# Minimizing Data Movement Using Distant Futures

Barry Sly-Delgado
University of Notre Dame
South Bend, IN, USA
bslydelg@nd.edu

Douglas Thain
University of Notre Dame
South Bend, IN, USA
dthain@nd.edu

## ABSTRACT

Scientific workflows execute a series of tasks where each task may consume data as an input and produce data as an output. Within these workflows, tasks often produce intermediate results that may serve as inputs to subsequent tasks within the workflow. These results can vary in size and may need to be transported to another worker node. Data movement can become the primary bottleneck for many scientific workflows thus minimizing the cost of data movement can provide a significant performance benefit for a given workflow. **Distant futures** enable transfers between worker nodes, eliminating the need for intermediate results to pass through a centralized manager for future tasks invocations. Additionally, asynchronous transfers enable increased concurrency by preventing the blocking of task invocations. This poster shows the performance benefit received from the implementation of distant futures within a workflow that produces numerous intermediate results.

**ACM Reference Format:**
Barry Sly-Delgado and Douglas Thain. 2023. Minimizing Data Movement Using Distant Futures. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

```python
import ndcctools.taskvine as vine

def gen_matrix(n):
    ...
def empty_matrix(n):
    ...
def matrix_multiply(a,b):
    result = empty_matrix(len(a))
    for i in range(len(a)):
        for j in range(len(b[0])):
            for k in range(len(b)):
                result[i][j] += a[i][k] * b[k][j]
    return result

opts = {"min-workers":5,"memory":8000,"disk":8000}
e = vine.Executor(name="my_app",batch_type='sge',opts=opts)
a = e.submit(matrix_multiply,gen_matrix(20),gen_matrix(20))
b = e.submit(matrix_multiply,gen_matrix(20),gen_matrix(20))
c = e.submit(matrix_multiply,a,b)
print(c.result())
```

**Figure 1: TaskVine Futures Example**
*This application utilizes Taskvine's futures paradigm. It creates two futures a and b that are then passed as arguments to future c. The ultimate result is printed by calling c.result()*

## 1 INTRODUCTION

A scientific workflow executes a series of tasks on a set of compute nodes, often towards the computation of a single result. Within these workflows, tasks may produce intermediate results which are outputs that are to be consumed by one or more tasks as an input. In certain paradigms, intermediate results are returned from the compute node to a centralized manager for it to be redistributed at a later time. This can be inefficient and can become more apparent with many intermediate results, large results, and limited bandwidth.

Futures are a common paradigm in which a task submission returns a reference to a result that may be computed at a later time [2]. The introduction of futures to scientific workflows can aid in the increase of concurrency as a result may not be needed immediately, allowing for other operations to occur without blocking. This poster presents **Distant futures** which expands on futures by distributing references to intermediate results to worker nodes on a compute cluster. Distant futures leverage the local storage of the compute cluster and retain intermediate results at the location of computation. Thus, subsequent tasks that consume the intermediate data can be scheduled to the location in which the data is present, removing the need to move the data. In the scenario where a task cannot be scheduled to the worker in which the data resides, intermediate data can be transferred between workers asynchronously exploiting the in-cluster bandwidth and removing an extra hop of movement that would have gone through the manager. This can occur when a worker is busy with other tasks

**Asynchronous transfers** enable concurrency within the distant futures paradigm by preventing workers from blocking before tasks invocations. That is, by transferring intermediate results asynchronously, tasks which depend on data or futures that can be resolved locally are not blocked from execution if a previous task needs to resolve a distant future by transferring an intermediate result. Thus, Having transfers be done asynchronously can be pertinent to the overall performance of a workflow and any blocking behavior can be very detrimental if the size of intermediate results are large.

**TaskVine**, a workflow executor for scientific applications, enables the creation of futures via its future executor. TaskVine's future executor is a subclass of Python's concurrent.futures executor. Similarly to Python, a future is created by submitting a function along with its arguments to the executor. The result of

*Conference'17, July 2017, Washington, DC, USA*
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

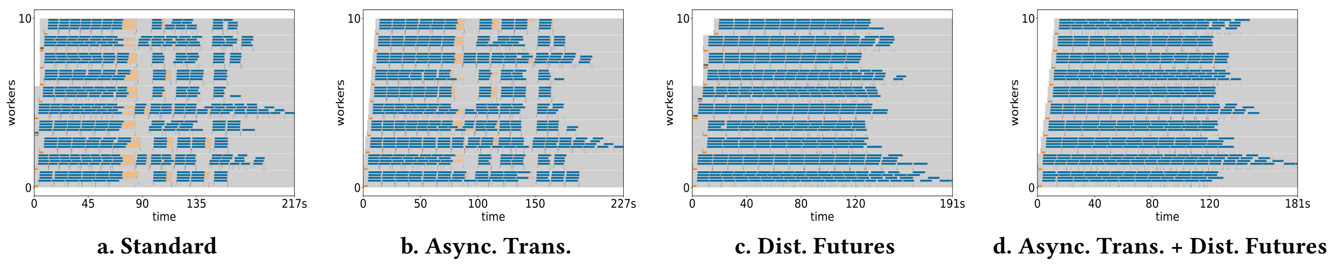

**Figure 2: Worker View of Matrix Example on different Configurations**

*Execution of the TaskVine future matrix example using different configurations. These figures show that removing the extra hop through the manager can significantly reduce the execution time of the overall application. Additionally, asynchronous transfers reduce the startup time for task execution. Blue bars represent the time in which a task is executing. More compact tasks result in better performance. Light orange bars represent time in which intermediate results are waiting to be retrieved from the manager. Less is better. Dark orange bars show the time a worker spends making input transfers and staging data. Dark grey bars are input transfers via the manager.*

the task can then be resolved by calling future.result(). TaskVine creates a corresponding **future task** when a future is created. This task serializes the function along with its arguments and sends them to a worker node for computation. A distant future can be created by passing a future as an argument to a future task. If an argument is a future, instead of waiting for the future result to be resolved, TaskVine replaces the future with a reference to the given future task's output. This enables the creation of distant futures. The TaskVine manager will receive cache updates via the workers once results are produced. Once all of a future task's dependencies, including the results of other future tasks if applicable, are available on the compute cluster, that task will be scheduled. TaskVine prioritizes data locality and will attempt to schedule a task where the necessary data dependencies are already present. Figure 1 shows an example application using TaskVine's future executor and the creation of a distant future.

## 2 RESULTS

To show the effectiveness of distant futures and asynchronous worker transfers, we ran an example workflow that will make the best use of these features. The workflow performs a series of matrix multiplications on randomly generated matrices of size 400 x 400. The series of multiplications forms a binary tree of operations resulting in a single matrix. Each matrix is roughly 1.4MB and each task requires 600MB environment that must be transported to each worker. The workflow has a total of 511 tasks that are executed by 10 workers submitted to a local compute cluster each with 4 cores.

This workflow is run in four configurations for each possibility that toggles distant futures and asynchronous worker transfers. While distant futures are disabled, intermediate matrix results are returned to the manager and are dispatched to future tasks. Conversely, when enabled, intermediate results remain on the worker or are transferred between workers when necessary. While asynchronous transfers are disabled the worker will stage necessary input files such as the environment and transfer intermediate matrices synchronously. When enabled, this process is asynchronous allowing the worker to receive new communications from the manager and start new tasks.

Figure 2 shows selected executions of the matrix workflow on each configuration from the workers' perspective. Figure 2a is the standard configuration without distant futures and asynchronous transfers. The absence of distant futures results in more separation between task executions as the manager has to receive intermediate matrices before dispatching them to new tasks this is also apparent in Figure 2b which has asynchronous transfers enabled but is also not using distant futures. Figures 2c and 2d introduce distant futures into the workflow which creates a noticeable drop in task separation as new tasks can either be scheduled to the intermediate matrices present on a worker or can be transferred between workers taking advantage of in-cluster bandwidth and eliminating the need for a hop though the manager. Unlike Figure 2d, Figure 2c does not enable asynchronous worker transfers. For this workflow, this primarily makes a difference during startup. As the manager is communicating with a worker, a worker may be busy staging or transferring data causing the manager to stall. This is removed in Figure 2d when asynchronous transfers are enabled.

The benefit received by using distant futures can heavily depend on a number of factors such as the bandwidth available within a cluster, the size of intermediate data, the availability of a shred file system, and local disk space. However, Eliminating transfers that would have otherwise needed an extra hop through the manager can generally improve performance and is more apparent with larger intermediate results.

## 3 RELATED WORK

Parsl [1] is a python library that enables the Creation of distributed applications. This library allows for the creation of futures by annotating python functions. Dask [3] is another library for distributed applications. Dask creates a task graph that can be optimized and given to an executor for execution. Dask.distributed allows for futures to be passed in as arguments to other tasks in which they are transferred between workers. [4] explores the use of distributed futures for fine-grained tasks in a manner that is fault-tolerant.

**Acknowledgement:**
This work was supported in part by NSF grant OCI-1931348.

## REFERENCES

[1] Yadu Babuji. 2019. Parsl: Pervasive Parallel Programming in Python *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3307681.3325400

[2] Henry C Baker Jr and Carl Hewitt. 1977. The incremental garbage collection of processes. *ACM SIGART Bulletin* 64 (1977), 55–59.

[3] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *SciPy*.

[4] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. 2021. Ownership: A Distributed Futures System for Fine-Grained Tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 671–686. https://www.usenix.org/conference/nsdi21/presentation/cheng