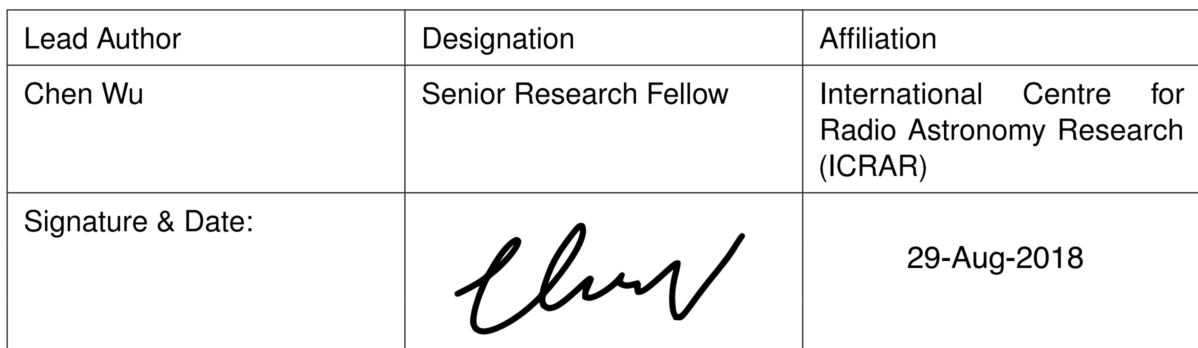
SDP Memo 066: Partitioning SKA Dataflows for Optimal Graph Execution

Document Number . . . . SDP Memo 066
Document Type . . . . MEMO
Revision 1
Author . . . . Chen Wu
Release Date 2018-08-31
Document Classification . . . . Unrestricted
Status . . . . Draft

Lead Author	Designation	Affiliation
Chen Wu	Senior Research Fellow	International Centre for Radio Astronomy Research (ICRAR)
Signature & Date:		29-Aug-2018

Revision	Date of issue	Prepared by	Comments
1	2018-08-31	Chen Wu Rodrigo Tobar Andreas Wicenec	Initial publication

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 2

2 Related work 3

3 Graph Execution 4

4 Dataflow partitioning 6
- 4.1 Partitioning Algorithm 6
- 4.2 DoP Constraint Evaluation 8

5 Conclusions 11

References 11

List of Figures 11

List of Tables 12

1 Introduction

The Square Kilometre Array (SKA) will be the largest radio telescope in the world Braun et al. (2015). The two components of the first phase of SKA (SKA1) — SKA-Mid and SKA-Low — will jointly produce large amounts of data at a rate of one Terabyte (TB) per second, with the second phase data rate reaching at least ten times higher. The graph execution engine in the SKA Science Data Processor (SDP) needs to schedule tens of thousands of algorithmic components to ingest and transform millions of parallel data chunks in order to solve a series of large-scale inverse problems within the power budget. This poses a great challenge, since the current generation of radio astronomy data processing systems are designed to handle data approximately two to three orders of magnitude smaller than that of the SKA1.

To tackle this challenge, we developed the Data Activated Liu Graph Engine (DALiuGE[1]) to execute continuous, time-critical, data-intensive workflows in order to produce science-ready data products. Compared to existing astronomical workflow systems, DALiuGE has several

[1] `https://github.com/ICRAR/daliuge`

advantages such as separation of concerns, data-centric execution, graph-based dataflow scheduling, and native support for streaming processing.

DALiuGE has been deployed and tested in existing SKA pathfinder projects to address practical data challenges. For example, DALiuGE is currently being used as the data reduction framework (Vinsen et al., 2018) of the COSMOS HI Large Extragalactic Survey (CHILES) project (Fernández et al., 2013) for the Karl E. Jansky Very Large Array. DALiuGE logical graphs and modules have been adopted (Tobar and Kitaeff, 2017) in the HI pipeline for the Five-hundred-meter Aperture Spherical radio Telescope (FAST) (Nan, 2006). Data-driven DALiuGE workflows have been developed (Wei et al., 2018) as an alternative to message passing interface (MPI) processes for managing parallel execution of the distributed SAGEcal calibration algorithm (Yatawatta, 2015) on distributed compute nodes. DALiuGE has also been integrated (Guzman, 2018; Ord et al., 2018) into the software package of the Australian SKA Pathfinder (ASKAP) (ATNF, 2018) to execute ASKAP data processing. Moreover, the DALiuGE docker container (Grange, 2018) has been developed to integrate the Low-Frequency Array (LOFAR) pre-facet calibration pipeline (Horneffe, 2017). More recently, a Dask (Rocklin, 2015) integration layer has been developed in DALiuGE (Tobar, 2018) to directly execute any Python-based pipelines — e.g. those defined using the Algorithm Reference Library (Cornwell et al., 2018) — without any code changes.

A technical overview of DALiuGE and its operational production systems are described in Wu et al. (2017). In this memo, we focus on the DALiuGE graph scheduling sub-system. In particular, we discuss technical details on dataflow partitioning algorithms and implementations. By extending previous studies on graph scheduling and partitioning, we lay the foundation on which we can develop polynomial time optimization methods that minimize both workflow execution time and resource footprint while satisfying resource constraints imposed by individual algorithms. We show preliminary results obtained from three radio astronomy data pipelines.

2 Related work

The dataflow computation model Dennis and Misunas (1975) represents workflows as Directed Acyclic Graphs (DAG), where vertices are stateless computational tasks (i.e. functions) and edges connect the output of one task with the input of another. Although the dataflow model exploits parallelism inherent in DAGs through data dependencies, mapping an irregular DAG onto hardware resources for optimal execution is an NP-hard problem Chaudhary and Aggarwal (1993). Early work attempted to derive data structures (e.g. assignment graph Bokhari (1981) or allocation graph Towsley (1986)) from the original DAG in order to perform tractable searching and optimization algorithms (e.g. using the maximum flow solutions Stone (1977)). While these algorithms were able to uncover an optimal solution in polynomial time, the growth rate of the assignment graph is $O(N \times M)$, where N denotes the number of vertices in the original DAG and M denotes the number of available processors. Therefore, as the DAG size and resource pool grows substantially (e.g. from tens of tasks running on a laptop to millions of tasks running on thousands of processors), these exact optimization methods quickly become intractable.

A variety of heuristics-based algorithms Kwok and Ahmad (1999) have been developed for scheduling DAGs on multiprocessors. These heuristics in general fall into two alternative approaches — one-phase or two-phase. In the one-phase approach (e.g., the popular HEFT algorithm Topcuoglu et al. (2002)), DAG scheduling is performed by directly mapping a ranked list of workflow tasks to another ranked list of resource units (e.g. processors or nodes) based on some aggregated run-time workflow profiles and resource statistics. In contrast, the two-phase approach (e.g. Liou and Palis (1997); Sarkar (1987)) first partitions the DAG

into a number of clusters based on heuristics such as load balancing Karypis and Kumar (1998), minimal data movement, etc. In the second phase, these clusters are then mapped onto actual hardware resources for execution. We currently adopt the two-phase approach because the output from the first phase encodes a resource demand abstraction (RDA) from intrinsic properties of the DAG. The RDA becomes the input for resource mapping in the second phase. More importantly, the RDA provides a more accurate estimate of resource demand for future capacity planning and observation scheduling for the telescope manager. However, most two-phase algorithms were targeted to multiprocessors on a single compute node, where each workflow task consumes exactly one processor. Our workflows need to run across clusters of compute nodes, each consisting of multiple processors. More importantly, each workflow task inherently demands multiple yet different number of processors/cores and different amount of memories. Dealing with this kind of complexity in resource demand and multiplicity in resource capabilities is the main focus of this memo. Moreover, unlike most existing DAG scheduling/mapping algorithms, our partitioning algorithm aims to reduce the overall resource footprint given these complexities and constraints.

On the other hand, the advantage of the one-phase approach is its flexibility to incorporate run-time resource heterogeneity. We will discuss the application of the one-phase approach to our DAG mapping problem in the next memo. This memo focuses solely on the two-phase approach.

Although significant progress Bateni et al. (2017); Martella et al. (2017); Tsourakakis et al. (2014) has been made recently to partition vary large graphs for various social network analysis and machine learning applications, direct application of these graph partitioning algorithms for dataflow partitioning often leads to sub-optimal solutions. This is because the DAG (or general graph) representation G of the dataflow does not encode the notion of workflow execution working set W_t - a small set of workflow tasks that are being executed at time t. Only tasks in W_t consume resources, other tasks are either waiting for the completion of their “upstream” tasks in W_t or have already completed their executions. Therefore, partitioning the entire graph G (e.g. on the order of millions of nodes) for subsequent resource mapping is (1) wasteful given that $|W_t| \ll |G|$, and (2) ill-posed since W_t is time-dependent and is unknown at the time of graph partitioning.

3 Graph Execution

Following the two-phase approach, the four steps of the graph execution are illustrated in Figure 1. We briefly introduce them in this section. Readers are referred to Wu et al. (2017) for a detailed technical discussion on graph execution.

Starting from the top left corner, a staff astronomer composes a logical graph representing high-level data processing capabilities (e.g., “Image deconvolution”) using resource oblivious dataflow constructs and workflow task components. The first step unrolls the logical graph by expanding all parallel branches and loops, instantiating tasks in all branches and iterations and connecting them with directed edges as per the logical graph definition. The result of unrolling is the Physical Graph Template (PGT) shown in the top right corner of Figure 1. It should be noted that, unlike traditional dataflow graph representations, DALiuGE models both data and tasks as graph vertices, known as “Drops”. In a nutshell, Drops (Wicenec et al., 2018) are software objects wrapping a generic payload. Shown as parallelograms in a PGT, *Data Drops* trigger the execution of their consumer tasks, which are *Application Drops* denoted by rectangles in a PGT.

The second step divides the PGT into a set of logical partitions such that certain perfor-

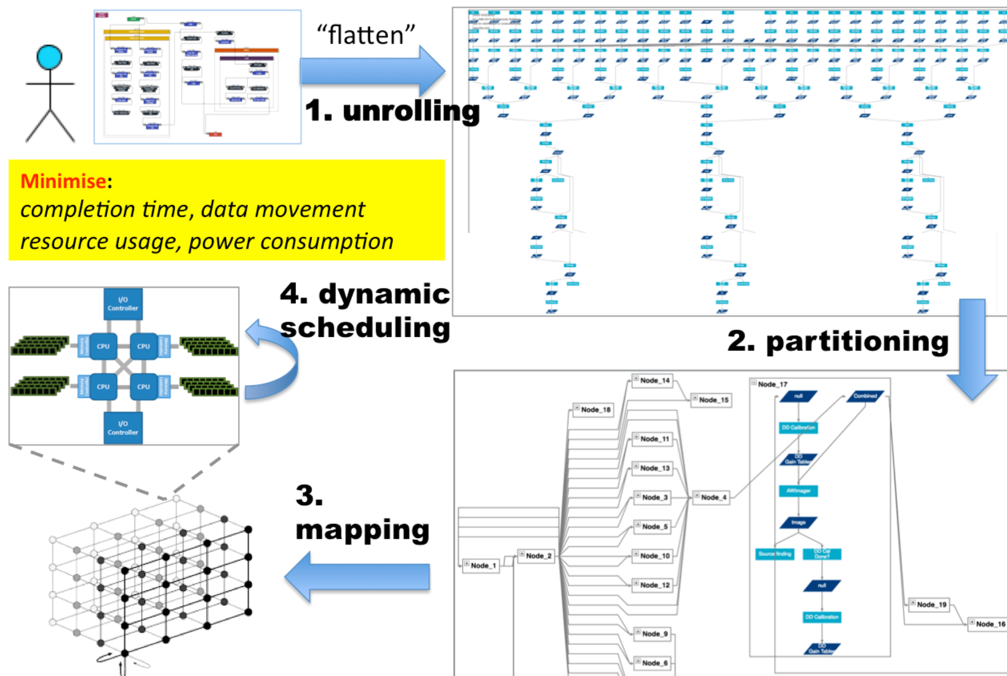


Figure 1: The complete dataflow execution cycle consists of four major steps — unrolling, partitioning, mapping and dynamic scheduling. Both unrolling and partitioning are performed offline. Mapping happens just a few minutes before the workflow execution, and dynamic scheduling is done in real-time during execution. While the first three steps target the entire graph across multiple resources, the last step focuses on tasks using local resources on a single node.

mance requirements (e.g. total completion time, total data movement, etc.) are met under given constraints (e.g. resource footprint, collocation criteria, device locality, etc.). This step outputs the Physical Graph Template Partition (PGTP), which provides the Telescope Manager with an approximate solution to construct the observation scheduling blocks months or weeks prior to observation and compute resource allocation. An example of PGTP is shown at the bottom right of Figure 1, where 19 partitions are produced and one of them is visually expanded with 11 enclosing workflow tasks. Furthermore, a resource reservation that contains 19 nodes can be submitted to the telescope manager weeks before the associated observation takes place.

The third step maps each logical partition of the PG onto a given set of currently available resources in certain optimal ways. In principle, each partition is placed onto a physical compute node in the cluster. Such placement requires real-time information on resource availability, and we currently assume resource pools consisting of nodes with identical capabilities of computing, storage, and interconnect. In cases where the number of partitions p is greater than the number of available nodes m, DALiuGE can be configured to merge the p PGT partitions into m virtual clusters with the goal of balancing the overall workload (both compute time and memory usage) evenly before mapping.

The final step involves optimal execution of tasks that have been allocated to a single node by the previous two steps. DALiuGE currently offloads this step to local schedulers provided by the host OS running on each compute node. We are currently working on the integration of graph-based GPU schedulers for dynamically scheduling GPU accelerated workflow tasks on single node with multiple GPUs.

The remainder of this memo discusses technical details of the second step — dataflow partitioning.

4 Dataflow partitioning

During graph partitioning, a PGT of N vertices is decomposed into M partitions, each of which conceptually represents a compute node with a pre-defined resource capacity vector C. The goal of graph partitioning is to obtain an estimate on the minimum number M^* of compute nodes needed to execute the PGT and its corresponding PGT completion time T_{M^*}. Initially, the partitioning algorithm lets $M = N$ with each vertex being an individual partition. The algorithm then iteratively decreases M through partition merging (line 11 in Algorithm 1). This is equivalent to keeping the PGT completion time T monotonically non-increasing as exemplified in Figure 3. See *Theorem 1* for a proof. Therefore, a partition scheme that produces M^* ideally achieves the minimum PGT completion time T^*, thus $T_{M^*} = T^*$ under the current graph partitioning algorithm. This follows the "data locality" principle which suggests that the unit cost of data movement between two partitions is far greater than that within the same partition. Therefore, fewer partitions lead to faster completion with less data movement, resource usage and lower operational cost.

On the other hand, a smaller M^* corresponds to a greater resource demand per partition since more Drops are allocated to each partition. This means the aggregated resource demand from concurrently-running Drops in a given partition is more likely to exceed C, slowing down the graph execution due to resource over-subscription. An ideal partitioning solution not only obtains an optimal M^* but also ensures that resource demands in all partitions stay below C at any point during the graph execution. Satisfying this constraint avoids unpredictable execution delay due to resource over-subscription, thus ensuring $T_{M^*} = T^*$. Formally, the graph partitioning is formulated as a constrained optimization problem:

$$\begin{aligned} \min_{p} \quad & M(PGT, p) \\ \text{s.t.} \quad & R_i(t) \quad \leq \quad C, \quad i = 1, \ldots, M., \quad \forall t \in \Big[0, T(PGT, p)\Big] \end{aligned} \tag{1}$$

where $M(\cdot)$ is a function that outputs the number M of partitions given a PGT and a partition solution p. $T(\cdot)$ is a function that outputs the completion time T given a PGT and a partition solution p. $R_i(t)$ denotes the aggregated resource demand from all running Drops in partition i at time t. We refer to the constraint defined in Equation 1 as the *DoP constraint*, where "DoP" stands for *Degree of Parallelism*. Figure 2 exemplifies partitioning solutions that do (not) satisfy the DoP constraint.

Once the optimal graph partitioning solution p is available, both M^* and T_{M^*} (i.e. T^*) are used by the telescope manager for the generation of observation and computing resource schedules well before the observation takes place.

4.1 Partitioning Algorithm

The main idea of the partitioning algorithm (Algorithm 1) is to iteratively reduce data movement between inter-node Drops by "merging" them into the same node, where the cost of intra-node communication is negligible. Given a PGT g, the algorithm sorts all edges in g based on their weights in a descending order. The edge weight here denotes the volume of data "on the move" from one Drop to the next. Each drop is initially allocated to a separate node. Then going through all edges in a descending order of their weights, the algorithm merges two partitions associated with the two Drops on both ends of the edge if the merged partition meets the DoP constraint defined in Equation 1. The merging sequence is somewhat "greedy" since it looks for larger edge costs before dealing with smaller ones. However, this may not necessarily lead to a globally optimal solution especially for large graphs. Therefore we also developed

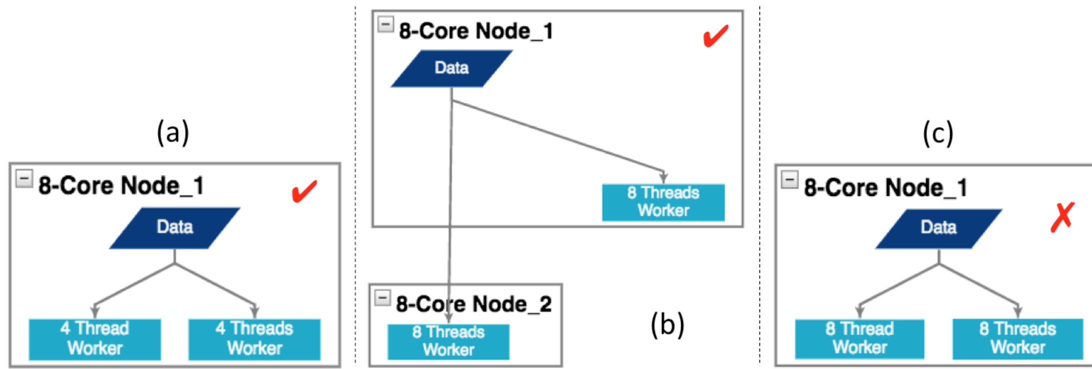


Figure 2: Three solutions to partitioning a simple fork-like Physical Graph Template. Solution (a) places all three Drops inside a single partition. So the two worker Drops will run in parallel after the data becomes available, thus consuming 8 cores (four threads each) at the same time. This satisfies the DoP constraint given the resource capacity C for a compute node includes 8 cores. However, solution (c) does not satisfy the DoP constraint since at some point 16 threads will be running in parallel on a single 8-Core machine. Consequently, the expected completion time for either worker is no longer guaranteed due to resource over-subscription. To remedy this, solution (b) separates the two worker Drops in two different partitions, each of which has sufficient resource capacity to execute 8 threads. Although the data movement between the two partitions incurs additional cost compared to Solution (c), Solution (b) produces far more reliable estimates on both completion time and resource demands with a potentially shorter completion time thanks to adequate resource provisioning.

several alternative merging sequences based on local search heuristics such as simulated annealing, Monte Carlo Tree Search, particle swarm optimization. For some PGTs, we find these heuristics obtain slightly better merging sequences, but often at an excessive running cost orders of magnitude higher than Algorithm 1. We are currently working on quantitative evaluations of the trade-offs between the merging sequence optimality and the running cost of these heuristics.

Although the iterative edge zeroing procedure is based on the graph clustering algorithm Sarkar (1987), we added two important additional changes. First we allow two existing partitions to re-merge again in order to further reduce the number of partitions, which in turn reduces the total completion time as suggested in *Theorem 1*. Second, we evaluate the DoP constraint in order to accept or reject partition merging (line 12) proposals. The evaluation of the DoP constraint not only considers each graph vertex's processing requirement in terms of maximum number of concurrent threads, memory usage, etc., but also incorporates predefined resource capacities for each partition including number of cores, memory capacity, etc.

Theorem 1. *The edge zeroing statement at line 10 in Algorithm 1 ensures the completion time T of g is strictly non-increasing.*

Proof. If the edge e is on the longest path L of g with a length T, there are two possibilities after e's weight becomes zero — L remains the longest path of g or another path L' becomes the longest path of g. In the first case, let T' be the new length of L. It is easy to verify that $T' == T - e.weight < T$. In the second case, let T' be the length of L'. It must be true that $T' \leq T$ because otherwise L' (rather than L) would have been the longest path before the edge zeroing takes place.

If the edge e is not on the longest path L of g, there are also two possibilities after e's weight becomes zero — e remains off the longest path L of g or e becomes part of the "new" longest path L' of g. In the first case, since L is not affected whatsoever, its completion time T remains

input : A DAG g with a list el of edges and a list nl of nodes
output : A list l of partitions

```
1  initialise l as an empty list
2  el.sort_by_weight(reverse ← true)
3  foreach element n of nl do
4      part ← create_partition()
5      part.add(n)
6      l.add(part)
7  end
8  foreach element e of el do
9      origin_weight ← e.weight
10     e.weight ← 0                                   // edge zeroing
11     u, v ← e.nodes()
12     new_part ← try_merge_partition(l, u, v)
13     if new_part == NULL then
14         e.weight ← origin_weight
15     end
16 end
17 return l
```

Algorithm 1: The partitioning algorithm based on Sarkar (1987) with two important additions — evaluation of the DoP constraint and merger of existing partitions

the same, thus non-increasing. In the second case, let T' be the length of L'. It must be true that $T' \leq T$ because otherwise L' (rather than L) would have been the longest path before the edge zeroing takes place. □

4.2 DoP Constraint Evaluation

In this subsection, we discuss the DoP evaluation algorithm defined in the `try_merge_partition` function called at line 12 in Algorithm 1. As shown in Equation 1, this boils down to efficiently computing the total resource usage $R_i(t)$ summed over all running Drops inside a given partition i at a particular time t. To do this, we first establish the equivalence between the set $D(t)$ of Drops running in parallel at time t and the concept of *antichain* Marcus (2008) — a set of *mutually unreachable* vertices of a DAG g associated with a given partition.

Theorem 2. *If all Drops in $D(t)$ are running in a non-streaming mode, $D(t)$ is an antichain of g.*

Proof. The non-streaming running mode excludes one possible form of parallelism — pipelining. All other forms of parallelisms require Drops in $D(t)$ be mutually unreachable on r because otherwise they would never have been running in parallel due to their inter-dependencies as a result of reachability. □

We define the *length* L of an antichain $D(t)$ as the number of Drops in $D(t)$, and define the *weighted length* W of an antichain $D(t)$ as the aggregated weight summed over all Drops in $D(t)$. The weight of the jth Drop d_j in an antichain is the pre-determined peak resource usage denoted by $w(d_j)$. Let A denote the set of all antichains in an partition graph i. It then follows

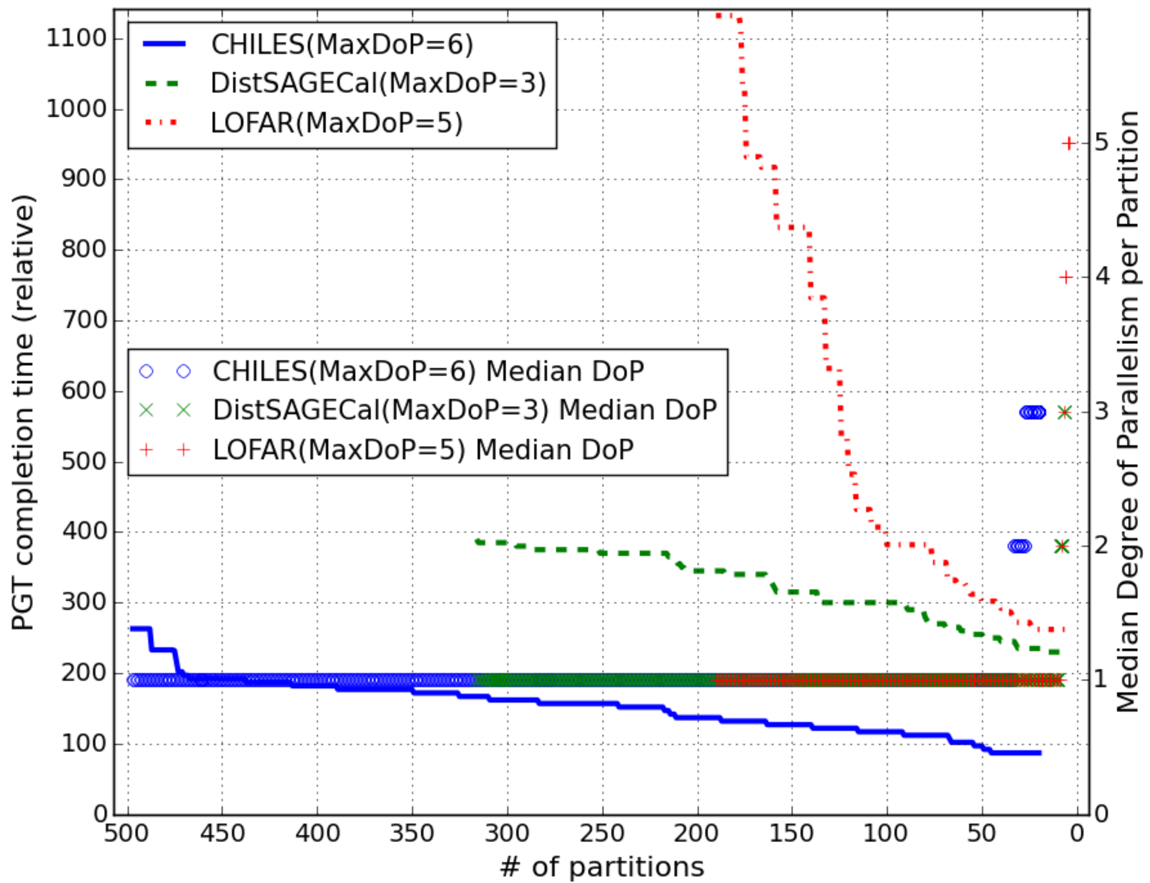


Figure 3: The PGT completion time is monotonically non-increasing as the number of partitions decreases for three different radio astronomy pipeline graphs. It also shows the partition solution that produces the minimum number M^* of partitions (i.e. the bottom right end of each curve) also results in the shortest execution time T^*.

from Theorem 2 that the total resource usage $R_i(t)$ is bounded by some antichain(s) D that has the maximum (longest) weighted length amongst all antichains in A:

$$R_i(t) \leq W_{max} = \max_D \sum_{j=1}^{L} w(d_j),$$
$$\text{where } d_j \in D,\ L = |D|,\ D \in A,\ \forall t \in \Big[0, T(PGT, p)\Big] \tag{2}$$

Equation 2 bounds a time-dependent value $R_i(t)$ by a time-invariant constant W_{max} such that if $W_{max} \leq C$ for a given partition, the constraint condition $R_i(t) \leq C$ in Equation 1 will be satisfied. However, finding the antichain D^* that produces W_{max} is not trivial since the cardinality of A — the total number of antichains in a partition graph g — can be in the order of 2^n, with n being the number of vertices in g. Therefore, enumeration and evaluation of all antichains is computationally unfeasible in practice, where a typical partition has at least tens or even hundreds of tasks (e.g. there could be up to one billion antichains for a graph with merely 30 vertices).

To compute the maximum antichain length for a given graph in polynomial time, one can apply Dilworth's Theorem Dilworth (1950), which states that the maximum length of an antichain is equal to the minimum number of chains needed to fully "cover" the graph. In particular Fulkerson Fulkerson (1956) established the equivalence between the maximum antichain length and the maximum matching in a constructed split graph (a.k.a. bipartite graph). As a result, the longest antichain — the antichain that has the maximum cardinality — of a graph can be discovered in $O(|E|\sqrt{|V|})$ time. However, Equation 2 suggests that the longest antichain does not necessarily have the longest weighted length unless $w(d_j) = 1, \forall j \in [1, L]$. Hence, whilst we can efficiently solve W_{max} for a special case where each Drop consumes only one unit of resource (e.g. 1 core, 1G of RAM, etc.), we need a different algorithm to evaluate more generic cases where Drops consume arbitrary units of resources (e.g. 16 cores, 375 MB of RAM).

In the following, we discuss details of Algorithm 2 that efficiently computes W_{max} for generic cases based on Cong (1993) to compute a maximum weighted k-family. While a k-family covers a union of at most k antichains in a DAG, we are interested only in a special case (where $k = 1$) in order to solve our problem of computing the maximum weighted length of a single antichain.

```
input  : partitions A and B with their associated DAGs g_A and g_B
         an optional g_dag representing the unpartitioned physical graph template
output : the maximum weighted antichain length of the merged partition A ⋃ B

1  function get_pi_solution(g):
2      S ← create_split_graph(g)
3      H ← admissible_graph(S)
4      f' ← maximum_flow(H, s, t)
5      R ← residual_graph(H, f')
6      foreach element r_node ∈ R.nodes() do
7          if R.has_path(s, r_node) then pi[r_node] ← 0
8          else pi[r_node] ← 1
9      end
10     return pi
11 end
12 pi ← get_pi_solution(g_A ⋃ g_B)
13 W_max ← 0
14 for h ← 0 to 1 do
15     foreach element nd_x ∈ S.X do
16         nd_y ← S.counter_part(nd_x)
17         if h = 1 − pi[nd_x] + pi[node_S] and 1 = pi[nd_y] − pi[nd_x] then
18             W_max ← W_max + S.edge(node_S, nd_x).capacity
19         end
20     end
21 end
22 return W_max
```

Algorithm 2: Calculate W_{max}, the maximum weighted antichain length in a partition

The central idea of Algorithm 2 is to exploit the equivalence between the weighted maximum anti-chain of the original DAG g and the minimum-cost maximum-flow (MCMF) solution of the split graph S created at Line 2. The equivalence is proved in Cong (1993) and more generally in Cameron (1985). Note that number of nodes of S is $2V + 2$ where V is the number of the

original DAG, which is the union g of the two DAGs g_A and g_B. This ensures that a polynomial algorithm on S remains tractable on g.

To find the MCMF solution, we first derive the admissible graph H from S (line 3), and run the normal maximum flow algorithm Goldberg and Tarjan (1988) to obtain the flow f' in $O(V^2\sqrt{E})$ time (line 4). We then construct the residual graph R from f' (line 5). R has the identical set of vertices as H, and if there are no edges going from the source vertex s of R to some vertex x, then we set the node potential π of x to 1 (line 8). In the end, the maximum weighted antichain W_{max} is calculated (line 14 to 20) based on expressions defined in Theorem 3.1 Cong (1993). Figure 3 shows the results of running Algorithm 1 and 2 by scheduling three different radio interferometry imaging workflows.

5 Conclusions

Optimal scheduling of large-scale, data-intensive workflows is challenging. In this memo, we discussed related work on graph scheduling and proposed polynomial time optimization methods that minimize both workflow execution time and resource footprint while meeting resource demand constraints imposed by individual algorithms. We show preliminary results obtained from three radio astronomy data pipelines.

This is the first memo on the optimal mapping of DALiuGE dataflows to homogeneous resources. The next memo, to be released in the near future, will discuss a different set of technical solutions that we have developed in order to deal with heterogeneous resources.

List of Figures

1 The complete dataflow execution cycle consists of four major steps — unrolling, partitioning, mapping and dynamic scheduling. Both unrolling and partitioning are performed offline. Mapping happens just a few minutes before the workflow execution, and dynamic scheduling is done in real-time during execution. While the first three steps target the entire graph across multiple resources, the last step focuses on tasks using local resources on a single node. 5

2 Three solutions to partitioning a simple fork-like Physical Graph Template. Solution (a) places all three Drops inside a single partition. So the two worker Drops will run in parallel after the data becomes available, thus consuming 8 cores (four threads each) at the same time. This satisfies the DoP constraint given the resource capacity C for a compute node includes 8 cores. However, solution (c) does not satisfy the DoP constraint since at some point 16 threads will be running in parallel on a single 8-Core machine. Consequently, the expected completion time for either worker is no longer guaranteed due to resource oversubscription. To remedy this, solution (b) separates the two worker Drops in two different partitions, each of which has sufficient resource capacity to execute 8 threads. Although the data movement between the two partitions incurs additional cost compared to Solution (c), Solution (b) produces far more reliable estimates on both completion time and resource demands with a potentially shorter completion time thanks to adequate resource provisioning. 7

3 The PGT completion time is monotonically non-increasing as the number of partitions decreases for three different radio astronomy pipeline graphs. It also shows the partition solution that produces the minimum number M^* of partitions (i.e. the bottom right end of each curve) also results in the shortest execution time T^*. 9

List of Tables

References

ATNF: ASKAPsoft processing of ASKAP data, https://www.atnf.csiro.au/computing/software/askapsoft/sdp/docs/current/pipelines/introduction.html, 2018.

Bateni, M., Behnezhad, S., Derakhshan, M., Hajiaghayi, M., Kiveris, R., Lattanzi, S., and Mirrokni, V.: Affinity Clustering: Hierarchical Clustering at Scale, in: Advances in Neural Information Processing Systems, pp. 6867–6877, 2017.

Bokhari, S. H.: A shortest tree algorithm for optimal assignments across space and time in a distributed processor system, IEEE transactions on Software Engineering, pp. 583–589, 1981.

Braun, R., Bourke, T., Green, J., Keane, E., and Wagg, J.: Advancing Astrophysics with the Square Kilometre Array, Advancing Astrophysics with the Square Kilometre Array (AASKA14), 1, 174, 2015.

Cameron, K.: Antichain sequences, Order, 2, 249–255, 1985.

Chaudhary, V. and Aggarwal, J. K.: A generalized scheme for mapping parallel algorithms, IEEE Transactions on Parallel and Distributed Systems, 4, 328–346, 1993.

Cong, J.: Computing maximum weighted k-families and k-cofamilies in partially ordered sets, Computer Science Department, University of California, 1993.

Cornwell, T., Wortmann, P., Harding, P., Tamerus, A., Stolyarov, V., Farreras, M., Hadjigeorgiou, C., Tobar, R., and Dulwich, F.: Algorithm Reference Library, https://github.com/SKA-ScienceDataProcessor/algorithm-reference-library, 2018.

Dennis, J. B. and Misunas, D. P.: A preliminary architecture for a basic data-flow processor, in: ACM SIGARCH Computer Architecture News, vol. 3, pp. 126–132, ACM, 1975.

Dilworth, R. P.: A decomposition theorem for partially ordered sets, Annals of Mathematics, pp. 161–166, 1950.

Fernández, X., Van Gorkom, J., Hess, K. M., Pisano, D., Kreckel, K., Momjian, E., Popping, A., Oosterloo, T., Chomiuk, L., Verheijen, M., et al.: A pilot for a very large array hi deep field, The Astrophysical Journal Letters, 770, L29, 2013.

Fulkerson, D. R.: Note on Dilworth's decomposition theorem for partially ordered sets, in: Proc. Amer. Math. Soc, vol. 7, pp. 701–702, 1956.

Goldberg, A. V. and Tarjan, R. E.: A new approach to the maximum-flow problem, Journal of the ACM (JACM), 35, 921–940, 1988.

Grange, Y.: Prefactor dockerfile for DaLiuGE demonstration, https://github.com/ygrange/prefactor-DaLiuGE, 2018.

Guzman, J.: Status of the ASKAP software and preparation towards SKA construction, in: Computing for SKA Colloquium 2018: Towards Construction, Institute for Radio Astronomy and Space Research, AUT, https://irasr.aut.ac.nz/__data/assets/pdf_file/0009/151002/JC-Guzman-C4SKA18-ASKAP-updated.pdf, 2018.

Horneffe, A.: A. Prefactor: Pre-facet calibration pipeline, https://github.com/lofar-astron/prefactor, 2017.

Karypis, G. and Kumar, V.: Multilevelk-way partitioning scheme for irregular graphs, Journal of Parallel and Distributed computing, 48, 96–129, 1998.

Kwok, Y.-K. and Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Computing Surveys (CSUR), 31, 406–471, 1999.

Liou, J.-C. and Palis, M. A.: A comparison of general approaches to multiprocessor scheduling, in: Proceedings of the 11th International Parallel Processing Symposium, pp. 152–156, IEEE, 1997.

Marcus, D.: Graph theory: a problem oriented approach, The Mathematical Association of America, 2008.

Martella, C., Logothetis, D., Loukas, A., and Siganos, G.: Spinner: Scalable graph partitioning in the cloud, in: IEEE 33rd International Conference on Data Engineering, pp. 1083–1094, IEEE, 2017.

Nan, R.: Five hundred meter aperture spherical radio telescope (FAST), Science in China series G, 49, 129–148, 2006.

Ord, S., Tobar, R., Wu, C., Devereux, D., and Dolensky, M.: Joint Astronomy CALibration and imaging software, https://github.com/ICRAR/jacal, 2018.

Rocklin, M.: Dask: Parallel computation with blocked algorithms and task scheduling, in: Proceedings of the 14th Python in Science Conference, 130-136, Citeseer, 2015.

Sarkar, V.: Partitioning and scheduling parallel programs for execution on multiprocessors, Ph.D. thesis, 1987.

Stone, H. S.: Multiprocessor scheduling with the aid of network flow algorithms, IEEE transactions on Software Engineering, pp. 85–93, 1977.

Tobar, R.: Integration of ARL with the DALiuGE Execution Framework, https://confluence.ska-sdp.org/display/WBS/Integration+of+ARL+with+the+DALiuGE+Execution+Framework, 2018.

Tobar, R. and Kitaeff, S.: Realtime data reduction pipeline for spectral observations on FAST radio telescope, https://github.com/ICRAR/FAST-HI, 2017.

Topcuoglu, H., Hariri, S., and Wu, M.-y.: Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems, 13, 260–274, 2002.

Towsley, D.: Allocating programs containing branches and loops within a multiple processor system, IEEE Transactions on Software Engineering, pp. 1018–1024, 1986.

Tsourakakis, C., Gkantsidis, C., Radunovic, B., and Vojnovic, M.: Fennel: Streaming graph partitioning for massive scale graphs, in: Proceedings of the 7th ACM international conference on Web search and data mining, pp. 333–342, ACM, 2014.

Vinsen, K., Dodson, R., Foster, S., and Tobar, R.: aws-chiles02, https://github.com/ICRAR/aws-chiles02, 2018.

Wei, S., Wang, F., Wu, C., and Tobar, R.: Using DALiuGE for Distributed SAGECal with DynlibAppDrop and MemoryDrop, https://github.com/astroitlab/sagecal-daliuge-dynlib, 2018.

Wicenec, A., Pallot, D., Tobar, R., and Wu, C.: DROP Computing: Data Driven Pipeline Processing for the SKA, in: Astronomical Society of the Pacific Conference Series, edited by Lorente, N. P. F., Shortridge, K., and Wayth, R., vol. 512 of *Astronomical Society of the Pacific Conference Series*, p. 319, 2018.

Wu, C., Tobar, R., Vinsen, K., Wicenec, A., Pallot, D., Lao, B., Wang, R., An, T., Boulton, M., Cooper, I., et al.: DALiuGE: A graph execution framework for harnessing the astronomical data deluge, Astronomy and Computing, 20, 1–15, 2017.

Yatawatta, S.: Distributed radio interferometric calibration, Monthly Notices of the Royal Astronomical Society, 449, 4506–4514, 2015.