BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100001

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Workflow Critical Path: A data-oriented critical path metric for Holistic HPC Workflows

Daniel D. Nguyen [*], Karen L. Karavanic

Department of Computer Science, Portland State University, Portland, OR, United States of America

ARTICLE INFO

Keywords:
Critical path analysis
Holistic HPC Workflows
Parallel performance tools
Workflow critical path

ABSTRACT

Current trends in HPC, such as the push to exascale, convergence with Big Data, and growing complexity of HPC applications, have created gaps that traditional performance tools do not cover. One example is Holistic HPC Workflows — HPC workflows comprising multiple codes, paradigms, or platforms that are not developed using a workflow management system. To diagnose the performance of these applications, we define a new metric called Workflow Critical Path (WCP), a data-oriented metric for Holistic HPC Workflows. WCP constructs graphs that span across the workflow codes and platforms, using data states as vertices and data mutations as edges. Using cloud-based technologies, we implement a prototype called Crux, a distributed analysis tool for calculating and visualizing WCP. Our experiments with a workflow simulator on Amazon Web Services show Crux is scalable and capable of correctly calculating WCP for common Holistic HPC workflow patterns. We explore the use of WCP and discuss how Crux could be used in a production HPC environment.

1. Introduction

The term *workflow* is used throughout scientific computing with different contexts and meanings. For example, some scientific applications are developed with a workflow management system such as Pegasus [1] or Kepler [2], that schedules, runs, adapts, and summarizes a large number of lightweight tasks. Yet many computational science applications are implemented outside of any structured workflow management system. They comprise multiple steps, where each is a distinct library, script, or application with a specific functionality and design. For example, a science code might call an existing modeling code that is treated as a black box. These *Holistic HPC Workflows* are the focus of this work. Holistic HPC Workflows are an increasingly important paradigm with the potential for performance bottlenecks caused by movement and copying of large datasets, and inefficient interfaces between the separate components and applications. Today these workflows often include analysis and visualization of very large data sets, using methods developed for Big Data such as machine learning, analytics, and visualization. This growing complexity requires new ways of characterizing performance at the workflow level [3]. Workflow management systems (WMS) like Pegasus offer researchers a way to organize, execute, and analyze their scientific jobs. However, the performance analysis is tightly coupled to the WMS, thus these systems do not solve the problem of holistic performance analysis for workflows designed outside of such a system.

Analyzing the performance of Holistic HPC workflows presents a challenge for many existing performance tools, that are able to accurately and efficiently diagnose the performance of each individual component, but not to diagnose problems that span across them [4,5]. For instance, tools such as HPCToolkit [6] and TAU [7] use profiling and tracing techniques to detect performance bottlenecks in parallel applications. Some tools such as Darshan [8] and IPM [9] use I/O tracing to characterize I/O behavior of parallel applications. These tools were designed to analyze the performance of a single parallel code using the common approaches of message passing interface (MPI), multithreading (OpenMP), acceleration (CUDA), or a hybrid approach. However, diagnosing Holistic HPC Workflows requires integrated analysis across the separate components. A recent U.S. Department of Energy report on the future of scientific workflows called out this need for research “extending single-application performance validation tools to workflows of applications” [10].

One motivating example for our work is the Groningen Machine for Chemical Simulations (GROMACS) [11]. GROMACS is a scientific framework for simulating molecular dynamics of biochemical modules such as proteins, lipids, and nucleic acids. It models these molecular dynamics by solving Newtonian equations of motion for systems with hundreds to millions of particles. A common workflow pattern in GROMACS involves setting up a simulation environment, adding a solvent medium, generating an initial molecular model, calculating energy minimization, calculating initial equilibrium, and calculating actual molecular dynamics [12]. Each step can correspond to a single application using a shared file system, managed by a job scheduler like SLURM. Analyzing the workflow performance of GROMACS proved difficult; attempts included using a top-down approach by deconstructing

* Corresponding author.
E-mail addresses: ddn2@pdx.edu (D.D. Nguyen), karavan@pdx.edu (K.L. Karavanic).

https://doi.org/10.1016/j.tbench.2021.100001
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 23 December 2021

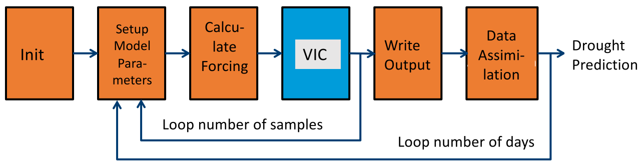


Fig. 1. Application layer of DroughtHPC workflow.

workflow into I/O, communication, and computation components and subsequently instrumenting the workflow applications to record these metrics [13].

A second motivating example is *DroughtHPC* [14]. This application, developed at Portland State University, predicts drought for a target geographical area. It utilizes the Variable Infiltration Capacity model (VIC) [15] to simulate meteorological samples over a given time period. A python script is used to perform data assimilation and call VIC in a loop (see Fig. 1). Every call to VIC inputs and outputs 25 files. The number of calls equals the number of samples needed multiplied by the number of days needed. Locating workflow bottlenecks for DroughtHPC, particularly due to dataflow and the control flow of the entire workflow, was challenging [16]. To investigate performance bottlenecks, researchers manually ran a variety of measurement tools to focus attention to the key bottlenecks. The overhead of calls to the VIC hydrologic model from within a python loop and significant file creation, reads, and writes, represented main performance bottlenecks. The DroughtHPC study shows a need for one performance tool that can detect common dataflow patterns and diagnose runtime bottlenecks across different phases in a scientific workflow.

Holistic HPC Workflow Diagnosis is also an important aspect of the procurement process for major new systems at large science labs. Describing the workload accurately is essential to matching the capabilities of the future systems to the needs of the lab. Fig. 2 shows an example of the phases associated with common large-scale scientific simulation workflows, with data retention timescales divided into temporary, campaign, and forever. The temporary timescale describes application data that is typically discarded at completion of a phase or run. The campaign timescale includes data used throughout the execution or set of executions of the entire scientific workflow. The archive timescale describes data stored for longer archival purposes. This type of diagram is modeled after those developed by The Alliance for Application Performance at Extreme Scale (APEX) [17].

In this paper we present our initial work to address this need. *Workflow Critical Path* (WCP) is a data-oriented critical path metric for Holistic HPC Workflows. Building on earlier work on Critical Path Analysis for individual MPI applications [18–20] we have developed a technique for determining the critical path across an entire Holistic HPC Workflow. This has the potential to help researchers better understand data movement patterns and potential bottlenecks occurring across the complex memory hierarchy and storage systems in a large-scale HPC cluster. Our approach is designed to focus developers' optimization efforts, avoiding the need to separately analyze each participating application and manually determine where to focus. It also allows the detection of performance bottlenecks related to moving from one stage of the workflow to the next, for example, copying and transforming simulation output data for analysis with a visualization tool.

The key contributions of this paper are:

1. **We define Workflow Critical Path (WCP), a novel performance metric for Holistic HPC Workflows.** WCP describes the critical path for an entire HPC workflow by defining a program activity graph (PAG) where vertices represent *data state* and edges represent *data mutations*.
2. **We present Crux,** a distributed, runtime tool that calculates WCP. Crux follows a service-oriented architecture and deploys on a target number of nodes in an HPC cluster. Crux provides an API for building workflow PAGs and computing WCP. It also provides a user interface (UI) for visualizing WCP data. Our Crux prototype can be deployed in the Cloud using Amazon Web Services or locally using Docker.
3. **We developed a configurable HPC workflow simulator framework, and used it for a detailed capability, scaling and performance study.** The configurable workflow simulators allow users to simulate representative workloads.

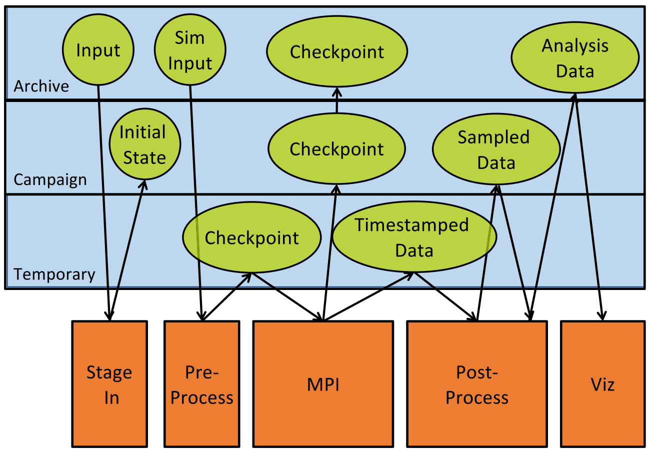


Fig. 2. APEX style workflow diagram. The brown boxes along the bottom show the major steps in the workflow, and the blue rectangles show the levels of memory and storage for the data at each step.

This represents a first step towards *Holistic HPC Workflow* performance diagnosis [21].

2. Related work

There are a number of workflow management systems in use today, for example Kepler and Pegasus. Such systems generally include performance monitoring infrastructure, however the applications must be specifically implemented for the specific workflow management system. Crux on the other hand targets Holistic HPC Workflows, that comprise separately developed components to solve a single problem.

Annotation-based (also referred to as application-instrumented) distributed monitoring schemes developed for commercial server environments rely on applications to explicitly tag every record with a global identifier that links these message records back to the originating request. These systems tend to be very accurate but potentially slow, as all system components must be instrumented. One example is Dapper, developed by Google [22]. It has been used for a large, production distributed systems tracing framework. In a Dapper trace tree, the tree nodes are basic units of work which are referred to as spans. The edges indicate a causal relationship between a span and its parent span. A span is a simple log of timestamped records which encode the span's start and end time, any RPC timing data, and zero or more application-specific annotations. A span can contain information from multiple hosts and in fact every RPC span contains annotations from both the client and server processes. Crux follows an annotation-based approach, however it greatly reduces the overhead by only creating nodes for data operations.

Facebook's end-to-end performance tracing infrastructure, Canopy, is another example of a large-scale, runtime performance tool that can record and process over 1 billion traces per day [23]. Canopy has several features similar to WCP. Canopy models trace data as DAGs with nodes representing events in time, with events defined more broadly and at a lower level than WCP. Canopy authors noted how infeasible it was to expose traces at that particular level of granularity since end-users, i.e. Facebook engineering teams, would not understand the mappings to higher-level concepts. To address this, Canopy constructs a modeled trace of events, which are higher-level representations of lower-level performance data. WCP focuses on the end-to-end movement and transformation of data across an entire HPC workflow instead of performance within any particular workflow component. For example, WCP is not intended to diagnose one MPI application. Like WCP, Canopy also derives the critical path of its trace data and visualizes the critical path to the end user.

Research into CPA for parallel programs started in the 1980s with work such as Yang & Miller [20]. Their approach involved constructing a directed, weighted graph, called program activity graph (PAG), whose vertices represent events (e.g. send/receive and process creation/termination events) in a program and whose edges represent the duration of the event. They were able to return the longest path on a scale of tens of thousands of nodes. Critical path analysis evolved in the 1990s with techniques such as using piggybacking critical path data on MPI messages to compute the critical path profile during runtime [19] Hollingsworth demonstrated that using this technique, most programs can tolerate a 5%–10% level instrumentation overhead without suffering significant change of the critical path length. Critical path for individual MPI applications has continued to be improved and scaled up with increasing numbers of MPI ranks [24–27]. Overall, critical path analysis is useful in identifying the cause of a program's total execution time, diagnosing bottlenecks to application scalability, and predicting overall performance [24].

Our work targets holistic HPC workflows, thus requiring a novel approach to monitor separate components and merge their graphs. In preliminary work towards this same goal, Herold & Williams introduced a top-down performance analysis approach to monitor workflow applications [18]. They implemented a tracing infrastructure that interfaces with the resource manager to provide summarized performance metrics for workflow, jobs, and job steps. In contrast, we focus on defining a specific metric, WCP, and a runtime approach to its calculation and visualization.

3. Workflow critical path (WCP)

Workflow Critical Path is calculated by constructing a program activity graph (PAG) spanning all components of a holistic workflow, representing *data state* as vertices and *data mutations* as edges. The result is a PAG that can be analyzed for *data state patterns* through an entire HPC workflow (Fig. 3).

A *data state* comprises:

- Size — the size of the data, for example 10 MB;
- Time — the creation timestamp;
- Origin — the original application that produced the state;
- Location is the current storage location, for example "node1 disk1" ; and
- Label - a meaningful descriptor for the data state (e.g. file.csv, byte_stream).

An edge represents a *data mutation*, an operation that changes a data state. An edge comprises:

- cost — the elapsed time between two connected data states; and
- mutation — the operation performed on a data state resulting in a state change.

The resulting graph allows WCP to describe a data set evolving over time. This focus on data generalizes time: unlike profilers, the "cost" captured in each edge includes all computation and I/O activity between each two data states. Reducing I/O activity cost thus potentially changes or improves the critical path. Reducing computation time also potentially changes or improves the critical path, just as in computation-oriented approaches.

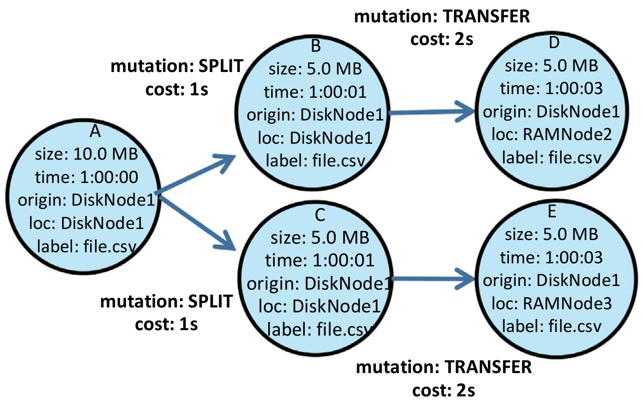


Fig. 3. Trivial example of a data state undergoing different mutations. Vertex A represents a 10.0MB file called file.csv on a disk belonging to Node1. Vertex A undergoes a SPLIT mutation that divides file.csv into file1.csv and file2.csv. The result is two new data states, B and C. Vertices B and C undergo a TRANSFER mutation that transfers file1.csv and file2.csv from disk on Node 1 to memory on nodes 2 and 3 respectively.

3.1. Critical path algorithm

The critical path represents the *longest* path through the graph of data state mutations based on execution time. Thus, critical path algorithms are typically shortest path algorithms modified to find the path with the longest execution time [28]. A well-known algorithm that solves the single source shortest path (SSSP) problem is Dijkstra's algorithm which has a worst-case performance of O(|E|+|V| log |V|) where |V| is the number of vertices and |E| is the number of edges. Delta-stepping [29] is a distributed variant that divides Dijkstra's algorithm into phases that can be executed in parallel on distributed memory architectures for an average-case time of $O(\log^3 n/\log\log n)$. For our WCP prototype we use a version of Delta-stepping implemented for shared memory architectures described by Kranjčević et al. [30]. The input of the Δ-stepping algorithm is a graph given by its vertices V, edges E, and the cost function c, a source node s, and an optional parameter $\Delta > 0$ used to divide all the outgoing edges of each vertex into two categories, called light and heavy edges, based on whether the cost of that edge is smaller or larger than Δ. The Kranjčević et al. implementation of Delta-stepping performs $O(|V|^{1+\frac{1}{d}})$ operations total for graphs representing d-dimensional square lattices. Their testing showed an average parallel efficiency of at least 50% over Dijkstra. The pseudocode for this algorithm is shown in Fig. 4.

Traditional PAGs where nodes represent computations and edges represent computational activities typically store the duration of computational activities as the edge weight and employ a longest path algorithm to return the critical path [28]. Since WCP represents data state as vertices and data mutations as edges, we use the elapsed time between data states as the edge weight.

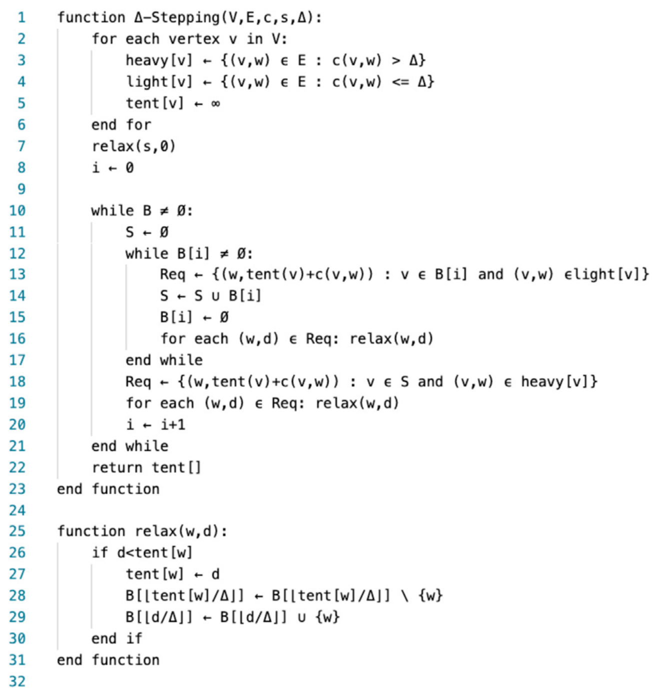
We store a weight property, called cost, for edge E such that the cost equals the inverse of elapsed time, i.e. difference between timestamped values of vertex A and vertex B.

$$cost_E = \frac{1}{time_B - time_A}$$

The inverse elapsed time means that edges between data state vertices with large time differences will receive a small cost value and vertices with small time differences will receive a large cost value.

4. The crux prototype

To enable further study of WCP, we implemented a prototype, Crux, along with the tooling needed to build and deploy. Crux comprises the following modules:



```
function Δ-Stepping(V,E,c,s,Δ):
    for each vertex v in V:
        heavy[v] ← {(v,w) ∈ E : c(v,w) > Δ}
        light[v] ← {(v,w) ∈ E : c(v,w) <= Δ}
        tent[v] ← ∞
    end for
    relax(s,0)
    i ← 0

    while B ≠ ∅:
        S ← ∅
        while B[i] ≠ ∅:
            Req ← {(w,tent(v)+c(v,w)) : v ∈ B[i] and (v,w) ∈light[v]}
            S ← S ∪ B[i]
            B[i] ← ∅
            for each (w,d) ∈ Req: relax(w,d)
        end while
        Req ← {(w,tent(v)+c(v,w)) : v ∈ S and (v,w) ∈ heavy[v]}
        for each (w,d) ∈ Req: relax(w,d)
        i ← i+1
    end while
    return tent[]
end function

function relax(w,d):
    if d<tent[w]
        tent[w] ← d
        B[⌊tent[w]/Δ⌋] ← B[⌊tent[w]/Δ⌋] \ {w}
        B[⌊d/Δ⌋] ← B[⌊d/Δ⌋] ∪ {w}
    end if
end function
```

Fig. 4. Pseudocode for the Delta-Stepping Algorithm.

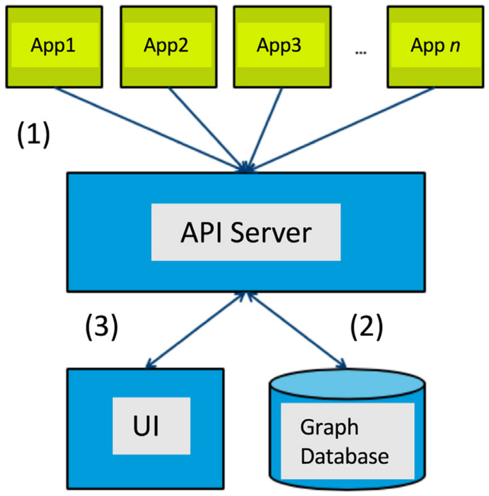


Fig. 5. Crux API interaction with HPC clients. (1) HPC applications (green) make API calls to Crux (blue) through HTTP requests to the API server. (2) The API server communicates to the graph database using a compatible protocol. (3) The UI is a standalone application making HTTP requests to the API server for CRUD (create, read, update, and delete) actions on the graph database.

Crux API: An HTTP, application programming interface (API) that exposes representational state transfer (REST) endpoints to workflow HPC applications. The Crux API server implements routines to build workflow PAGs; interfaces with the Crux Database; performs data integrity checks; and manages Crux's performance metadata;

Crux Database: A database that stores a workflow PAG and executes Crux's critical path algorithm for finding the WCP;

Crux UI: A user interface (UI) to visualize workflow PAGs and WCP.

In the remainder of this section we describe each of these modules, Crux deployment, and examples of Crux.

4.1. Crux API

The Crux API is an HTTP API that follows a representational state transfer (REST) architecture, chosen for benefits such as scalability and portability. The Crux API must follow several constraints. First, it must define stateful objects as API resources for clients to access. The API resources should map to Crux's data state schema. For example, a client should be allowed to query a specific data state vertex in the database by sending an HTTP GET request to an API. Second, the Crux API must be a manager of the Crux database. It must implement logic that tells the database how to perform simple CRUD actions such as creating a vertex or updating an edge, or more complicated actions like submitting queries needed to calculate the critical path from two data state vertices in the PAG. Third, it must enforce the Crux data state schema so that clients cannot send malformed requests. Fourth, the API must provide support to the Crux UI for any backend requests and must provide common application features such as user login and access token management (See Fig. 5 and Table 1).

We identified the following properties as most important when comparing different backend tools and languages for the Crux API prototype: rapid development, high performance and asynchrony. Thus we chose to implement the Crux API using Python's Asynchronous Server Gateway Interface (ASGI), a core library used by a popular Python backend framework, Django. We chose FastAPI [31], a framework built around Starlette, which is a lightweight ASGI framework. FastAPI is a fast Python framework that integrates with standards for OpenAPI and the JSON schema.

REST-based web services are typically organized into resources, logical objects we want to expose to the user. The Crux API includes two resources: states and mutations. A resource is identified by a Uniform Resource Identifier (URI). Clients can access that resource by sending an HTTP request method to that URI. RESTful web APIs typically deploy URIs following the pattern `scheme://host:port/version/resource`. Parameters can also be used in URIs. For example, an API might have a path parameter `/users/{ID}` which lets the client specify a certain user with a specific ID. Parameters can also exist in the form of query parameters which lets a client sort or filter on a particular resource. For example, `/states?location=disk01` returns all data states located on disk01. For Crux, we use a combination of path and query parameters for clients to access resources.

Our Crux prototype includes six types of data mutations, based on common data operations observed in scientific applications:

TRANSFER — Transfer of data between one physical location to another (e.g. staging in data from storage to compute node)

CONVERT — Conversion of data format or schema (e.g. JSON to CSV)

APPEND — Appending data to existing data (e.g. adding timestamps to data points in a file)

SPLIT — Splitting of data into multiple locations (e.g. mpi_scatter())

MERGE — Merging data from different sources (e.g. mpi_gather())

DELETE — Permanent deletion of data

4.2. Crux database

The Crux Database is the backend storage for the Crux API. The database must support concurrent control to manage write operations from multiple API instances, and scaling to accommodate collected PAG data, representing vertices, i.e. data state, and edges, i.e. data mutations, of an entire HPC workflow. For an example workflow of 5 applications, each generating 100 data states and performing 100 data mutations, Crux's database must hold 50,000 entries.

For the Crux prototype, we wanted a solution that was well documented, showed strong concurrency use cases, and came with graph

Table 1
The Crux API.

HTTP Method	Path State Info	Description
GET	/states	Returns a list of data states
GET	/states/{ID}	Returns a data state with matching ID
POST	/states	Creates a new data state
GET	/mutations	Returns a list of data mutations
POST	/mutations/transfer start state, end state	Creates a TRANSFER data mutation between a start data state vertex and an end data state vertex.
POST	/mutations/convert start state, end state	Creates a CONVERT data mutation between a start data state vertex and an end data state vertex.
POST	/mutations/split start state, end states	Creates a SPLIT data mutation between a start data state vertex and ending at all end data state vertices
POST	/mutations/merge start states, end states	Creates a MERGE data mutation between all start data state vertices and ending at an end data state vertex.
POST	/mutations/append start state, end state	Creates an APPEND data mutation between a start data state vertex and an end data state vertex
POST	/mutations/delete start state, end state	Creates a DELETE data mutation between a start data state vertex and an end data state vertex
POST	/wcp start state: id, end state:id	Returns a list of data state vertices representing the workflow critical path between a start data state vertex and an end data state vertex

algorithm support optimized for that database. To this end, we chose a graph database, Neo4j [32], with a large ecosystem of tools and support. Neo4j uses a query language called Cypher and uses a convention of referring to vertices as *nodes* and edges as *relationships*. Cypher can be used to describe patterns of nodes and relationships and filter those patterns based on labels and properties. For example, the following Cypher query returns all data state nodes with matching property values:

```
MATCH (n)
WHERE n.size = {size} and
            n.location = {location} and
            n.time = {time} and
            n.origin = {origin}
RETURN n
```

To integrate Neo4j into Crux, we use a containerized version. We developed a custom Python library to express Crux data state, and data mutation schemas as proper Cypher queries to create nodes and relationships. We use a Python Neo4j client to execute write transactions between Crux API server and Neo4j. We calculate WCP using Neo4j's `algo.shortestPath.deltaStepping()` routine which implements delta-stepping for shared memory architectures described by Kranjčević et al. The Cypher query:

```
MATCH (start)
WHERE id(start) = {start_id}
CALL algo.shortestPath.deltaStepping.stream
     (start, "cost", 3.0)
YIELD nodeId, distance
RETURN algo.getNodeById(nodeId)
     AS destination, distance
ORDER BY distance
```

4.3. Crux UI

The Crux UI is a user interface to visualize critical path data in the Crux Database. This includes visualizing program activity graphs, critical paths, and various metadata like workflow runtime. In addition, the Crux UI provides features such as user authentication and profiles. The Crux UI runs as a standalone application and communicates to the Crux database via the API server. For users to access the Crux UI, the UI application must be properly exposed so authenticated end users can reach it from their location. For example, if end users are outside of the HPC cluster environment, the Crux UI can sit behind a public load balancer which routes public traffic to the UI instance.

We implemented the Crux UI for our prototype with the Neo4j Browser. The Neo4j Browser is a general-purpose UI that lets users query, visualize, administrate and monitor a Neo4j database. With this simpler approach, users can view a workflow PAG being constructed during runtime and submit Cypher queries against the graph database.

To visualize WCP in the Neo4j browser, we use the `algo.shortestPath.stream()` routine in the following Cypher query:

```
MATCH (start), (end)
WHERE id(start) = {start_id} and
     id{end} = {end_id}
CALL algo.shortestPath.stream (start, end,
     "cost")
YIELD nodeId, cost
RETURN algo.asNode(nodeId), cost
```

We needed to make one adjustment to the default behavior to ensure correctness for MERGE data mutations. A MERGE data mutation signifies the combining of two or more data states, such as combining of data from files to create a new file. When this occurs in Crux, a new data state vertex gets created and edges from each of the pre-merge data state vertices get added. At this point, each edge receives an elapsed time calculated from the parent vertex's timestamp and the timestamp of the new data state vertex. However, the critical path should be the path that includes the pre-merged date state vertex or vertices with the *smallest* elapsed time to the new data state vertex For Neo4j's shortest path algorithm to correctly return this path, we assign a large integer value as the cost for the other non-critical paths (see Fig. 6).

We were able to accomplish most needed functionality for Crux with Neo4j, however, the browser falls short of our particular needs. A production version of Crux would require a different approach for the Crux UI.

Fig. 9 shows a diagram of Crux installed in an HPC cluster. A basic installation of Crux requires the following:

- Minimum of 3 nodes located in the HPC cluster. These allocated nodes shall be on the same network as other compute nodes and accessible via HTTP.
- Crux UI installed on 1 node behind a load balancer or reverse proxy. This allows end users outside the HPC cluster network to reach Crux. The UI shall target HTTP requests to the Crux API server via another load balancer.
- At least one instance of the Crux API server installed on at least 1 node. Depending on workflow size, it may be appropriate to install multiple instances over multiple nodes. We expect private load balancer(s) to distribute API calls from client HPC applications efficiently to an API instance.

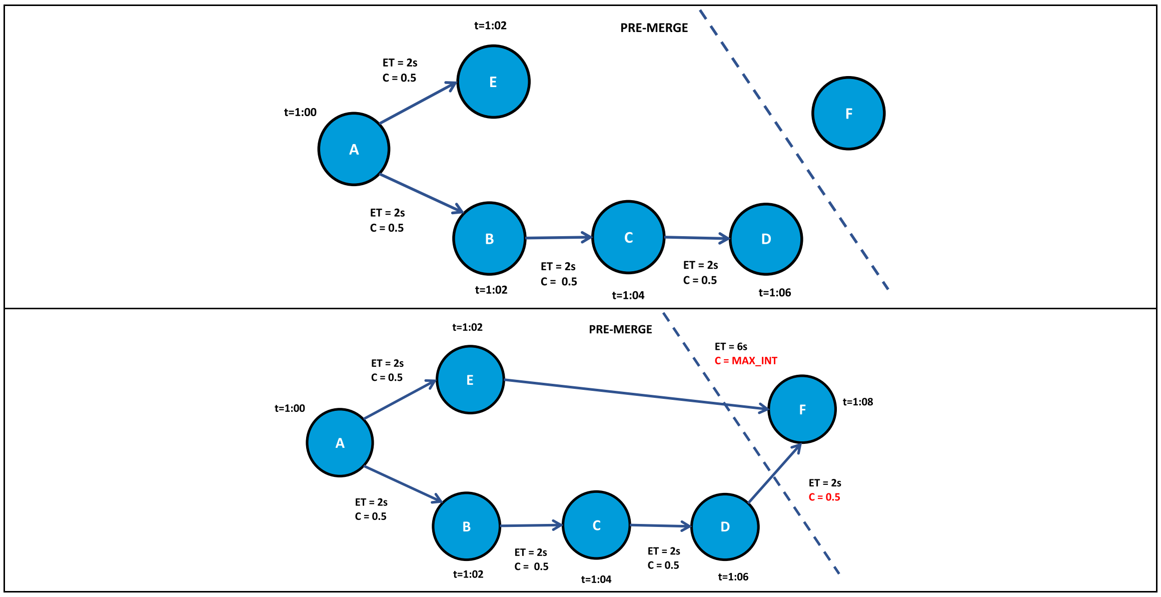


Fig. 6. Handling MERGE data mutations to use with Neo4j's shortest path algorithm. Top image shows a PAG with timestamped values for data state vertices A, B, C, D, and E along with data mutation edges with elapsed time (ET) and cost (inverse ET) shown. The intention is to merge vertex D and E. Vertex D took 6s to create from A whereas E took only 2s to create. Vertex F represents the new data state vertex from merging E and D. Bottom image shows Vertex F created at t=1:08 resulting in an elapsed time of 2s between D and F and 6s between E and F. Since the critical path must be ABCDF, we assign edge EF a high integer value in order for Neo4j's shortest path algorithm to return ABCDF as the critical path between A and F.

- Crux database installed on 1 node.

The Crux API is designed as a stateless server. It does not track or store data from clients. Clients of Crux must make appropriate API calls to Crux. This means that client HPC applications must support the same protocol (e.g. HTTP) to communicate with Crux. Furthermore, clients must know how to create data state information defined by Crux's schema. The following pseudocode shows an application loads in a data file `input.txt`. In order to model this in Crux, a total of 3 Crux API calls are needed.

4.4. Example

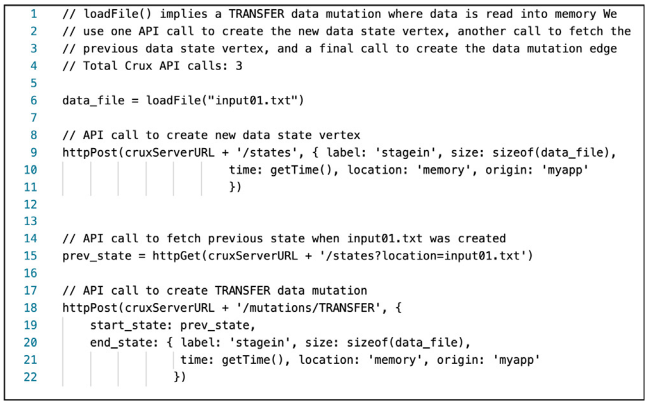
To illustrate WCP and Crux in practice, we instrumented two applications written in Python and C that perform similar I/O operations. Both programs stage in data, perform computation on the data, and write the results out to a new file. We inserted a total of 6 Crux API calls in each program. In the Python application, we included our custom Python library called Crux to access utility functions that help create and manage Crux data states. In C we also include utility functions that wrap around the `libcurl` library to help execute HTTP requests (see Figs. 7–9).

5. Crux workflow simulator

In order to effectively test WCP, we developed a workflow simulator for Crux, with a system of distributed applications to simulate representative scientific workflows. The simulator system serves as a lightweight, local testbed to examine Crux's performance (see Fig. 10).

We designed five representative workflows, each of which exhibits a characteristic element or pattern we have observed in HPC workflows, motivated in particular by the APEX report and DroughtHPC. The 5 workflows are:

1. Generic. These jobs include staging in data, preprocessing data, MPI, postprocessing data, and visualizing data. We consider this the simplest of workflows in that there is only one data source and the critical path will depend on the MPI rank that takes the longest. Simulators used: 'Stagein', 'Preprocess', 'MPI', 'Postprocess', and 'Viz'. TOTAL_MPI_RANKS = 4 (see Fig. 11).



```
// loadFile() implies a TRANSFER data mutation where data is read into memory We
// use one API call to create the new data state vertex, another call to fetch the
// previous data state vertex, and a final call to create the data mutation edge
// Total Crux API calls: 3

data_file = loadFile("input01.txt")

// API call to create new data state vertex
httpPost(cruxServerURL + '/states', { label: 'stagein', size: sizeof(data_file),
                        time: getTime(), location: 'memory', origin: 'myapp'
                        })


// API call to fetch previous state when input01.txt was created
prev_state = httpGet(cruxServerURL + '/states?location=input01.txt')

// API call to create TRANSFER data mutation
httpPost(cruxServerURL + '/mutations/TRANSFER', {
    start_state: prev_state,
    end_state: { label: 'stagein', size: sizeof(data_file),
                time: getTime(), location: 'memory', origin: 'myapp'
               })
```

Fig. 7. Crux API calls for capturing a load from input.txt.

2. *Data Splits*. MPI-based workflow with data splitting across a number of physical nodes. These jobs include staging in data, preprocessing data, MPI, postprocessing data, and visualizing data. Simulators used: 'Stagein', 'Preprocess', 'MPI', 'Postprocess', and 'Viz'.
3. *Checkpoint*. A workflow that includes more than one run of a parallel codebase with a checkpoint file created in between runs. The time duration to transfer the file is configurable. We simulate checkpoint files being written to storage between runs of parallel tasks representing the scientific simulation. Simulators: 'Stagein', 'Preprocess', 'MPI', 'Checkpointout', 'Checkpointin', 'MPI2', 'Postprocess2', 'Viz'.
4. *Multiple Sources*. Simulates a workflow that involves loading more than one source of data. The loading occurs between workflow jobs. Simulators: 'Stagein', 'Preprocess', 'MPI', 'Postprocess', 'Load', 'MPI2', 'Postprocess', 'Viz'.
5. *Create Delete*. Simulates a workflow that involves creating temporary files and deleting them between runs of a scientific simulation. Simulators: 'Stagein', 'Preprocess', 'MPI', 'Filecreate', 'Postprocess', 'MPI2', 'Postprocess', 'Viz'.

```
int main(int argc, char *argv[])
{
   char *startVertex, *inputVertex, *endVertex,
        *dataVertex;
   char *relationshipData;
   char *url, *data;

   // Init Crux and curl library
   curl_global_init(CURL_GLOBAL_ALL);
   startVertex = connectCrux();

   data = readFile("input.txt");

   // API calls to create new data state for
   // loading an input file into memory and to
   // create new data mutation TRANSFER to
   // represent loading a file
   inputVertex = newDataState("input.txt", 0);
   post(newURL (HOST, "/nodes/myapp"), inputVertex);
   post(newURL(HOST, "/relationships/transfer"),
   newRelationship(startVertex, inputVertex));

   sampleComputation(data);

   // API call to create new data state shows
   // data converted after computation in memory
   // API call to create new data mutation CONVERT
   dataVertex = newDataState("", strlen(data));
   post(newURL(HOST, "/nodes/myapp"), dataVertex);
   post(newURL(HOST, "relationship/convert"),
        newRelationship(inputVertex, dataVertex));

   writeFile("output.txt", data);

   // API call to create new data state for writing
   // to the new file, output.txt
   // API call to create new data mutation TRANSFER
   // to represent creating a file
   endVertex = newDataState("output.txt", 0);
   post(newURL(HOST, "/nodes/myapp"), endVertex);
   post(newURL(HOST, "relationships/transfer"),
   newRelationship(dataVertex, endVertex));
   curl_global_cleanup();
   return 0;
}
```

Fig. 8. Example of a simple C code with the inserted Crux API calls shown in boldface.

To make the simulators extensible, we create a common configuration file from which each simulator loads. This file parameterizes values such as maximum wait time between jobs or input dataset size. Each simulator starts an HTTP server and implements run_simulation()which takes a list of previous data states and returns a list of new states once all simulated jobs have completed. The pseudocode below highlights the basic logic in each simulator (we use the terms "nodes" and "relationships" to refer to vertices and edges respectively in order to follow Neo4j's naming convention).

```
// Input:   Previous data state vertex, prev_state
//             URL string to Crux API, url
// Output:     New data state vertex, new_state
run_simulator(prev_state, url)

// Simulate new preprocess data state
new_state = simulateDataState('preprocess')

// Call Crux API to create new data state  vertex
postRequest(url + '/nodes/preprocess', new_state)

// Call Crux API to create new data mutation edge,
// APPEND,
// between new data state and prev data state
postRequest(url + '/relationships/append',
    prev_state, new_state)
return new_state
```

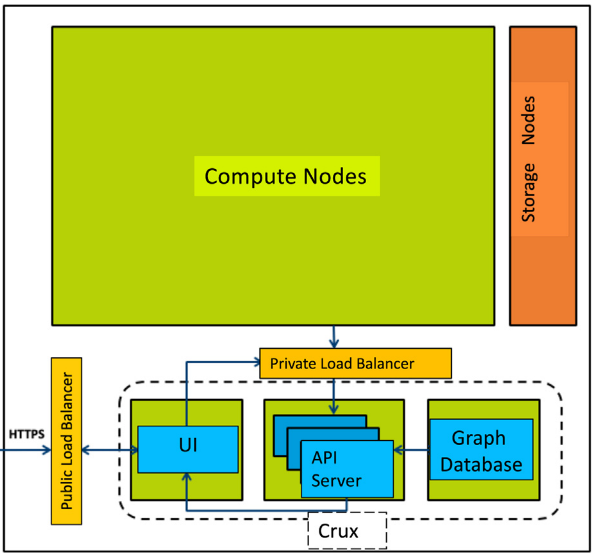


Fig. 9. Deployment of Crux in an HPC cluster. Crux components (blue) are deployed on 3 dedicated nodes. Two load balancers (yellow) are used to route traffic: a public load balancer which securely handles HTTPS requests from a user outside the HPC cluster network, and a private load balancer which routes traffic from compute nodes to Crux API server instance (in this case 3 running instances). The public load balancer can also be a reverse proxy. Storage nodes (orange) are displayed for reference.

In order to orchestrate simulator applications at runtime, we design a controller application called simulator manager. Simulator manager knows when to schedule each simulator's main routine. It also facilitates the passing of data between applications and performs health checks on each before starting. Simulator applications therefore only need to communicate with the simulator manager and not each other. The simulator manager's main routine receives an ordered list of URLs to each simulator. It initiates a null data state and enters a loop to call the first simulator with the null data state. The return value is a new data state which gets assigned to the previous state variable. The loop is then continued with the second simulator being called and so on. The pseudocode below outlines the basic algorithm.

```
// Input: ordered list of urls to each simulator, simulator_urls
// Output: void
run(simulator_urls)
    prev_state = null
    for url in simulator_urls
        prev_state = startSimulator(url, prev_state)
    return void
```

We use Docker to package Crux components as container images. Containers are isolated environments by OS-level virtualization. A container shares a host's kernel with other containers, but each container will only the see contents assigned to it. Docker is a set of tools for building and deploying containers. We choose to implement Crux with Docker for a variety of reasons. First, using containers for development offers benefits such as isolation, reproducibility, portability, and version control. Second, container images are lightweight compared to most virtual machine images. This is important when deploying Crux with workflow simulators since all Crux components and simulators run as individual containers (the largest being the API container at ~900MB and the smallest being a simulator container at ~115MB). Third, containers make it easy to deploy to cloud environments, which we leverage for testing purposes. (See Fig. 10).

Workflow Simulator components are implemented as standalone applications:

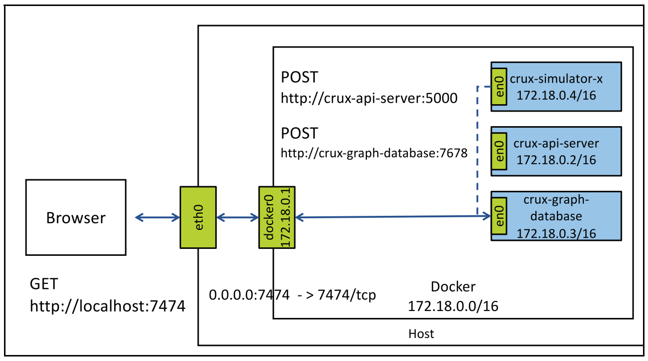


Fig. 10. Example of local deployment of Crux with simulators using Docker. Crux components and workflow simulators (blue) running locally as Docker containers. Network interfaces (green) shown to illustrate how containers run on separate virtual network than the host.

- The Simulator Manager is a controller and communicates with all simulator applications via HTTP. It knows when to launch a certain simulator app and send data between apps when needed.
- Simulator apps are configured to represent common HPC workflow jobs such as pre-processing data, running an MPI job, or performing post analysis. Each simulator app makes API calls to the Crux API server.
- The Crux API server, implemented in Python, sends Cypher queries via the Bolt protocol at bolt://graphdatabase.crux:7687.
- The web browser opens the Neo4j Browser at http://localhost:7474/browser/.

6. Evaluation

We conducted a series of experiments to demonstrate Crux capabilities and performance. For these tests we used Docker 19.03.5, Docker-compose version 1.24.1, Terraform v0.12.5, Python 3.7.6, Neo4j 3.5, Neo4j Graph 3.5.4.0, and AWS ECS Agent 1.32.0. The AWS EC2 Image used was Amazon Linux AMI 2018.03.y x86_64 ECS HVM GP2 t2.medium (2 vCPU, 4 GB RAM), and the local system was macOS Mojave (10.14.6) running on a 2.9 GHz Intel Core i9 (12 core) with 32 GB 2400 MHz DDR4.

6.1. Capability and WCP correctness

To explore WCP correctness, we configured the Crux workflow simulator to generate and execute the five characteristic workflows detailed in Section 6. The workflow simulator is deployed locally and makes API calls to a Crux instance on AWS. We kept runtimes short with modest total vertices in order to better provide screenshots of the entire PAG and workflow critical path.

For each of the 5 workflows, we configured a Crux simulator, then deployed it locally using a tool called Docker-Compose which launches all simulator components as containers. The simulator containers make API calls to a remote deployment of Crux on AWS. The workflow simulation is complete when we receive a JSON string from the Crux API which contains the workflow critical path. At this point, we collected screenshots of the visualized workflow PAG and workflow critical path via Crux's Neo4j Browser. Results are shown in Table 2 and Fig. 11. WCP was correctly computed in all cases. While the five cases do not cover every possible workflow pattern, we believe they cover key workflow patterns and thus are representative. The next step will be to explore WCP with full applications.

Table 2
Results of 5 simulator studies.

Workflow	*WCP Correct?*	*Cost*	*Elapsed time* (s)
Generic	Yes	9.492	6.584
Data Splits	Yes	22.159	57.590
Checkpoint	Yes	74.854	7.616
Multiple Sources	Yes	67.242	6.5413
Create Delete	Yes	74.868	7.2561

6.2. Performance and scalability

To characterize the scalability of the Crux prototype we measured the time required to create data state vertices on local and remote (AWS based) deployments of Crux (Fig. 12). Next, we scaled out the instances of Crux's API server to 1, 2, and 3 instances, and performed the same time measurement (Fig. 13).

6.3. Crux overhead

The main overhead of Crux will scale with the number of instrumented API calls required to create a full program activity graph. To approximate the number of Crux API calls needed for a smaller scale HPC application we used the DroughtHPC example. We estimated the number of Crux data states that would have to be created at 300k for VIC and 12k for the python code, for a total of 312k Crux API calls. A full-scale deployment will be required to accurately assess the overhead.

Dedicating nodes to deploy Crux in an HPC cluster implies taking nodes away that could otherwise be used as compute resources. However, we demonstrate Crux's ability to run as containers on modest EC2 instances. A small HPC cluster could potentially dedicate one node for running Crux on virtual machines or containers instead of directly on bare metal systems.

6.4. Discussion

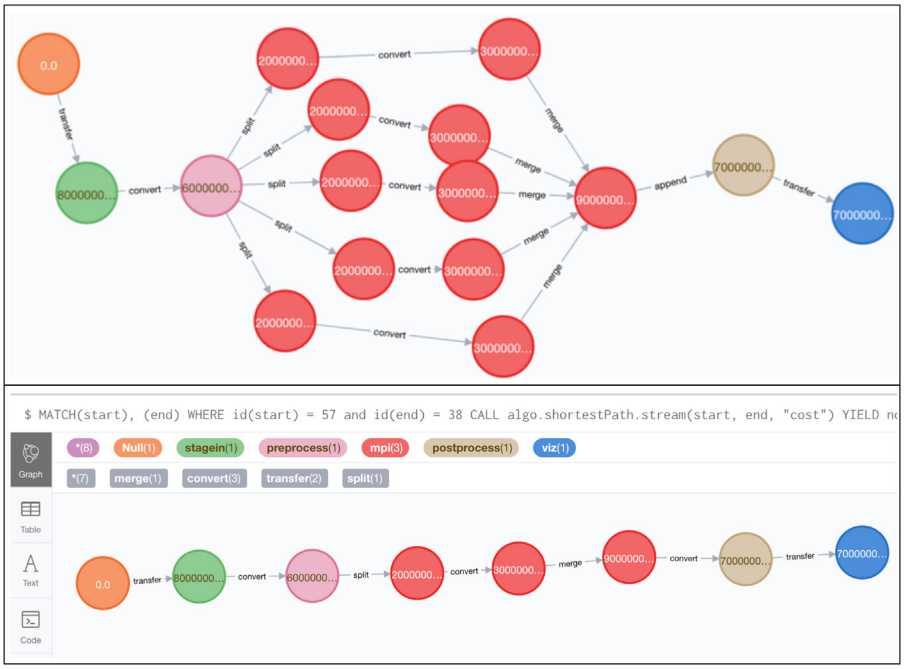
Our performance experiments suggest that network proximity of Crux to application clients improves Crux's performance. This is consistent with our original expectations. Surprisingly, we observed that scaling out Crux API instances did not improve overall performance of Crux on AWS. This suggests that the limiting factor to Crux's performance may be the load balancer responsible for distributing traffic to the API instances. Another limiting factor could be the performance of Crux's database. Having multiple API instances would have diminishing returns if Crux's Neo4j instance is unable to process more requests.

Through our tests we observed limitations with the capability of the Neo4j Browser for Crux: inability to highlight a path within a graph; incorrect inclusion of one or more edges; and poor scaling to more than 2,000 vertices. Also, Neo4j Browser did not offer ways for us to visualize results from previous workflow runs or easily export graph data. Although sufficient for our initial prototype, a more comprehensive UI would be warranted for a production tool.

7. Conclusions and future work

In this paper, we introduced a novel metric, Workflow Critical Path (WCP) for Holistic HPC Workflows. We described a prototype tool called Crux for calculating WCP. To evaluate Crux we developed a set of simulators to simulate HPC workflows and workflow patterns; and designed a cloud-based, test environment on AWS. Early results suggest that Crux can be used to efficiently calculate WCP. WCP shows promise as a useful diagnostic metric focused across an entire workflow.

Our continuing efforts include improvements to the prototype: a custom Crux GUI to address the limitations we observed with the Neo4j Browser, and the functionality to easily allow a user to save workflow critical path results from multiple runs. For Crux to be adopted to



```
$ MATCH(start), (end) WHERE id(start) = 57 and id(end) = 38 CALL algo.shortestPath.stream(start, end, "cost") YIELD n
```

Fig. 11. Generic workflow PAG and WCP. Top image shows the entire PAG with 5 jobs: stage in (green), preprocess (pink), MPI (red), postprocess (tan), and visualization (blue). All Crux PAGs begin with a null vertex (orange) created during database initialization. Bottom image shows the WCP. The critical path in this execution is the path with longest elapsed time through the MPI job. Cost = 9.492.

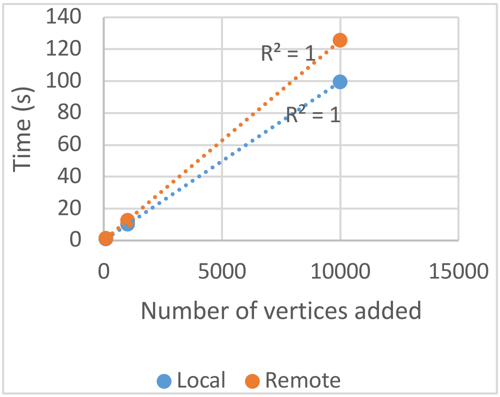


Fig. 12. Time to add vertices to local and remote deployment of Crux. Measured time required for a local Python client make Crux API calls to add 100, 1000, and 10000 data state vertices on local and remote deployments of Crux. In all cases, the time to created vertices was less on the locally deployed Crux. Trendline for both cases suggest a linear relationship between number of vertices to add and overall time ($R^2 = 1$).

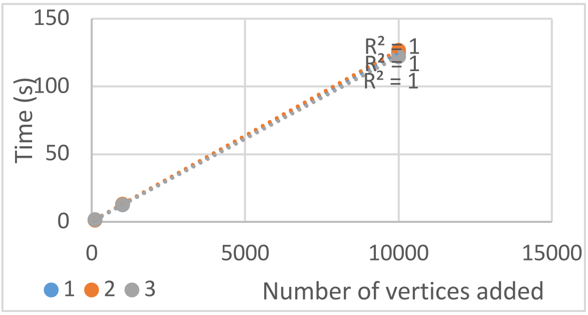


Fig. 13. Time to add vertices against number of Crux API instances. Measured time required for a local Python client make Crux API calls to add 100, 1000, and 10000 data state vertices on a remote deployment of Crux with 1, 2, and 3 API server instances. There was a significant difference in time to add 100 vertices between 1, 2, and 3 API server instances at $p < .05$ [F(2, 6) = 2.6949, p = 0.0135] where 1 API server instance performed fastest (mean = 1.3792 s). Results did not show significant difference in time for adding 1000 and 1000 vertices between 1, 2, and 3 API server instances. Trendline in all cases suggest a linear relationship between number of vertices to add and overall time ($R^2 = 1$).

production use, the instrumentation must be automated, whereas the initial prototype requires manual insertion of instrumentation. We are currently testing full-scale applications with Crux.

Acknowledgments

David Montoya was instrumental in the development of key insights and ideas contained in this work. Sonja Johanson performed testing and documented the Crux tool, and Yasodha Suriyakumar provided the DroughtHPC example. PSU students Jaspar Alt, Kobe Davis, and Kristina Frye participated in group discussions. Portions of this work were conducted at the Ultrascale Systems Research Center (USRC) supported by Los Alamos National Laboratory, United States under Contract No. DE-AC52-06NA25396 with the U.S. Department of Energy. This work supported in part by the New Mexico Consortium.

References

[1] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P.J. Maechling, R. Mayani, W. Chen, R.F. d. Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Gener. Comput. Syst. 46 (2015) 17–35.

[2] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M.B. Jones, E.A. Lee, J. Tao, Y. Zhao, Workflow management and the Kepler system, in: Concurrency and Computation: Practice and Experience, vol. 18, (10) 2006, pp. 1039–1065.

[3] E. Deelman, T. Peterka, I. Altintas, C.D. Carothers, K.K. v. Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, J. Vetter, The future of scientific workflows, Int. J. High Perform. Comput. Appl. 32 (1) (2017) 159–175.

[4] S. Moore, D. Cronk, K. London, J. Dongarra, Review of performance analysis tools for MPI parallel programs, in: EuroPVM/MPI: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Santorini/Thera, Greece, 2001, pp. 23–26.

[5] J. Sairabanu, M.R. Babu, A. Kar, A. Basu, A survey of performance analysis tools for OpenMP and MPI, Indian J. Sci. Technol. 9 (43) (2016) 1–7.

[6] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, N.R. Tallent, HPCToolkit: Tools for performance analysis of optimized parallel program, Concurr. Comput.: Pract. Exper. 22 (6) (2010) 685–701.
[7] S.S. Shende, A.D. Malony, The TAU parallel performance system, Int. J. High Perform. Comput. Appl. 20 (2) (2006) 287–311.
[8] S. Snyder, P. Carns, K. Harms, R. Ross, G.K. Lockwood, N.J. Wright, Modular HPC I/O characterization with Darshan, in: 2016 5th Workshop on Extreme-Scale Programming Tools, Salt Lake City, UT, 2016.
[9] D. Skinner, Performance monitoring of parallel scientific applications, 2005, [Online]. Available: https://www.osti.gov/servlets/purl/881368-dOvpFA/ (Accessed 15 2021).
[10] E. Deelman, T. Peterka, I. Altintas, C.D. Carothers, K.K. v. Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, J. Vetter, The future of scientific workflows, Int. J. High Perform. Comput. Appl. 32 (1) (2017) 159–175.
[11] Gromacs, [Online]. Available: http://www.gromacs.org/ (Accessed 15 2021).
[12] F. Affinito, A. Emerson, L. Litov, P. Petkov, R. Apostolov, L. Axner, B. Hess, E. Lindahl, M.F. Iozzi, Performance Analysis and Petascaling Enabling of GROMACS, 2012, [Online]. Available: http://www.prace-ri.eu/IMG/pdf/Performance_Analysis_and_Petascaling_Enabling_of_GROMACS.pdf (Accessed15 2021).
[13] C. Herold, B. Williams, Top-down performance analysis of workflow applications, in: The International Conference for High Performance Computing, Networking, Storage, and Analysis, Dallas, TX, 2018.
[14] Y. Suriyakumar, K.L. Karavanic, H. Moradkhani, Performance Analysis of DroughtHPC and Holistic HPC Workflows, ICPP 2018 Research Poster Extended Abstract, 2018, [Online]. Available: http://oaciss.uoregon.edu/icpp18/publications/pos131s2-file1.pdf (Accessed 15 2021).
[15] J.J. Hamman, B. Nijssen, T.J. Bohn, D.R. Gergel, Y. Mao, The variable infiltration capacity model version 5 (VIC-5): infrastructure improvements for new applications and reproducibility, Geosci. Model Dev. 11 (8) (2018) 3481–3496.
[16] H. Cooney, H. Yan, K. Karavanic, H. Moradkhani, A Workflow-Based Performance Study of a Drought Prediction System, 2016, [Online]. Available: https://pdxscholar.library.pdx.edu/cgi/viewcontent.cgi?article=1002&context=mcecs_mentoring (Accessed 15 2021).
[17] APEX Workflows, 2016, [Online]. Available: https://www.nersc.gov/assets/apex-workflows-v2.pdf (Accessed 15 2021).
[18] N. Tallent, D. Kerbyson, A. Hoisie, Representative paths analysis, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17). ACM, 2017.
[19] J.K. Hollingsworth, Critical path profiling of message passing and shared-memory programs, IEEE Trans. Parallel Distrib. Syst. 9 (10) (1998).
[20] C. Yang, B. Miller, Critical path analysis for the execution of parallel and distributed programs, in: Proc. of the 8th Intl. Conf. on Distributed Computing Systems, IEEE, 1988, pp. 366–373.
[21] K.L. Karavanic, Performance Tools and Holistic HPC Workflows, 2018, [Online]. Available: https://dyninst.github.io/scalable_tools_workshop/petascale2018/assets/slides/Karavanic2018.pdf (Accessed 15 2021).
[22] B.H. Sigelman, L.A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag, Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, 2010, [Online]. Available: https://static.googleusercontent.com/media/research.google.com/en//archive/papers/dapper-2010-1.pdf (Accessed 15 2021).
[23] J. Aldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K.W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, Y.J. Song, Canopy: An end-to-end performance tracing and analysis system, in: SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles, 2017, pp. 34–50.
[24] M. Schulz, Extracting Critical Path Graphs from MPI applications, in: 2005 IEEE International Conference on Cluster Computing, Burlington, MA, 2005, pp. 27–30.
[25] I. Dooley, L.V. Kale, Detecting and using critical paths at runtime in message driven parallel programs, in: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), Atlanta, GA, 2010, pp. 19–23.
[26] J. Chen, R.M. Clapp, Critical-path candidates: scalable performance modeling for MPI workloads, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, 2015, pp. 29–31.
[27] D. Bohme, F. Wolf, D.R. De Supinski, M. Schulz, M. Geimer, Scalable critical-path based performance analysis, in: Proc. of the 26th IEEE Intl. Parallel and Distributed Processing Symp, IEEE, 2012, pp. 1330–1340.
[28] C. Alexander, D. Reese, J. Harden, Near-critical path analysis of program activity graphs, in: Proc. of the Second Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, IEEE, 1994, pp. 308–317.
[29] U. Meyer, P. Sanders, Δ-Stepping: a parallelizable shortest path algorithm, J. Algorithms 49 (1) (2003) 114–152.
[30] M. Kranjčević, D. Palossi, S. Pintarelli, Parallel delta-stepping algorithm for shared memory architectures, in: 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2016), 2016.
[31] FastAPI, [Online]. Available: https://fastapi.tiangolo.com/ (Accessed 15 2021).
[32] neo4j, [Online]. Available: https://neo4j.com/ (Accessed 15 2021).