Liberating the Data Aware Scheduler to Achieve Locality in Layered Scientific Workflow Systems

Colin Thomas[1], Douglas Thain[2]
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN
cthoma26@nd.edu, dthain@nd.edu

***Abstract*—Large scale scientific workflows are typically run using a task-based distributed workflow system. Many contemporary workflow systems are organized in a layered or modular fashion, with one component managing the DAG or workflow graph structure, and another component acting as the scheduler or executor. These components communicate and regulate the flow of tasks from the application to the remote execution site. Scientific workflows perform significant amounts of I/O, typically using an HPC parallel filesystem. Node-local storage technologies are often used as an effective supplement to parallel filesystems, relieving them of the large amount of bandwidth consumed by intermediate data reads and writes which would otherwise cause I/O bottlenecks. Node-local storage techniques depend upon effective scheduling to place tasks close to their necessary data, thus benefiting from data locality. Effective locality based scheduling is a challenge however. The conventional layered architecture results in the scheduler considering tasks on an individual basis with a narrow view of the greater DAG. We present a modified architecture of a workflow system which allows the efficient construction of dependency-based task groups which are passed through the DAG manager, scheduler, and finally to the remote worker node where improved use of data locality can be achieved. These modifications were implemented using the Parsl parallel library and TaskVine execution engine. We evaluate this implementation with a benchmark application and a Montage workflow. We compare the results between a conventional data-aware scheduler, our task grouping implementation, and a workflow system using only shared storage. We find that task grouping achieves a significantly greater degree of data locality, thereby improving performance and reducing total data movement in the cluster when compared to the other two methods.**

***Index Terms*—HPC, Scientific Computing, Workflows, Storage, Node Local Storage, Scheduling**

I. Introduction

Modern HPC workflows involve large amounts of tasks and considerable amounts of I/O. Generally HPC facilities provide a parallel filesystem which is accessible by all nodes across the cluster such that remote nodes can access input data and write their outputs to a shared directory. Certain I/O patterns which are intense in metadata operations may cause the parallel filesystem performance to be the limiting factor in overall workflow performance. There are a number of research efforts which advocate for the cooperative use of node-local storage in HPC workflows in order to keep data closer to the cluster nodes, and relieve the load from the parallel filesystem [1] [2] [3].

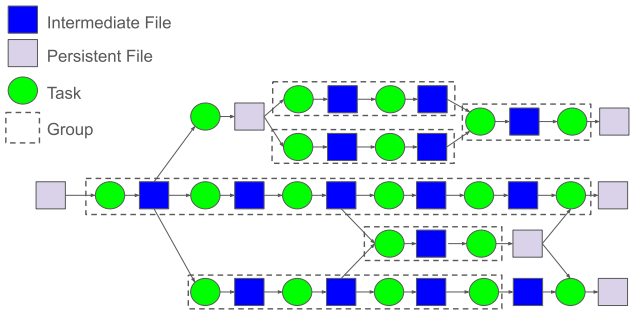


Fig. 1: Example Workflow DAG.
Task groups are outlined in boxes. This simple annotation replaces the work of a complex scheduling algorithm.

These local-storage techniques are most effective when complemented by data-aware scheduling that can place tasks at the location where dependent data is cached, therefore benefiting from data locality [4]. However, this scheduling pattern is a challenge in a dynamic distributed execution where resources fluctuate, and new tasks may appear at any time. Many HPC workflow systems are constructed in a layered or modular fashion, one component being the DAG manager which understands the task dependencies and arranges them into a directed-acyclic-graph, or DAG. The DAG manager then releases tasks whose dependencies are ready to the scheduler or executor, which is responsible for dispatching tasks to the pool of remote worker resources. This fundamental pattern of communication between a typical DAG manager and scheduler complicates the synthesis of upcoming task dependency information and the placement of data in node-local storage. **The main issue is that the DAG manager only reveals to the scheduler tasks which are ready to run, leaving the scheduler unaware of future task dependencies.** Given a data-aware scheduler, this future task dependency information would allow the scheduler to assign remote workers for long dependency chains, liberating the scheduler from the communication impediment and allowing it to make full use of its capabilities. We present a modified workflow system which identifies these dependency chains at the DAG manager, passing them to the scheduler to form task groups. This allows the scheduler to easily assign batches of dependent tasks to

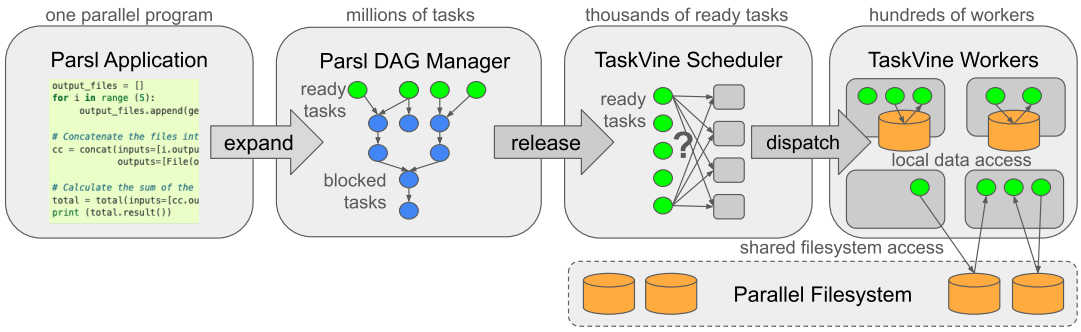


Fig. 2: Workflow System Architecture

The application creates tasks which are arranged into a graph by the DAG manager. The ready tasks are released to the scheduler, which are then dispatched to workers. The workers have local storage as well as shared access to a parallel filesystem.

```
import parsl
from parsl.data_provider.files import File

f_image_list = File('imglist')
f_output = File('taskvinetemp://images.tbl')
f_mosaic = File('m17.fits')

output_table = create_meta_table(
            inputs=[f_image_list],
            outputs=[f_output])
mosaic = add_to_mosaic(
            inputs=[output_table.outputs[0]],
            outputs=[f_mosaic])
mosaic.result()
```

Fig. 3: Example Parsl Application

A simple Parsl application. Data dependencies are expressed in the function invocations, and "result" is called on the final task in the workflow.

individual nodes, achieving the optimal level of data locality and reducing scheduling overhead.

We implement these concepts using the Parsl [5] parallel library and TaskVine [3] workflow execution engine. Our implementation is evaluated using a benchmark application, as well as a practical scientific application from the field of astronomy. The evaluation applications were chosen to illustrate the functional difference between three workflow execution methods. We first compare the task-grouping implementation to our standard TaskVine data-aware scheduler without group scheduling or the forward knowledge provided by the modified DAG manager. We also compare these methods to an independent contemporary workflow system using an HPC shared filesystem instead of node-local storage in order to provide a context in which many HPC workflows are currently executed. We find that task grouping achieves a significantly greater degree of data locality when compared to the conventional scheduler. This increased data locality results in significant performance benefits in comparison with both the conventional scheduler and shared filesystem implementation.

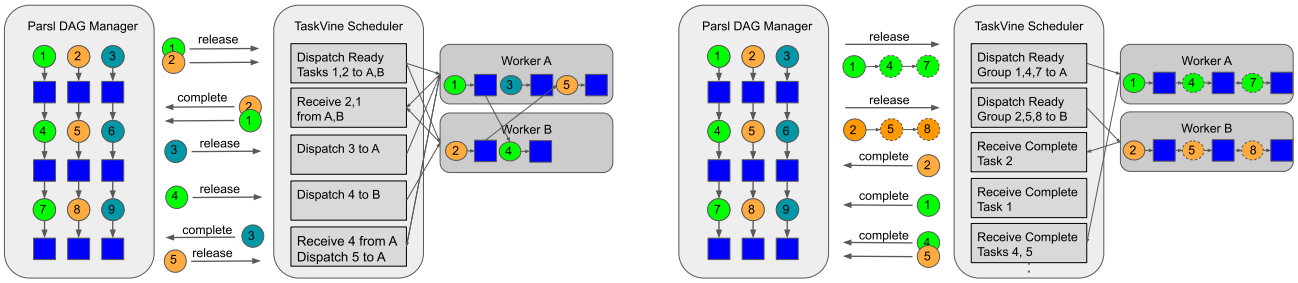
II. Architecture

This work was built upon TaskVine [3] and its data-aware scheduling capabilities. TaskVine is a workflow description framework and execution engine for data-intensive HPC applications with a focus on local storage management. TaskVine may be used as a standalone workflow system through a Python or C API, or as an executor module for a number of different higher-level workflow DAG managers. In this work, we use the Parsl [5] parallel library as our DAG manager coupled with the TaskVine executor for scheduling and resource management.

The architecture of TaskVine and Parsl is described in Figure 2. The first component is the application. A user will write the application to describe their workflow, creating individual tasks and annotating them with data dependencies and products. There are multiple ways for a user to invoke execution. Tasks may be called to run individually, or the final tasks in the workflow may be called in the style of futures, such that each prior task must be run in order to obtain the final result. It is the latter method which is the most concise and optimal for the workflow system, as this ensures any issues with the declaration of data dependencies will be identified. A simplified example of a Parsl application is seen in Figure 3. The order of execution is indicated by specifying data dependencies, or that the input of one task is the output of another. The DAG is realized when the final task of the series is invoked.

The application can be seen producing tasks to send to the Parsl DAG manager, or data-flow-kernel, which constructs a graph of the workflow identifying the order of execution. Ready tasks which are colored in green are those whose dependencies are fulfilled. At the beginning of the workflow the ready tasks are only at the top level of the DAG. The DAG manager is the central director of the workflow system, and aims to maintain awareness of all things pertaining to the execution such as task statuses and the creation of data. The workflow DAG is built around data dependencies, so when tasks complete and output is created it indicates a new set of tasks are ready to run. In the event of task failure the capability of the DAG manager to respond is limited. It may choose to retry the failed task or give up. Since it is largely disconnected from the cluster resources and data locations it must delegate more complex recovery methods to the scheduler.

The ready tasks are passed by the DAG Manager to the TaskVine scheduler. TaskVine is responsible for the main-



(a) Individual Task Scheduling

(b) Group Task Scheduling

Fig. 4: Comparing Individual and Group Task Scheduling

Tasks are typically scheduled on an individual basis without using the context contained in the DAG or other tasks in the queue. This allows the potential for missed data locality. Group task scheduling identifies data locality opportunities and schedules them together.

tenance of the worker pool, understanding the available resources, and is also aware of task data dependencies and where they reside on node local disks. Using this information the TaskVine scheduler will determine where to send the tasks it has been given by the DAG manager. Both the application and the workers have access to a parallel filesystem, where workers may be directed by the application to retrieve and create data. Each worker also possesses a local disk where it may cache data. TaskVine provides a number of capabilities regarding the management of node-local storage in the cluster. Individual files are tracked by which workers are in posession of them, and the scheduler may take advantage of this information by sending tasks to workers who posess some or all of the required input data. TaskVine can direct remote transfers to occur between workers in the cluster, such to avoid a costly transfer from an outside source, or to distribute a large amount of files in a peer-to-peer fashion to avoid a large distributed read from the shared filesystem. The management of data in TaskVine also enables efficient fault tolerance and recovery when tasks fail or resources are lost. Data replication may be configured such that files which are costly to produce are replicated across multiple worker nodes. Thus, in the event a worker is lost its valuable data will survive on other workers and recovery can begin without the costly regeneration of the data. If the data has been lost altogether, before conceding the task has failed to the DAG manager, TaskVine may attempt to recreate the data by rescheduling the tasks which produced it.

III. The Data Locality Challenge

It is important for us to clearly identify the limitations of conventional task scheduling when aiming for data locality benefits. We have introduced TaskVine and its data-aware scheduling capabilities. We must address why simple data-awareness in the scheduler is not sufficient to achieve an optimal execution at scale.

Figure 4a shows a partial workflow execution with task placements made by the scheduler which miss opportunities for data-locality. Ideally each one of the three task sequences should be placed on a specific worker so their intermediate data remains local. The problem is easily illustrated when the amount of concurrent work available exceeds the capacity of the worker resources. There are 3 sequences and only 2 workers, so some tasks cannot be scheduled immediately. The problem begins when the scheduler is considering task 3. From the perspective of the scheduler the placement of task 3 is inconsequential since there are no input dependencies for which to seek locality benefits. Task 3 is arbitrarily placed, and when the scheduler considers task 4 the ideal worker is now occupied. Determined to schedule task 4 it places it on the other worker, resulting in a missed locality opportunity.

By considering tasks on an individual basis the scheduler is bound to occupy workers who possess dependencies for other tasks, delaying their execution. One solution for this may be to do away with the absolute scheduling deadline and relax the policy such that if no scheduling opportunity is available after 5 attempts, then concede to scheduling a task separate from the data. This is in fact a description of delay scheduling [6] which was shown to be generally effective. The problem remains however, that the scheduler will occupy workers which will then cause these delays to occur. The workflow will only benefit by the proportion of tasks which were successfully scheduled by data-locality. The cost of scheduling a task away from its data can vary significantly depending on the data size and bandwidth of the network it will need to be transferred on. While perhaps trivial for short running tasks with small dependencies, this can become more significant with greater scale and size of data.

Rather than finding a perfect balance of determination vs. concession at the scheduler which may become ineffective when faced with a different application with different task runtimes and I/O behavior, **we intend to avoid this scheduling challenge altogether by exploiting our knowledge of the DAG to make task associations ahead of time**. While delay scheduling is an effective method for effective scheduling of short running on-demand tasks, our system can use the knowledge of some or all of the workflow DAG up front, allowing it to quickly make group associations to pass to the scheduler. This is illustrated in Figure 4b, where a partial execution of the prior workflow is shown except the DAG manager is communicating whole task groups to the scheduler. This will enable the scheduler to achieve locality, but how should these groups be constructed?

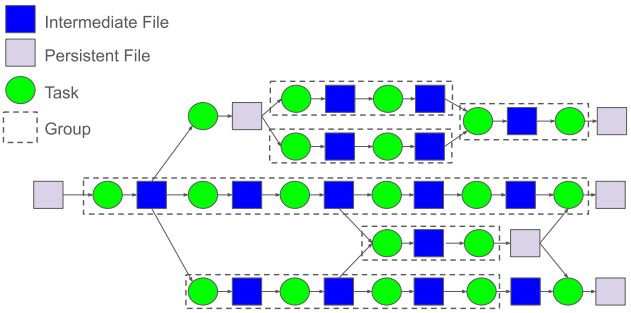


Fig. 5: Workflow DAG With Groups.

IV. Task Grouping

Consider a workflow described by the DAG in Figure 5. This DAG features 6 distinct task sequences which are outlined by boxes. We will consider these sequences to be our groups with the intent of each group executing with perfect use of data locality.

Let us assume the resource requirements of these tasks are uniform, and each task will occupy the resources of one entire node. Even if there is a large pool of remote workers available it is almost certainly the case that the optimal execution of this workflow would involve no more than 5 nodes, as there is no potential for greater concurrency. Using local storage, ideally each node will be responsible for a single sequence at a time, caching the intermediate data for fast reuse. From our perspective we can rather simply formulate an optimal execution for this DAG given 5 worker nodes. Consider the perspective of the scheduler in this execution, and how it is only provided knowledge of the immediately ready to run tasks. The task groups vary in length and some begin before others, leaving some nodes idle for periods of time. As the number of tasks in the system increases, the scheduler considering each ready task individually may choose to schedule a task which breaks a potential locality chain, unaware of the significance this poses to the larger execution scheme. The DAG manager has a complete view of the DAG, just as complete as our view of Figure 5, yet it withholds this knowledge from the scheduler.

Consider a scheme in which the DAG manager offers these dependent chains of tasks to the scheduler. This allows the scheduler to easily recognize the sequences. Rather than scheduling individual tasks, the scheduler may send each sequence to a worker in a single operation. The worker needs only a simple capability to receive a queue of tasks which describes their order of execution. We propose a modified relationship between a DAG manager and data-aware scheduler in which dependency chains apparent to the DAG manager are revealed to the scheduler, enabling the scheduler to understand the flow of data through a particular series of tasks, and schedule this series with maximal use of data locality.

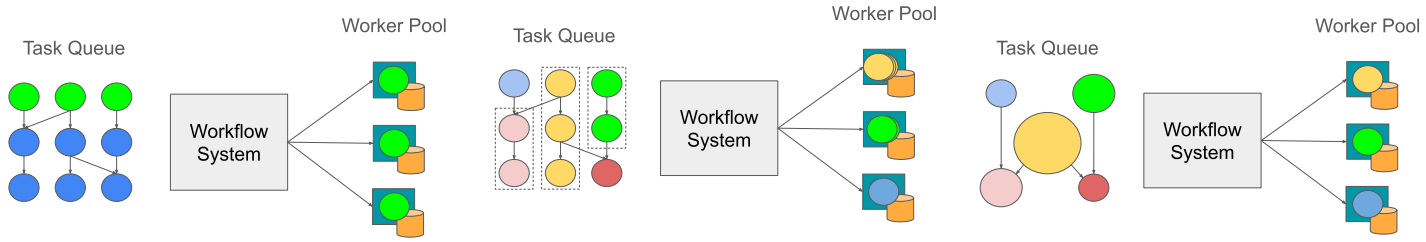
Enabling this capability is a multifaceted effort. The DAG manager must be modified to identify task sequences and communicate them to the scheduler. The scheduler must be able to recognize these sequences, and communicate them to workers who are capable of understanding execution order and data dependencies. The scheduler must maintain a balanced policy, maximizing the data locality benefits while still keeping the system moving and avoiding any prolonged periods of waiting to enforce an absolute policy. The main benefits of this system model are the reduction of I/O costs as a result of increased data locality, reduced inter-task latency by queuing tasks at the workers, and a reduced number of task communication events as a result of scheduling the tasks as groups. These benefits are not without potential disadvantages which will need to be addressed. Scheduling groups of tasks into queues at the workers releases a number of previously tightly-controlled variables into an unpredictable distributed system. In the case of task or worker failure it will be more complex to recover the system.

A. Grouping Strategy

There are multiple ways we may choose to associate tasks and present them to the scheduler. Figure 6 shows a simple DAG presented in three ways. Part **a.** shows the basic individual task scheduling strategy. Part **b.** shows groups in the DAG identified by color, yet the tasks retain their individuality. In contrast, part **c.** shows a merging, or fusing of tasks into larger abstract units. Let us consider these latter two methods and their potential advantages. The key difference is the granularity of tasks as presented to the scheduler and worker. In the case where task individuality is retained tasks remain lightweight and any mechanisms for fault recovery or checkpointing may still function efficiently, since upon failure we can identify which task had failed within the group and resume execution at the closest available checkpoint. In the merged abstract task method however, we simplify the presentation of the DAG to the scheduler, resulting in fewer overall scheduling operations and communications between the workflow system and workers, which may greatly benefit the system at scale. This benefit trades off the cost of checkpointing and fault tolerance becoming less granular, as we may no longer resume a sequence of tasks in the middle since they have been generalized into a larger task unit. This also requires the scheduler to be quite certain of its decision making process. There are many reasons some tasks may not be able, or should not be grouped together. Resource requirements may not align, or the DAG may be structured in such a way that a task in the middle of a group may require an outside dependency. Events may occur in the system where a previously feasible task group can no longer run. In order to recover the merged abstract task group must be deconstructed and reconsidered. With both strategies considered, we decide to move forward with the method of Figure 6**b.** in order to retain task individuality, enable fault recovery and flexibility of group configurations.

B. Resource Utilization

Tasks may have a diverse set of resource requirements. Some tasks may demand a large amount of cores, memory, or disk, while others request very few resources. Regardless



(a) Individual Tasks. (b) Group Task Queues. (c) Merging Task Sequences.

Fig. 6: Scheduling Task Sequences

Three methods of scheduling tasks. Part ***a.*** *shows the conventional method. Part* ***b.*** *and* ***c.*** *show two potential methods for grouping tasks, one preserving individuality in order to retain information, and one abstracting it in favor of simplicity.*

of the method used to group and schedule tasks, resource requirements must be considered and understood in order to make the best scheduling decisions. Since we are grouping tasks with the intention of a group running in a single location, the worker to which the group is sent must have sufficient resources to run the most demanding task in the group. There is a potential for inefficiency if we consider a sequence of tasks where the majority request few resources, yet one task requires a large enough allocation that would cause the entire sequence to be placed on a powerful worker. Therefore for a majority of the time the sequence is executing the worker resources are being underutilized. Whether or not this wasted resource utilization is detrimental to the workflow performance depends on the I/O benefits achieved by the group task, which would be determined by the size and quantity of intermediate data in the sequence, which is not necessarily known before execution. The workflow system may decide to terminate a task group when it encounters a task with significantly disparate resource requirements, or it may adopt a policy which favors grouping tasks above all resource considerations, hoping that I/O benefits will outweigh the cost of resources sitting idle. Both of these approaches may find advantage in different situations, and the task grouping implementation must make at some level an uninformed policy decision as to which approach to follow. It was decided in this work to place a priority on the formation of large, potentially diverse task groups as opposed to smaller uniform groups.

adjustbox

V. Implementation

A. TaskVine Modifications

Our implementation in TaskVine groups tasks on declaration by their intermediate data dependencies. In TaskVine files may be specified as intermediate data, which is referred to as ”temporary” being that their nature is ephemeral. They are deleted on completion of the workflow and they are not returned to the user. Creating task groups based on intermediate data follows a simple logic. When a new task is declared with an intermediate output it is considered for a group. If the task has no intermediate input then a new group is created with the task being the head of the group. When a task is declared with a intermediate inputs, we will assign it to the group associated with the task creating the input. Any outputs of tasks added to the group will be associated with the group, so subsequent tasks can be added. This method of group construction requires very little overhead or additional work since it is performed in the moments that tasks are declared. This performs optimally using the conventional workflow construction where the majority of tasks are declared up front before execution commences, however tasks can be declared and added to groups at any time over the course of the workflow.

There is a decision to be made regarding group construction and their maximum size. At the moment a group will be constructed for any length sequence of tasks so long as they all depend on associated intermediate data. This is simple for a straight line of dependent tasks, but becomes more complex when faced with branches such as in Figure 5, where intermediate dependencies within one group become the initial input of another group. The decision to be made here is whether to absorb the additional sequence into the original group to run in parallel, or to create a new group which will be scheduled separately. In our implementation the policy dictates there is to be no parallelism within a group. In effect, if there are two consumer tasks of a single intermediate data item, the two tasks should be in different groups. This prioritizes more numerous simple groups utilizing more individual workers rather than attempting to make decisions about effective parallelism at runtime in an application with indeterminate behavior.

Once some groups have been created and it is time to schedule tasks to workers, the scheduler will choose a task. The head of a group must be scheduled first since it is the first element in a chain of data dependencies. The TaskVine scheduler is implemented using a priority based queue of available tasks. When task groups are created the head of each group is given a high priority. As a result the head of each group is considered and scheduled quickly, even when the scheduler has been given a large amount of tasks to run. When the scheduler is sending a group task, it will send all of the tasks in the group at once to form a queue at the worker. This effectively reduces inter-task latency by allowing the worker to immediately begin the next task instead of waiting for instruction after each task completes.

Scheduling groups of tasks into queues at the worker requires modification to the TaskVine worker, since the worker previously operated under the assumption that it would only be sent tasks which can immediately run. Now it must be

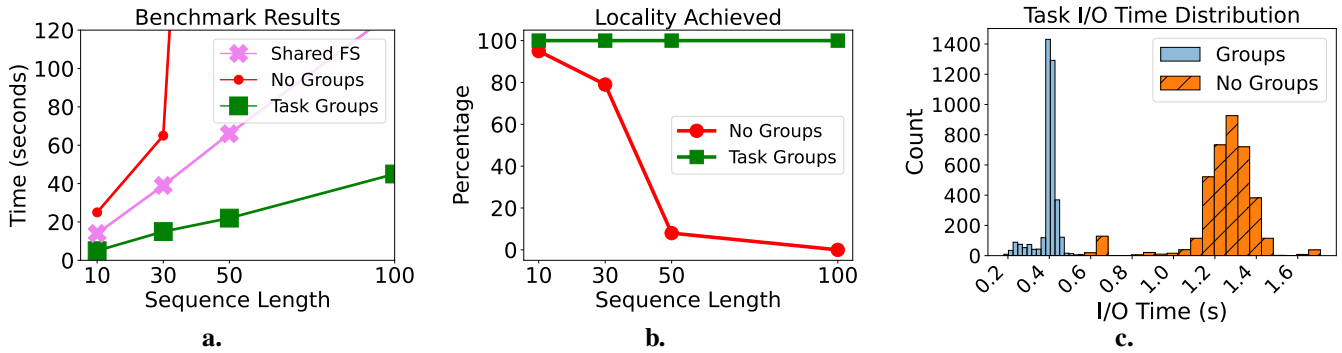


Fig. 7: Benchmark Results

able to receive a set of tasks from the manager which must be run in a particular order. When the worker considers the available tasks it will check the data dependencies required. If the worker is sent a sequence of tasks, and it considers a task which is not in the correct execution order it will find that the data dependencies cannot be fulfilled. Typically this will cause a task failure, since the task cannot be run without its dependencies. With the modification however, much like the TaskVine scheduler the worker will consider the other tasks it has been sent for the data dependency required, and if it finds that the data will be generated by another task in its queue it will carry on without failing the task. One may question whether having some order of execution implied by the scheduler when the group is sent to the worker would make execution more efficient, yet the cautious route was taken in not informing the worker about the nature of group scheduling. The simple routine of the worker is congruent with fault tolerance capability, which is especially important with the addition of task group scheduling. If one task in the sequence was to fail, the worker must recognize that all following tasks in the sequence can no longer be run at any time.

B. Parsl Modifications

Up to this point we have described the TaskVine implementation as if it were handed all of the tasks in the system, rather than only those which are ready to run as in the DAG manager executor scheme. When the TaskVine executor is being used with a DAG manager such as Parsl some modifications had to be made. Fortunately the architecture of Parsl lends itself well to this task, with their concept of data staging providers. Parsl allows users to write custom data staging providers, which define the behavior of different IO mechanisms such as FTP and HTTP file staging. While the utility here is not immediately obvious, the data staging provider is in fact the mechanism which notifies the Parsl DAG manager of the existence of output files, and therefore which task dependencies are fulfilled, influencing the set of ready tasks. For this work a custom data staging provider was written for TaskVine which ”fools” the Parsl DAG manager, or ”Data Flow Kernel” by reporting that all temporary output files have been created. This results in an interesting and mutually beneficial interaction between the two components. Given a workflow DAG with some potential group task sequences, Parsl will send to TaskVine all ready to run tasks as well as subsequent temporary-dependent tasks which may be placed into groups. These tasks may only be a subset of the entire DAG, allowing the scheduler to spend its time more effectively without iterating over later tasks and only focusing on the present tasks and creating groups.

VI. Evaluation

Our implementation was evaluated on a university HTCondor [7] cluster. All machines in the cluster are outfitted with SATA solid-state local disks and are interconnected by 10Gb/s ethernet links. Each worker used in this evaluation was a 12-core machine. The cluster is equipped with a Panasas ActiveStor [8] shared filesystem outfitted with 77 nodes and capable of 84 Gb/s read bandwidth and 94,000 read operations per second. Each evaluation application used the shared filesystem as its launch directory, and all FS operations that did not use node-local storage ocurred from this shared storage.

A. Benchmark Application

The benchmark application was constructed to assess the impact of task sequence length on three methods of workflow execution. All methods use Parsl to construct the DAG. ”No Groups” is an ordinary TaskVine execution in which the intermediate data is contained on node local storage. Tasks are dispatched one at a time, and only opportunistic data locality benefit is achieved since the DAG sections with grouping potential are not immediately revealed by the DAG manager. ”Task Groups” uses the implementation discussed, where the sequences are scheduled together in queues to each worker. ”Shared FS” uses Parsl with the standard High Throughput Executor, which directs workers to read and write data to a shared directory in the parallel filesystem. Thus, there is no data-aware component to the scheduler. Each run of the benchmark application consists of 20 task sequences of a set length. There are 20 12-core workers receiving the tasks, and each task occupies a whole worker. Each task in a sequence consumes a file from the previous task, and produces a new file which the next task depends on. The size of each intermediate data file is 200mB.

Figure 7 illustrates the behavior of the benchmark application. Part **a.** shows the scaling of the application and the effect on execution time. These results find task grouping to scale at

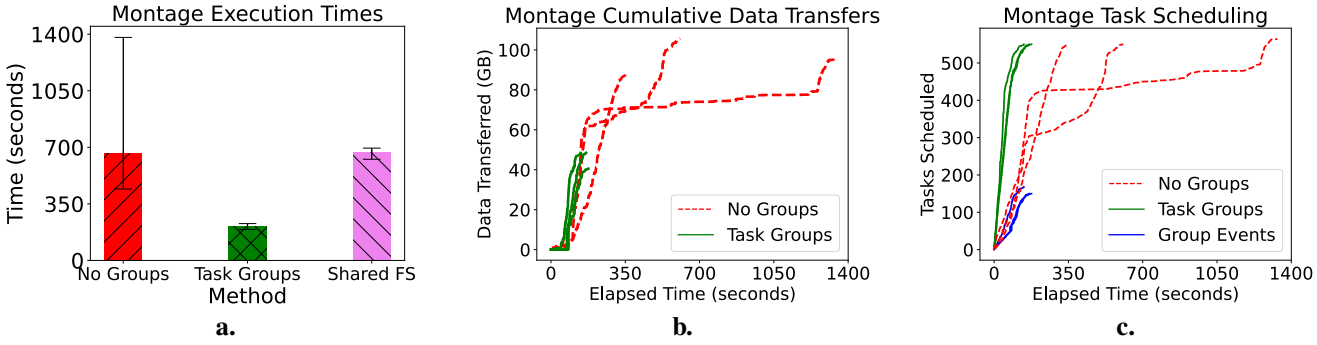


Fig. 8: Montage Results

an approximately linear rate in respect to sequence length. The shared FS runs also maintain linear scalability but at a greater coefficient. Without task groups, the data-aware scheduler is severely impeded as the number of tasks and intermediate data items grow. The cause of this is evident in parts **b.** and **c.** of Figure 7. Part **b.** shows the locality achieved with and without task grouping at each sequence length. The shared FS is not included since local storage is not used. The percentage of locality achieved is the amount of workflow data which remained local to one machine. Task groups maintain perfect locality across all benchmark scales where the ordinary data-aware scheduler quickly declines in effectiveness. Part **c.** is a histogram of the I/O latency across all benchmark runs. Task grouping results in a more narrow distribution concentrated at a range approximately a third shorter than the "no groups" distribution. Thus, the benchmark application benefits from task grouping by achieving improved data locality, therefore reducing overall task runtimes by eliminating the cost of network data transfers.

B. Montage

Montage [9] is a software from the domain of physics and astronomy which is used to assemble astronomical image data produced by telescopes into full images or mosaics. A typical Montage workflow begins with a set of FITS images. These images may go through several preprocessing and adjustment steps before being combined into the final mosaic. The set of input images, and their intermediate counterparts generated over the course of the workflow can add up to several gigabytes.

We evaluate our work using a custom montage application which handles a non-trivial amount of data. 50 mosaics are created in parallel on 50 workers. Each mosaic begins with a set of 91 distinct input FITS images, where each image is slightly over 2 megabytes in size. There are 12 tasks in the construction of each mosaic. Each task produces several intermediate data items, in some cases hundreds of files. The intermediate data produced must be read by one or more subsequent tasks in the sequence.

Each method evaluated was run several times in order to ensure any particular result was not a performance outlier. Statistics from three selected runs of each method are displayed in Figure 8. In part **a.** each bar displays the median execution time. The minimum and maximum times are indicated by the error bar. Both task grouping and the shared FS exhibit relatively consistent results, where the default TaskVine scheduler is largely unpredictable, sometimes competitive with Parsl HTEX and the shared filesystem, but in other cases severely impeded.

Execution Time No Groups	Local Files % No Groups	Execution Time Task Groups	Local Files % Task Groups
442s	72%	228s	91%
660s	66%	209s	92%
1380s	61%	192s	94%

TABLE I: Data Locality Achieved in Montage Runs

Part **b.** of Figure 8 shows the cumulative bytes transferred between workers. The shared FS runs are not pictured since there is no data transferred between workers. The graph shows that task grouping transfers significantly less data between workers than the ordinary scheduler, as well as avoiding any significant "plateau" in the rate of workflow progress.

This graph closely relates to part **c.**, where the number of tasks scheduled is graphed over time. Each run schedules over 500 tasks. The task grouping runs are consistent and without significant impediment. The "Group Events" line shows the number of group scheduling events which occurred compared to the total amount of tasks scheduled. Without grouping each task requires its own distinct scheduling event. With task group dispatching the application required less than 1/3 of the scheduling events and interactions between manager and worker.

The data locality achieved by TaskVine base and task grouping is displayed in Table I. This application involves approximately 117 thousand intermediate data files. The total amount of data used in the workflow amounts to 245.7GB. The general trend across all runs is consistent in showing that task grouping utilized over 20% more local data than the traditional scheduling method, resulting in significant performance benefits.

VII. Related Work

A. Workflow Systems

There are a number of workflow systems available to HPC users wishing to parallelize and distribute their code to a

cluster of computational resources. These workflow systems, such as Parsl [5], Dask [10], Ray [11], Kepler [12] and Pegasus [13] offer an API for users to describe their workflow in terms of tasks or functions with input and output dependencies. The workflow system interprets the user input and constructs a DAG to clarify the execution order and opportunities for parallelism. TaskVine also provides an API for workflow description, as well as the capability to act as a scheduler for some of the workflow systems described such as Parsl and Dask. The distinction of TaskVine when compared to other workflow systems is the management of data, and an emphasis on local storage usage which is coupled with TaskVine. Other workflow systems typically provide only a simple data staging interface to work with the shared filesystem. While data is what determines the structure of the DAG, other workflow systems do not concern themselves with how the data is handled. TaskVine manages the transfer of data from the submit location to remote workers and transfers between workers themselves, and is acutely aware of the location and characteristics of data assets within the cluster. This data-awareness is what enables scheduling based on data locality. The benefits of data aware scheduling are thoroughly explored in [4] and [6]. We discussed delay scheduling [6] at some length in our presentation of the scheduling challenge. Delay scheduling was shown to be highly effective in locality based scheduling when given tasks on-demand, such that no forward knowledge of tasks is known by the workflow system at all. The system operates with the assumption that the scheduler will make decisions found to be counterproductive given new information, and includes arrangements for canceling tasks in favor of replacing them with data-local tasks, and provisions not only node-local but rack-local locality awareness.

B. Group Scheduling

Grouping or ”clustering” tasks is a familiar notion for users of the workflow system Pegasus [13]. Pegasus allows extensive configuration regarding the clustering of tasks. Tasks may be clustered ”horizontally” by grouping tasks in the same level of the DAG, or tasks may be grouped by user annotation. It is important to note that the horizontal task clustering is grouping parallel jobs together instead of sequential jobs, and the goal of clustering in Pegasus is not exactly to benefit from data locality, but to submit less overall jobs to the cluster, since cluster batch schedulers may have a very high job startup overhead. In this way there is some similarity between the TaskVine implementation and Pegasus, since we are also focused on latency between tasks, and the cost of scheduling work to a remote resource, but our focus is on a different layer of the HPC software stack. While Pegasus schedules a workflow through a batch job provider such as HTCondor, TaskVine uses HTCondor to deploy its workers, and once deployed all task scheduling and communication occurs between the manager and worker processes. Another aspect which enables Pegasus to offer a rich set of job clustering features is the static compilation of the workflow DAG, i.e. that all tasks are known before runtime, and the user and software are free to design the DAG with certainties about its nature. TaskVine and its workflow system counterparts for which it may act as executor receive tasks at runtime, therefore unable to determine the exact structure of the DAG until all tasks have been revealed. While a precompiled DAG may easily reveal the ideal execution pattern, there are many benefits to a system which can react dynamically in an unpredictable distributed system. A workflow system in which tasks are declared at runtime may handle workflows with indeterminate execution patterns and overcome failures when resources are lost and data must be recreated.

C. Node-Local Storage

Storage technologies for use in HPC workflows are a fundamental component in the cluster environment. Parallel filesystems such as Lustre [14], Panasas [8], and Ceph [15] are typically installed in HPC facilities and used as a primary location for staging workflow data. Advances in distributed filesystem technologies focus on increasing total I/O operation throughput, since there are a number of I/O patterns in data-intensive applications which can reach performance limitations in the capacity of the filesystem to distribute data. There are a number of projects which support the use of node-local storage as an alternative or supplement to a parallel filesystem in order to overcome these limitations. Many of these technologies combine the local storage of nodes within the cluster into its own distributed filesystem such as BeeOND [1] and GekkoFS [2]. These and other works are considered ”burst buffers” [16] [17] [18] [19], which in effect move data closer to the execution site, creating a private shared filesystem for each instance. While using these technologies prodvides I/O benefits, the workflow system is still separate from the storage provider. Therefore the workflow system cannot determine exactly where the data is in the cluster. TaskVine is distinct from these technologies as it does not create a traditional FUSE-type shared filesystem across the workers which must be installed and mounted at the execution site. Rather each worker’s local disk is tracked and managed separately by the workflow system. The coupling of storage management to the workflow system allows TaskVine to be aware of exact data locations and to place tasks on the same worker where the data exists.

VIII. Conclusion

This work has presented an implementation of task group scheduling using the TaskVine and Parsl workflow systems. We have shown that a key issue in local-storage based workflows is the ability to achieve data locality in the face of scheduling complexity. Our evaluation found a significant improvement in locality-based scheduling effectiveness resulting in performance benefits which open new avenues for local-storage based workflows in the HPC ecosystem.

IX. Acknowledgment

This work was supported by National Science Foundation grant OAC-1931348

REFERENCES

[1] "BeeGFS," https://www.beegfs.io/c/, 2023, [Online; accessed 20-July-2023].

[2] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "Gekkofs - a temporary distributed file system for hpc applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 319–324.

[3] B. Sly-Delgado, T. S. Phung, C. Thomas, D. Simonetti, A. Hennessee, B. Tovar, and D. Thain, "Taskvine: Managing in-cluster storage for high-throughput data intensive workflows," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1978–1988. [Online]. Available: https://doi.org/10.1145/3624062.3624277

[4] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 301–316.

[5] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–36. [Online]. Available: https://doi.org/10.1145/3307681.3325400

[6] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 265–278. [Online]. Available: https://doi.org/10.1145/1755913.1755940

[7] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[8] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system." in *FAST*, vol. 8, 2008, pp. 1–17.

[9] J. C. Jacob, D. S. Katz, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams, "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," 2010. [Online]. Available: https://arxiv.org/abs/1005.4454

[10] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, K. Huff and J. Bergstra, Eds., 2015, pp. 130 – 136.

[11] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 561–577.

[12] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurrency and computation: Practice and experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[13] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[14] S. Cochrane, K. Kutzer, and L. McIntosh, "Solving the hpc i/o bottleneck: Sun™ lustre™ storage system," *Sun BluePrints™ Online, Sun Microsystems*, 2009.

[15] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. USA: USENIX Association, 2006, p. 307–320.

[16] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 807–818.

[17] O. Yildiz, A. C. Zhou, and S. Ibrahim, "Eley: On the effectiveness of burst buffers for big data processing in hpc systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 87–91.

[18] L. Pottier, R. F. da Silva, H. Casanova, and E. Deelman, "Modeling the performance of scientific workflow executions on hpc platforms with burst buffers," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 92–103.

[19] L. Cao, B. W. Settlemyer, and J. Bent, "To share or not to share: Comparing burst buffer architectures," in *Proceedings of the 25th High Performance Computing Symposium*, ser. HPC '17. San Diego, CA, USA: Society for Computer Simulation International, 2017.