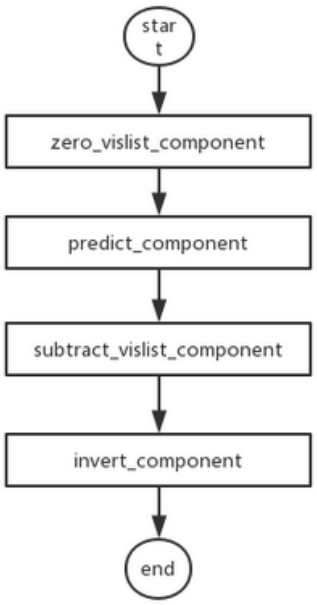
- A Test Algorithms from ARL (Flowcharts)
- B Test Scheduler Dask (Notes)
- C Test Hardware (Key Features)
- D Dask Scheduler Runtime Overhead (Test Results)
- E Dask for ARL ICAL in Single Node (Test Results)
- F Dask for ARL Invert in Single Node (Test Results)
- G Dask for Multiple nodes (Test Results)

Appendix A: Test Algorithm from ARL (Flowcharts)

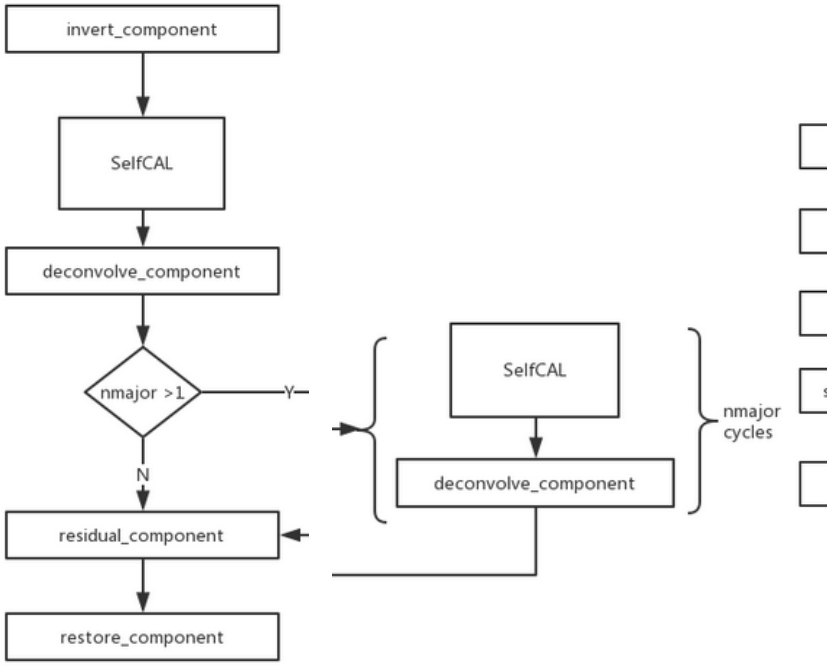
CIP



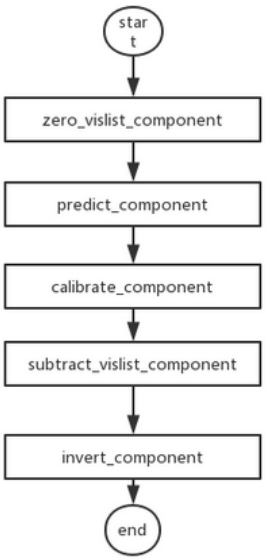
Residual Component



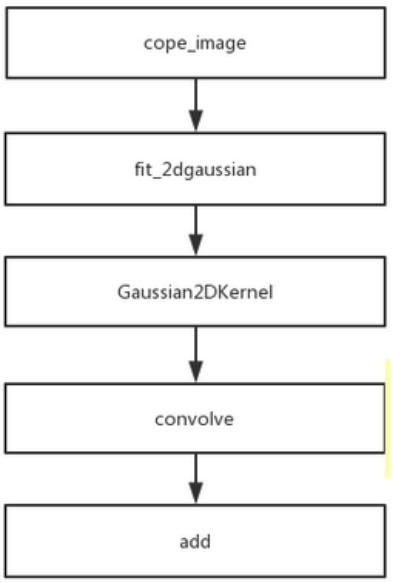
ICAL



SelfCAL



Restore Component



Appendix B: Test Scheduler Dask (Notes)

- Dask is an open source software library in the Python ecosystem supplying tools to parallelize Python code (dask.delayed) and to schedule the execution of the code in a distributed parallel computing manner (dask.distributed) in a single compute node or a compute cluster. It abstracts away all low level complexity involved for the comfort of the application programmer.

- Dask.distribute is a scheduler which organises allocation of tasks and data to workers in the compute cluster. This involves a lot of inter-worker communication. Dask uses TCP as the default transport protocol and RPC for remote node communication, and deploys Tornado asynchronous networking library to manage simultaneous communications. Distributed computing is hard for two reasons: (1) Consistent coordination of distributed systems requires sophistication. (2) Concurrent network programming is tricky and error prone.

- Consider the above summary as the first level overview of Dask. Below is the second level information consisting of a number of notes pertaining to scaling performance and features. This appendix does not mention all Dask features or any API or command line syntax at all.

  a) One scheduler to worker communication cycle takes about 10ms. This is roughly the overhead cost of Dask.distributed per task.
  http://distributed.readthedocs.io/en/latest/journey.html

  b) Workers also incur a latency overhead cost of 1ms.
  http://distributed.readthedocs.io/en/latest/actors.html

  c) When a user scatters data from their local process to the distributed network this data is distributed in a round-robin fashion grouping by number of cores.
  http://distributed.readthedocs.io/en/latest/locality.html

  d) Dask is written in Python and only really supports Python. It interoperates well with C/C++/Fortran/LLVM or other natively compiled code linked through Python.
  http://dask.pydata.org/en/latest/spark.html

  e) At 60% of memory load, Dask will spill least recently used data to disk. The threshold figure is user adjustable.
  http://distributed.readthedocs.io/en/latest/worker.html

  f) Dask can run fully asynchronously and so interoperate with other highly concurrent applications. Internally Dask is built on top of Tornado co-routines
  http://distributed.readthedocs.io/en/latest/asynchronous.html

  g) The Client methods scatter, map, and gather can consume and produce standard Python Queue objects. This is useful for processing continuous streams of data. However, it does not constitute a full streaming data processing pipeline like Storm. It is not a proper streaming system.
  http://distributed.readthedocs.io/en/latest/queues.html

h) You can use resources such as GPU and memory with Dask Collections such as arrays and delayed objects.
http://distributed.readthedocs.io/en/latest/resources.html

i) The documentation describing how Dask allocates workers to cores in a compute node cannot be found. It has therefore been assumed that workers were allocated to share available cores in the node as evenly as possible in integer numbers. Examples

16 workers for 32 cores → 2 cores per worker (1 physical and 1 hyper-thread)
32 workers for 32 cores → 1 core per worker

Appendix C: Test Hardware (Key Features)

Compute Node Hardware device abbreviation in brackets

- (i3) Intel CPU i3-7100
- (R5) AMD APU Ryzen5-2400G
- (E7) AMD Server CPU EPYC 7351P

CPU Model	i3	R5	E7
Process	14nm	14nm	14nm
Op Frequency	3.9GHz	3.6GHz	2.4GHz
# of Cores	2	4	16 (note 1)
# of Threads	4	8	32
SIMD Engine	AVX-2	AVX-2	AVX-2
L2 Cache	3MB	4*1/2MB	16 * 1/2MB
L3 Cache	0	4MB	8*8MB
CPU TDP	51w	65w	170w
Memory Channel	2	2	8
Memory Max			2TB
Memory installed	32GB	64GB	128GB
DDR4 Speed	2133	2400	2400

Note 1: EPYC is a system-on-chip in terms of packaging techniques. Each model of EPYC consists of a number of clusters of 4 cores, each having its own L3 cache and communicating with other clusters over a high bandwidth coherent inter-connect.
Ref: https://www.linleygroup.com/uploads/amd-epyc-performance-wp-final.pdf

Appendix D: Dask Scheduler Runtime Overhead (Test Results)

1. Test Purpose and Environment

    - The purpose was to reveal the runtime overhead of Dask for scheduling. One test was done with Dask – see Appendix B for 1 worker and 1 process. Another test was done without Dask in a single thread. The hardware device used was E7 (single node) - see Appendix C. The algorithms tested were CIP – see Appendix A. Grid sizes tested were 2048 and 4096.

2. Test Results and Interpretations

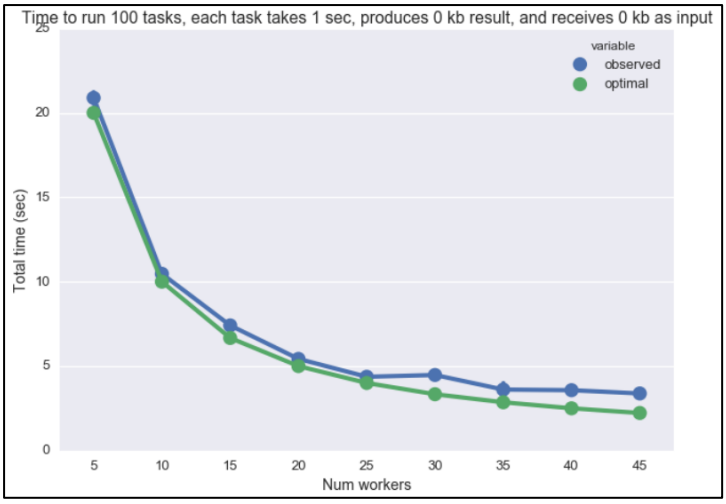
    - Dask took 79% additional runtime (in seconds) for a smaller grid size of 2048x2048. The extra runtime dropped to 24% for the bigger grid of 4096x4096.

CIP grid size	2048	4096
CIP w/o Dask	138	5619
CIP with Dask	248	6969
Dask Overhead	110	1350
Overhead %	79%	24%

    - Dask is a scheduler and it is not a free lunch. Dask is based on TCP (Transmission Control Protocol from the Internet Engineering Task Force) as the default transport protocol even though the test was carried out in one worker (core) and no inter-core or inter-node data transfer was needed in this test. The extra runtime taken was therefore solely a scheduler overhead.

    - The grid size of 4096 is 4 times of 2048. The runtime ratio is much higher than 4 times. Dask has provided for data to be swapped into SDD/HDD when memory utilisation hits 60%. See APP-B. Swapping is suspected to be accountable.

    - For reference, Dask incur a minor but visible overhead when there is no data transfer at all which takes data communication out of the equation. The overhead is solely a Dask administrative overhead. Source: https://github.com/dask/distributed/issues/531 dated 2016-09


Time to run 100 tasks, each task takes 1 sec, produces 0 kb result, and receives 0 kb as input
variable
observed
optimal
Total time (sec)
0
5
10
15
20
25
5
10
15
20
25
30
35
40
45
Num workers

Appendix E: Dask ARL ICAL in Single Node (Test Results)

1. Test Purpose and Environment

    - The purpose was to reveal the scaling behaviour of 3 different CPU hardware devices

    - The code for testing was obtained from ARL. Specifically this test was on ICAL. See APP-A for the algorithm component flow charts.

    - Dask for Single Node distributed computing as explained in Appendix-B or vendor's tutorials.
      http://dask.pydata.org/en/latest/setup/single-distributed.html

    - Hardware devices tested were E7, R5 and i3. See APP-C.

    - Test Dataset Dimensions
        - Baselines = 166 * 165 / 2
        - ntimes = 5
        - nfrequency =5
        - polarisation = 1
        - major cycles = 5
        - minor cycles = 1000
        - Cleaning = mmclean
        - Grid Size = 512, 1024, 2048, 4096 (square for all)

2. Test Results and Interpretations

    - Some interesting results on hardware performance were revealed. All tests were subject to the same science algorithms, same amount of test data (provided by ARL), same processes and the same scheduler. The memory installed was different but it was the same at 8GB per worker for all 3 hardware. For E7, only 4GB was available when 32 workers were tested.

      (w,t) is the number of workers and number of threads per worker respectively. (16,2) means 16 cores each handling 2 processes.

Runtime s	npixel = 512			npixel = 1024			npixel = 2048			npixel = 4096		
(w,t)	E7	R5	i3	E7	R5	i3	E7	R5	i3	E7	R5	i3
32,1	66.1			92.8			**151.3**					
16,2	86.9			88.6			**155.2**			2561		
16,1	**64.8**			**77.0**			**146.8**			2525		
8,4	92.6			102.7			177.9			2538		
8,2	73.6			86.0			193.6			**2488**		
8,1	80.3	66.2		78.4	**85.2**		158.4	155.5		2557	2267	
4,4	140.0			142.8			198.4			2548		
4,2	130.5	109.1		136.9	138.9		202.8	178.4		2500	**2182**	
4,1	106.4	90.7	**91.3**	120.3	105.6	**119.7**	213.1	174.9	**228.2**	2693	2515	**5269**
2,16	235.8			250.5			345.2			2630		
2,8	261.0			273.4			321.3			2632		
2,4	220.9			271.9			297.1			2608		
2,2	189.9		115.4	206.5		139.1	285.9		239.1	2694		5497
2,1	147.0	119.4	101.8	178.7	151.9	132.2	335.9	274.7	255.7	4870	3598	6987
1,1	254.5	190.8	147.2	305.9	236.5	184.8	585.2	489.8	345.2	8006	6232	9539

(a) Memory Real Estate per Worker.
Not surprising is that E7 failed to run the largest grid size of 4096 for 32 workers with 4GB each. This is an indication of the critical role of memory size for SDP pipelines.

(b) Number of Cores for Distributed Computing
E7 took the least time to complete execution for 512, 1024 and 2048 grid sizes with 16 workers. However, each case was only marginally faster than R5 with 8 workers. This result is not surprising as more cores should help.

(c) Operating Frequency
Its role is clear- higher the frequency, faster was the computation in a distributed manner. (i3) at 3.9GHz therefore took the least time for the same number of cores than (R5) and (E7). (R5) took over the crown when the grid size was large at 4096. Large grid size required more memory transfer or a bigger cache. (R5) indeed has double of the cache as (i3). See APP-C.

(d) Optimal Hardware Capabilities
Surprising is that R5 took the least time to complete running the bigger grid size of 4096 with 4 or 8 workers (faster than EPYC with more physical cores). R5 has a higher operating frequency, same amount of cache per core, and presumably better inter-core memory transfer as well. E7 has 16 cores in 4 clusters, making it essentially a set of NUMA processors in a single socket. Therefore some cores are further away than others.

(e) Scaling bottleneck appeared
- Grid size 512, shortest run time was achieved by E7 with 16 cores, meaning that 32 workers took longer run time and therefore counter-productive. As 32 workers were supplied with 16 physical and 16 virtual (hyper-threading) cores, the test result did not indicate if 32 physical cores would be faster. No conclusion can be drawn yet but definitely virtual cores did not help for pushing the envelope.

- Grid size 1024: as above
- Grid size 2048: as above
- Grid size 4096: E7 achieved the shortest execution time with 8 real cores each handling 2 processes. This means its 16 real cores were useless. This is conclusive as far as the set of test conditions is concerned and is a sad conclusion for E7. Inter-core memory bandwidth is an important factor for scaling and E7 did not do well due to its 4 core per die arrangement.

(f) Virtual Cores

In the small grid test cases for E7 with (32,1), 16 of the cores are physical and 16 are virtual (hyper-threading). Virtual cores were not useful for E7 as in (32,1) > (16,1) in runtime for 512 grid size. However, R5 or i3 did not have that many cores to get to that position and its virtual cores were helpful. Do not rule out virtual cores yet until close to the scaling bottleneck.

(g) Number of Processes per Worker.

Running two independent processes in parallel in two cores is always faster than running them in sequence in a single core. Processes shall be allocated to cores if available. This situation is reflected consistently in the table of test results. For example E7 (4,1) took 106.4s to process a grid size of 512 and (2,2) took 189.9s. The runtime would be longer if more processes were allocated to the same cores such as 220.9s for (2,4) and 261.0s for (2,8) even though the total amount of computation remains constant.

(h) SSD/HDD Data Swapping

The runtimes for grid size 4096 were substantially higher than for grid size 2048 in all hardware and worker number cases. It is possible that the extra time was caused by data swapping with SSD/HDD as Dask would start to do so when the memory space per worker hit 60% utilisation. See APP-B

Appendix F: Dask ARL Invert in Single Node (Test Results)

1. Test Purpose and Environment

    - The purpose was to reveal the scaling behaviour of a smaller pipeline in ARL to the one recorded in Appendix E. This test involved the Invert process of visibility data and Deconvolution of the dirty image. Another smaller test was done without Deconvolution (which has a high operational intensity) to see the difference in scaling.

    - Only one type of hardware being E7 was tested.

    - Test Dataset Dimensions
        - Baselines = 512 * 511 / 2
        - ntimes = 5
        - nfrequency =7
        - polarisation = 1
        - major cycles = 5
        - minor cycles = 1000
        - Cleaning = mmclean
        - Grid Size = 512, 1024, 2048, 4096 (square for all)

2. Test Results and Interpretations

    - Main memory installed was 128GB (same as last test). Each worker had 8GB if there were 16 workers, but 4GB only if there were 32 workers. As shown in the last test, both smaller pipelines failed to run the largest grid size of 4096x4096. In the table below, there are 2 columns of test results per grid size- the first column is the Invert process plus Deconvolution and the second column does not have Deconvolution. Red figures are the shortest runtime.

npixel	512	(no decon)	1024	(no decon)	2048	(no decon)	4096	(no decon)
1,1 (w,t)	100.38	88.84	111.53	92.81	171.05	104.33	393.93	143.17
2,1	58.56	50.79	64.96	52.64	110.68	57.81	310.60	78.07
4,1	33.25	27.18	43.78	28.90	84.91	32.71	274.63	49.38
8,1	20.38	15.73	31.03	16.73	89.43	20.48	258.53	35.26
16,1	19.70	16.23	29.25	16.71	88.06	20.58	261.22	34.75
32,1	19.77	15.82	29.73	17.37	89.10	21.30	error	error

    Since the E7 processor has 16 real cores only, negative scaling occurring at 32 cores if existing would not be conclusive because 32 real cores may continue to scale.

    Grid size 512 (no decon) test results are confusing as (32,1) continued to scale after (16.1) but the shortest time happened at (8,1). For larger grid size, no decon scaled better than with decon and scaling stopped sooner than

smaller grids. The most logical explanation for the lack of a consistent pattern in terms of scaling is that data communication among workers was incurred due to uneven allocation of dataset to workers (5 time slices and 7 frequency channels for workers).

Appendix G: Dask in Multiple Nodes (Test Results)

1. Test Purpose and Environment

- The purpose was to reveal the impact of involving multiple hardware nodes on scaling in terms of wall clock time and if and when sub-scaling and negative scaling happens.

- Test hardware devices were
  - E7 with 16GB x8 memory
  - R5 with 16GB x4 memory
  - I3 with 16GB x 2 memory

- Two test algorithms were covered. The “data intensive’ algorithm was based on ARL Restore but the code was amended for enabling tests to be done as recorded below. The ‘compute intensive’ algorithm was hand-crafted from scratch. Both algorithms were coded in Python.

```
from dask.distributed import Client
from time import time
from astropy.convolution import Gaussian2DKernel, convolve
import numpy as np

MAX = 4
POL = 4
NPIXEL = 512

model0 =
np.asarray(np.random.randn(MAX,POL,NPIXEL,NPIXEL),np.float32)
restored0 = np.zeros((MAX,POL,NPIXEL,NPIXEL),np.float32)

def restore(model):
    size = 3.0
    norm = 2.0 * np.pi * size ** 2
    restored = np.zeros_like(model)
    gk = Gaussian2DKernel(size)
    for pol in range(model.shape[0]):
        restored[pol, :, :] = norm * convolve(model[pol, :, :],
gk, normalize_kernel=False)
    return restored

def f_many(chunk):
    return [restore(model0[chunk])]

if __name__ == '__main__':

    client = Client("localhost:8786")

    t0 = time()
    A = client.map(f_many, range(MAX))
    B = client.gather(A)
    C = np.asarray(B)
    for i in range(MAX):
        restored0[i] = C[i,0]
```

```
t1 = time()
print("Client = %.3f" % (t1 - t0))
client.close()

for i in range(MAX):
    restored0[i] = restore(model0[i])
t2 = time()
print("Normal = %.3f" % (t2 - t1))
```

2. Test Results

- The data intensive algorithm test results were shown in Table Test A and Table Test B. The compute intensive algorithm test results were shown in Table Test C.

    - Test A was done on 3 compute nodes (E7, R5 and i3) and Dask was instructed to use these hardware nodes without specifying the number of workers in each. By default, Dask treated one node as one worker possessing multiple threads for parallel processing. As such, E7 had 32 threads, R5 had 8 and i3 had 4. All physical cores were capable of hyper threading, therefore, half of the threads were done on physical cores and half on virtual cores (hyper-threads).

    - Test B and C were done on the same 3 compute nodes and Dask was instructed to use physical cores only.

- Contents of test results in each table

    - The Scheduler was dask.distributed and it resided in the client. E7 was chosen as the client and where the scheduler resided.
    - Workers indicate the type of workers such as E7 by itself, or E7 plus R5 and so on.
    - The 3 hardware devices were connected with an Ethernet 100mb/s switch.
    - The data shape is shown in Row 1 of the table. (4,2,1024) means 4 frequency channels, 2 polarizations, and grid size of 1024 x 1024
    - Figures are wall clock runtimes of test in seconds
    - Interpretations are shown outside the table on the RHS
    - The yellow and blue colours are for ease of association between Test A and Test B

3. Test Interpretations for Test A and Test B

Table Test A- data intensive			Each node is a worker with multiple threads inclusive of virtual cores		
scheduler	workers	4,2,1024	4,2,2048	8,2,2048	<-- 1 mult-threaded worker per node (by default)
E7	E7	9.0	36.1	67.9	<-- single node with Dask is best
E7	E7 + R5	14.6	58.5	163.1	<-- this is negative scaling for all grid sizes
E7	E7+R5+i3	18.0	71.9	220.2	<-- further negative
E7	R5	22.1	87.0	284.1	<-- inter-node cost is high
E7	R5 + i3	21.5	85.7	280.2	<-- no extra inter-node cost incurred
E7	no Dask	23.7	94.2	190.0	<-- this is the bottom line

Table Test B- data intensive			Number of workers for physical cores only		
Scheduler	workers	4,2,1024	8,2,2048	22,2,1024	
E7	E7 (x16)	9.0	66.7	88.9	<-- physical cores only, similar performance as with virtua
E7	E7+R5(x4)	8.8	67.3	162.9	<-- extra external cores helped small grid but not larger gri
E7	E7+R5+i3(x2)	8.8	69.8	179.7	<-- there were 22 frequency channels for 22 workers
E7	R5 (x4)	22.1	283.0	520.1	<-- too much work (22) for 4 cores
E7	R5+i3(x2)	22.3	265.4	534.8	<-- extra cores from added node did not help
E7	no Dask	23.8	187.0	128.6	<-- this is the bottom line

- The test results show clearly that multi-node distributed computing incurred visible wall clock time extra to single node computing for the same application, same data shape, and amount of computation.

- The extra wall clock time would be due to network data transfer latency, throughput time, and the scheduler having to deal with different hardware devices.

- In terms of scaling, the above tests were considered to have up to 3 workers only as one device was treated as one worker. Scaling appeared to be prohibited in Test A when virtual cores were involved, but scaling was available a bit (almost flat) as shown in Test B for (4,2,1024) but not for bigger grid shapes.

- The inter-node data transfer burdens were so high for the large data shape (22,2,1024) that it was better off running the application in a single node without using Dask than in multiple nodes.

- Data transfer was really a big burden as Dask with 16 workers took not much less wall time to execute the big data shape of (22,2,1024) than without Dask at all. Without Dask does not imply no parallel computing as numpy would use parallel computing by default.

4. Test Interpretations for Test C

Table Test C- compute intensive			Physical cores only	
Scheduler	workers	10000	50000	<-- the 2 figures indicate the problem size
E7	E7 (x16)	7.7	121.2	
E7	E7+R5(x4)	6.3	91.4	<-- more cores even external help
E7	E7+R5+i3(x2)	5.5	78.5	<-- more cores even external help
E7	R5 (x4)	16.4	296.8	
E7	R5+i3 (x2)	10.5	176.5	<-- more cores even external help
E7	no Dask	35.1	869.1	<-- bottom line

- Test C being based on a compute intensive application produced favourable information as far as scaling is concerned. Scaling was present and it did not appear to be flat at all as shown in the first 3 test result rows. Adding 4 workers in R5 to 16 workers in E7 helped. Adding further 2 workers in i3 to the 20 workers in E7 and R5 further helped. This is good news to SDP.

- Even inter-node data transfer burden has not invalidated the use of external nodes as the last 3 rows in the table show. Take column 10000 as an illustration, R5 alone reduced the wall clock runtime in E7 without Dask to less than 50%, and R5+i3 reduced the wall clock runtime to less than 30%. This is an example of good scaling. More reduction was recorded for a bigger problem size as shown in column 50000.

End of Memo