Workflow Critical Path: a Data-Oriented Path Metric for Holistic HPC Workflows

Daniel D. Nguyen
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Computer Sciences Commons

Let us know how access to this document benefits you.

Recommended Citation
https://doi.org/10.15760/etd.7369

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.
Workflow Critical Path:
A Data-Oriented Path Metric for Holistic HPC Workflows

by

Daniel D. Nguyen

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Karen L. Karavanic, Chair
R. Bruce Irvin
Wu-chang Feng

Portland State University
2020
Abstract

Optimizing scientific application performance in HPC environments is a complicated task which has motivated the development of many performance analysis tools over the past decades. These tools were designed to analyze the performance of a single parallel code using common approaches such as message passing (MPI), multithreading (OpenMP), acceleration (CUDA), or a hybrid approach. However, current trends in HPC such as the push to exascale, convergence with Big Data, and growing complexity of HPC applications and scientific workflows, have created gaps that these performance tools do not cover, particularly involving end-to-end data movement through an HPC workflow comprising multiple codes, paradigms, or platforms.

To address this performance monitoring gap, we define a new metric called Workflow Critical Path (WCP), a data-oriented critical path metric for Holistic HPC Workflows. Using cloud-based technologies, we implement a prototype called Crux, a distributed analysis tool for calculating and visualizing WCP. Crux takes a novel, data-oriented approach by constructing program activity graphs (PAGs) using data states as vertices and data mutations as edges. Our experiments with a workflow simulator on Amazon Web Services show Crux is scalable and capable of calculating WCP for common Holistic HCP workflow patterns. We discuss how Crux and WCP could be used with production HPC applications.
Table of Contents

Abstract ................................................................................................................................ i

List of Tables ...................................................................................................................... iv
List of Figures ..................................................................................................................... v

Chapter 1: Introduction ....................................................................................................... 1
  1.1 Motivation ................................................................................................................. 3
  1.2 Definitions ................................................................................................................. 5
  1.3 Thesis Statement ....................................................................................................... 6
  1.4 Contributions ............................................................................................................. 6

Chapter 2: Background ........................................................................................................ 9
  2.1 Parallel Computing .................................................................................................... 9
  2.2 Critical Path Analysis ................................................................................................ 9
  2.3 High Performance Computing ................................................................................ 11
  2.4 Holistic HPC Workflows ........................................................................................ 12
  2.5 Instrumentation, Profiling, and Tracing .................................................................. 12

Chapter 3: Related Work ................................................................................................... 14
  3.1 Workflow Management Systems ............................................................................ 14
  3.2 Distributed Systems Tracing Tools ......................................................................... 17
  3.3 HPC Performance Measurement Tools ................................................................... 20
  3.4 Performance Analysis of Scientific Workflows ...................................................... 21

Chapter 4: Architecture ..................................................................................................... 22
  4.1 Data State ................................................................................................................ 22
  4.2 Crux UI .................................................................................................................... 24
  4.3 Crux API .................................................................................................................. 24
  4.4 Crux Database .......................................................................................................... 27
  4.5 Crux Critical Path Algorithm .................................................................................. 29
  4.6 Deployment in HPC clusters ................................................................................... 31
  4.7 Instrumentation of HPC Applications ...................................................................... 33
List of Tables

Table 1.2.1: HPC cluster categories with example clusters ................................................ 6
Table 3.1.1: Examples of Workflow Management Systems ............................................... 17
Table 4.1.1: Types of data mutations in Crux .................................................................. 23
Table 4.4.1: The Crux API .............................................................................................. 29
Table 4.8.1: Types of instrumentation approaches with pros and cons ............................ 34
Table 5.7.1: Tools and their specifications used in Crux experiments ............................. 55
Table 7.3.1: Crux API events in workflow simulator applications and implication for
          HCP applications ....................................................................................................... 78
Table 7.3.2: Estimating number of Crux API calls needed for DroughtHPC ................. 82
List of Figures

Figure 1.1.1: Application layer of DroughtHPC workflow ................................................ 4
Figure 3.1.1: Example Scientific Workflow showing data movement ............................. 15
Figure 4.1.1: Crux definitions for data state vertex and data mutation edge ................. 22
Figure 4.1.2: Example of a data state undergoing different mutations ......................... 23
Figure 4.3.1: Crux API interaction with HPC clients ....................................................... 26
Figure 4.6.1: Deployment of Crux in an HPC cluster ....................................................... 32
Figure 5.1.1: Crux OpenAPI Schema in browser, Part 1 .................................................. 38
Figure 5.1.2: Crux OpenAPI Schema in browser, Part 2 .................................................. 39
Figure 5.1.3: Crux OpenAPI Schema in browser, Part 3 .................................................. 40
Figure 5.3.1: Handling MERGE data mutations to use with Neo4j's shortest path algorithm ................................................................................................................... 44
Figure 5.4.1: Neo4j Browser ............................................................................................. 46
Figure 5.5.1: Crux Workflow Simulator deployed with Crux .......................................... 47
Figure 5.6.1: Example of local deployment of Crux with simulators using Docker ......... 50
Figure 5.7.1: AWS Elastic Cloud Service Terms .............................................................. 51
Figure 5.7.2: Example entries in the service registry used to identify Crux components 52
Figure 5.7.3: Configuring an AWS Application Load Balancer to route traffic to Crux components ................................................................................................................ 53
Figure 5.7.4: Example of target group apiserver ............................................................... 53
Figure 5.7.5: Networking of Crux on AWS ..................................................................... 54
Figure 6.1.1: Generic workflow PAG ............................................................................... 57
Figure 6.1.2: Generic workflow critical path ................................................................... 58
Figure 6.1.3: Crux PAG showing multiple data splits with 150 MPI ranks ..................... 59
Figure 6.1.4: Workflow critical path for PAG showing multiple data splits with 150 MPI ranks ........................................................................................................................................ 60
Figure 6.1.5: Crux PAG showing checkpoint files .......................................................... 61
Figure 6.1.6: Workflow critical path of PAG showing checkpoint files ........................... 62
Figure 6.1.7: Crux PAG showing two input data sources ................................................. 64
Figure 6.1.8: Workflow critical path of Crux PAG showing two input data sources ...... 64
Figure 6.1.9: Crux PAG showing file creation and deletion .............................................. 66
Figure 6.1.10: Workflow critical path of Crux PAG showing file creation and deletion . 67
Figure 6.2.1: Mean latency of HTTP servers at 50 requests per second ......................... 68
Figure 6.2.2: Mean latency of containerized HTTP servers at 50 requests per second .... 69
Figure 6.2.3: Request rate of HTTP servers against number of max workers ............... 69
Figure 6.2.4: Time to add vertices to local and remote deployment of Crux .................. 70
Figure 6.2.5: Time to add vertices against number of Crux API instances ..................... 71
Figure 6.2.6: Request count to Crux measured by AWS ALB ........................................ 72
Figure 6.2.7: Target response time of Crux measured by AWS ALB .............................. 73
Figure 6.2.8: Request count per target measured by AWS ALB ................................... 73
Figure 6.2.9: CPU and memory utilization for API server and graph database instances on AWS .................................................................................................................................... 74
Figure 7.1.1: Neo4j Browser attempting to display 10000 data state vertices ............... 76
Figure 7.3.1: Python application instrumented with 6 Crux API calls ......................... 79
Figure 7.3.2: C application instrumented with 6 Crux API calls ................................. 80
Figure 7.3.3: Crux PAG from Python and C application instrumented with 6 Crux API calls
Chapter 1: Introduction

High performance computing (HPC) clusters operate on a scale of hundreds to tens of thousands of physical machines, called nodes, and require large amounts of resources to build, run, and maintain. In order to fully optimize cluster performance, a variety of performance analysis tools and techniques have been developed. For instance, tools such as HPCToolkit [1] and TAU Performance System [2] use profiling and tracing techniques to detect performance bottlenecks in parallel applications. Some tools such as Darshan [3] and IPM [4] 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. Some performance tools solely target MPI which is the standard for scientific applications to execute parallel computation [5] [6]. One important metric used by these tools for analyzing HPC applications is critical path analysis (CPA), which examines an MPI codebase for the sequence of tasks that take the longest to complete, i.e. the critical path.

Trends over the last decade have led to changes in HPC applications that present new challenges and motivation for studying their performance, for example the push towards exascale and convergence with Big Data, the increasing complexity of HPC applications, and the expansion of scientific workflows. First, the push of HPC towards exascale and its convergence with Big Data means that achieving good performance with HPC applications is more difficult than ever. Use cases from fields such as machine learning, artificial intelligence, and autonomous vehicles bring a new scale of data sets that must be processed by supercomputers. Dataflow may become more of a common source
of bottlenecks for applications and researchers cannot just depend on faster hardware to improve application performance. Second, HPC applications are increasingly complex in terms of their codebase. It is now common for HPC applications to consist of multiple serial and parallel codes or libraries, with each one calling other HPC applications. For example, a science code might call an existing modeling code that is treated as a black box. This presents a need for tools that can detect performance bottlenecks between applications. Third, the growing complexity of scientific workflows requires new ways of characterizing performance at the workflow level [7]. Workflow management systems (WMS) like Pegasus offer researchers a way to organize and execute their scientific jobs. However, analyzing workflow performance is very platform dependent and tightly coupled to the WMS. These WMS do not solve the problem of identifying critical paths for workflows designed outside of such a system.

In this thesis, we address the challenge of detecting critical path in Holistic HPC Workflows. While critical path analysis techniques for parallel applications have evolved [8] [9] [10], there are few techniques for performing CPA across Holistic HPC Workflows. Having such a diagnostic tool would enable a variety of outcomes. First, it would provide a precise metric and mechanism for describing performance at the workflow level, a procedure which today is undefined. Second, examining performance of Holistic HPC Workflows at runtime would help researchers better understand data movement patterns and potential bottlenecks occurring across the complex memory hierarchy and storage system in an HPC cluster. Third, possessing the critical path for workflows would precisely inform developers where to focus workflow optimization efforts instead of requiring them to manually instrument each participating application or
run a suite of performance tools and analyze overwhelming amounts of output. Finally, as HPC workflows become more complex and the size and source of their input data increases, having critical path for workflows will present a new way for describing performance issues with data distribution and management in mind, ultimately allowing scientific applications to scale more effectively.

This thesis presents Workflow Critical Path (WCP), a data-oriented critical path metric for Holistic HPC Workflows. We define WCP, the challenges it aims to solve, the environments it targets, and how it can be integrated into existing HPC workflows. We present a prototype called Crux, a distributed, critical path analysis tool for Holistic HPC Workflows. We demonstrate Crux’s ability to calculate WCP for a variety of workflow patterns using a workflow simulator and showcase its performance on Amazon Web Services.

1.1 Motivation

The following cases are real-world examples that demonstrate the need for a performance tool that can diagnose a runtime HPC workflow. We use these examples as motivation for WCP.

1.1.1 Case: DroughtHPC and VIC

DroughtHPC is a program developed at Portland State University by Moradkhani et al. to predict drought for a target geographical area [11]. It utilizes the Variable Infiltration Capacity (VIC) model, a macroscale semi-distributed hydrologic model, to simulate meteorological samples over a given time period [12]. To obtain a sufficient number of samples, DroughtHPC calls VIC in a loop (Figure 1.1.1). Every call to VIC
inputs and outputs 25 files. The number of samples needed is multiplied by the number of
days needed.

![Figure 1.1.1: Application layer of DroughtHPC workflow](image)

Performance analysis of DroughtHPC shows that while MPI portions of the code are embarrassingly parallel, locating workflow bottlenecks, particularly due to dataflow, was not so apparent [13]. To investigate performance bottlenecks, researchers manually ran a variety of measurement tools like the python profiler, dtrace, and PerfTrack. Only after running these tools could they discover that 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. They achieved significant improvement by configuring the use of local disks for storing the intermediate files, which reduced contention for the shared NFS file server [11]. Overall, DroughtHPC shows a need for one performance tool that can detect common dataflow patterns and diagnose runtime bottlenecks across different phases in a scientific workflow.

1.1.2 Case: GROMACS Workflow

The Groningen Machine for Chemical Simulations (GROMACS) is a scientific framework for simulating molecular dynamics of biochemical modules such as proteins, lipids, and nucleic acids [14]. GROMACS 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 [15]. Each step can correspond to a single application communicating with each other over a shared file system and managed by a job scheduler like SLURM.

Analyzing the workflow performance of GROMACS is difficult especially when workflows become complex and a common tool for determining bottlenecks does not exist [16]. Attempts to solve this challenge include using a top-down approach by deconstructing workflow into I/O, communication, and computation components and subsequently instrumenting the workflow applications to record these metrics [15]. This approach can in theory be used to analyze other scientific applications that follow the GROMACS workflow. Nevertheless, GROMACS shows us that while in-application performance can be analyzed and monitored using existing performance tools, there still lacks a common tool that can provide a holistic measurement for workflow performance.

1.2 Definitions

We define the following “sizes” to classify HPC clusters [17] [18] [19] [20]

<table>
<thead>
<tr>
<th>Size</th>
<th>System</th>
<th>Compute nodes</th>
<th>Cores</th>
<th>Storage</th>
<th>FLOPS per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small ~100 nodes</td>
<td>Yeti – Columbia University Information Technology</td>
<td>167</td>
<td>2672</td>
<td>160 TB</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>El Gato – University of Arizona</td>
<td>131</td>
<td>2160</td>
<td>N/A</td>
<td>46 Tera</td>
</tr>
<tr>
<td>Medium 200+ nodes</td>
<td>Savio – Berkeley Research Computing Institutional/Condo Cluster</td>
<td>470</td>
<td>11,620</td>
<td>1.4 PB</td>
<td>450 Tera</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------------------------------</td>
<td>-----</td>
<td>--------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>MARCC/Bluecrab – University of Maryland</td>
<td>846</td>
<td>21,792</td>
<td>2.0 PB</td>
<td>900 Tera</td>
</tr>
<tr>
<td>Large 1000+ nodes</td>
<td>Blue Gene/Q Mira – IBM</td>
<td>49,152</td>
<td>786,432</td>
<td>70 PB</td>
<td>10 Peta</td>
</tr>
<tr>
<td></td>
<td>Summit – IBM Power System</td>
<td>4,608</td>
<td>202,752</td>
<td>250 PB</td>
<td>200 Peta</td>
</tr>
</tbody>
</table>

Table 1.2.1: HPC cluster categories with example clusters

1.3 Thesis Statement

Web- and cloud-based technologies can be used to efficiently calculate Workflow Critical Path, a data-oriented critical path metric for Holistic HPC Workflows.

1.4 Contributions

1. Defined a new performance metric called Workflow Critical Path:

We define a new metric called Workflow Critical Path (WCP) for Holistic HPC Workflows. WCP allows researchers to better understand how a holistic HPC workflow’s composition effects its overall, end-to-end performance. WCP describes the critical path for an entire HPC workflow by tracing the flow of data between HPC applications. It does this by defining a program activity graph (PAG) where vertices represent data state and edges represent data mutations.

2. Developed a prototype tool for WCP called Crux:
We implement Crux, a distributed, critical path analysis tool for Holistic HPC Workflows. 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. Crux is composed of the following modules:

a. **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 and interfaces with the Crux Database. The Crux API follows a REST architecture for benefits such as scalability and portability. In addition, the API performs data integrity checks and manages Crux’s performance metadata.

b. **Crux Database**: A database that stores a workflow PAG. Executes Crux’s critical path algorithm for finding the WCP.

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

3. **Developed a workflow simulator capable of simulating custom HPC workflows**: We developed a system of simulators to create custom HPC workflows. By following a common template, these modules are highly configurable and extendable. Simulating HPC workflows provides a convenient way to test Crux. It also allows users to simulate representative workloads (e.g. parallel MPI code, serial data preprocessing) in a configurable way.

4. **Designed a cloud-based test environment for Crux**: 


We developed tooling to automatically deploy and test Crux on Amazon Web Services. In addition, we containerize Crux components and designed a local deployment method of Crux using Docker. We configure Crux’s workflow simulator to run locally or on AWS.
Chapter 2: Background

2.1 Parallel Computing

Parallel computing involves using more than one processor thread to complete a task. For example, we can take advantage of multi-core processors by dividing a task into independent subtasks, i.e. subtasks that can be run concurrently, executing them on different cores, and merging the results if necessary. Parallel programming can fall into different categories, the two common being functional parallelism and data parallelism [21]. In functional parallelism, each process performs different sections of code that are independent. Data parallelism, on the other hand, involves performing the same calculation on independent pieces of data.

In HPC common paradigms include message passing with MPI and multithreading with OpenMP. The Message Passing Interface (MPI), is a specification for developers of message passing libraries [22]. In the message-passing parallel programming model, data is copied from the address space of one process to another process through cooperative operations on each process. Since its development in the early 1990s, MPI has become the industry standard for writing message passing programs on HPC platforms [23].

2.2 Critical Path Analysis

Critical path analysis (CPA) is a branch of parallel performance research that seeks to find the longest path of dependent activities and its time required to complete. This type of analysis is suited for parallel programs with the following characteristics [24]:

1. Can be broken down into a number of separate activities
2. Time required for each activity can be measured
3. Some activities must be executed serially while others are carried out in parallel
4. Each activity requires combination of resources (e.g. CPU, memory, I/O) and it is possible to have more than one combination of these resources resulting to different duration of execution

While the origin of CPA is rooted in operational fields such as project planning and scheduling starting the late 1950s and 1960s [25] [26], seminal research into CPA for parallel programs started in the 1980s with work such as Yang & Miller [24]. 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. Taking shortest path algorithms studied in graph theory [27], they were able to return the longest path on a scale of tens of thousands of nodes [24].

Critical path analysis evolved in the 1990s with techniques such as using a nontrace-based algorithm to compute the critical path profile during runtime [9]. 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 [9]. By the 2000s, critical path analysis evolved to target MPI and other parallel applications. For example, Schultz looked at extracting critical path graphs from MPI applications [28], Dooley & Kale performed runtime critical path detection for message driven parallel programs [29], and Chen & Clapp introduced critical-path candidates as a scalable performance modeling framework for MPI workloads up to 16K MPI ranks [30].

Many CPA approaches rely on storing portions of a PAG on each processor. Messages sent between each process gets augmented with the critical path leading up to
the message send. Thus, the critical path can be extracted through a backwards traversal of the distributed PAG [29]. 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 [10] [31].

2.3 High Performance Computing

High performance computing (HPC) refers to the practice of aggregating compute resources to deliver performance higher than possible with individual computers [32]. One typical HPC compute platform is a cluster of interconnected compute nodes. HPC clusters are tightly coupled compute, storage, and networking resources that enable users to run large scale scientific and engineering workloads. These workloads traditionally span a variety of government-funded or academic domains such as physics, seismology, astrology, electromagnetic and fluid dynamic simulations, molecular, genetics, and other life science research. However, more and more HPC workloads are coming from the commercial sector to perform jobs such as media and animation rendering, financial and economic simulations, weather prediction, machine learning, and more [33].

HPC clusters come in many sizes. The term “supercomputer” refers to the most powerful HPC systems in the world, a list of which is compiled annually by TOP500 [34]. At the time of this thesis, the fastest supercomputer in the world is the Summit (OLCF-4) built by IBM at the Oak Ridge National Laboratory, with 9216 IBM POWER9 22-core CPUs and a peak speed 200 petaflops [35]. Currently underway are plans to develop exascale supercomputers, i.e. a system capable of \(10^{18}\) or more floating point operations per second (FLOPS) [36].
2.4 Holistic HPC Workflows

Holistic HPC workflow performance diagnosis encompasses the monitoring and analysis of performance problems that span across traditionally separated aspects of an HPC effort [37]. This targets a common compute paradigm, where the workflow comprises a set of codes or applications, likely developed by different people or groups, possibly written in different languages. It includes codes not developed within a workflow management systems. Holistic HPC Workflow is important for emerging HPC applications that work on large datasets where dataflow bottlenecks can exist. By 2020, it is predicted that the average internet user will generate 1.5 GB of traffic per day [38]. Smart hospitals will be generating over 3,000 GB per day. A connected plane, over 40,000 GB per day. A connected factory over 1,000,000 GB per day. As exascale systems become a reality, dataflow bottlenecks need to be addressed.

2.5 Instrumentation, Profiling, and Tracing

Instrumentation is the insertion of code into a program for the purpose of performance measurement. Instrumentation can be classified based on the level at which code is being inserted: source code, compiler-based, or binary [39]. Source-code instrumentation requires users to add calls directly into a program’s source code. The benefit of instrumentation at this level is high portability in languages and granularity in where instrumentation calls are placed within routines. Compiler-based instrumentation can track routine entry and exit points using symbol and line map information recorded in an executable. Since this often involves handling dynamic shared objects, mapping source
code locations, and resolving symbol names, compiler-based instrumentation can carry a higher overhead compared to source-code instrumentation. Binary instrumentation inserts probes into a running program. By using a tool like Dyninst, probes can be placed at interval events or loops boundaries [39].

There are two common approaches to performance measurement: profiling and tracing. Profiling is largely based on sampling—interrupting a running program at fixed intervals to examine events of interest. The result is an inventory of performance events along with the timing of program execution as a whole. For example, the Linux profiler prof can be used to sample on-CPU kernel instructions and produce an inventory of instructions being executed within a time period [40]. Tracing, on the other hand, generates a log of timestamped events within a program. Tracing can reveal when and where an event occurred, such as when certain MPI calls are made in a parallel program. Both profiling and tracing can depend on some form of instrumentation. For example, for profiling based on measured process timing, instrumentation is triggered at function entries and exits [41]. Unlike profiling, data from a trace can be large and must be periodically written to a file or stable storage for later analysis. Common profiling and tracing tools used in HPC include Score-P, Vampir, TAU, and HPCToolkit [42].
Chapter 3: Related Work

3.1 Workflow Management Systems

Workflows are used in many scientific and computer science domains. While no universal definition exists, we refer to definitions used by research institutions focused on scientific workflow such as the Los Alamos National Lab (LANL), National Energy Research Scientific Computing Center (NERSC), and Sandia National Laboratories (SNL).

- **Task**: A logical entity or program that consumes and/or produces data, with a single task potentially mapping to a job, process, thread, or even function call [43].

- **Workflow**: A description of tasks needed to obtain results in a scientific investigation [44]. Tasks are commonly represented as directed acyclic graphs (DAGs), whose nodes represent workflow tasks that are linked via dataflow edges [45].

- **Workflow management system**: Framework for the design, execution, and monitoring of scientific workflows. Allows scientists to model entire data processing steps and their dependencies while abstracting details of application execution on HPC infrastructure [45]. Aids in the automation of workflows, namely, managing the execution of constituent tasks and the information exchanged between them [7].

Given different definitions and classes of workflows, workflow patterns can be difficult to generalize. The Alliance of Application Performance at Extreme Scale (APEX) [46] broadly categorizes common HPC workflows into two classes: large-scale scientific simulation and data-intensive workflows.
Figure 3.1: Example Scientific Workflow showing data movement
Common stages in a scientific workflow (orange) with input and output data (green). Data is shown moving through different memory layers (blue).

Figure 3.1.1 describes the phases associated with the more common large-scale scientific simulation workflow, 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. The campaign timescale includes data used throughout the execution of entire scientific workflow. The forever timescale describes data kept beyond one workflow run. Phases represent series of parallel and/or serial jobs submitted to a batch scheduler and executed when sufficient resources exist to execute submitted jobs. In Phase S1 (described as “stage in” in Figure 3.1.1), a scientist defines and creates the initial state from an input data set and writes this to file (e.g. creation of a complex 3-
D mesh). In Phase S2 (described as “pre-preprocess” in Figure 3.1.1), the computationally intensive simulation begins and runs for a number of times. Checkpoint dumps allow for jobs to continue or restart where one ends. Phase S2 can also provide timestep data so scientists to examine previously generated results at full resolution. Phase S3 shows the down-sampling of analysis data that occurs in conjunction with the S2 phase. Where S2 is the computationally intensive phase, S3 is typically the I/O intensive phase. Once all datasets are down sampled, they are typically post-processed to conform to some configuration for visualization or reporting tools which is the purpose of Phase S4 and S5.

Workflow management systems enable scientists to create and execute workflows without fully needing to provision underlying resources such as hardware and networking. However, choosing the right WMS can be difficult simply because selection criteria and features between each system may not be obvious. To address this problem, researchers have attempted to survey and compile comparable differences between WMS [47] [43]. For example, Table 3.1.1 highlights descriptions of popular HPC workflow management systems.

<table>
<thead>
<tr>
<th>WMS</th>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kepler</td>
<td>2005</td>
<td>A free software system for designing, executing, reusing, evolving, archiving, and sharing scientific workflows</td>
</tr>
<tr>
<td>Pegasus</td>
<td>2005</td>
<td>Maps abstract workflow descriptions onto distributed computing infrastructures</td>
</tr>
<tr>
<td>Fireworks</td>
<td>2015</td>
<td>A dynamic workflow system for running high-throughput calculation workflows at supercomputing centers</td>
</tr>
<tr>
<td>Swift</td>
<td>2007</td>
<td>Swift is a scripting language designed for composing application programs into parallel applications that can be executed on multicore processors, clusters, grids, clouds, and supercomputers</td>
</tr>
<tr>
<td>Taskfarmer</td>
<td>N/A</td>
<td>A utility developed in-house at NERSC to farm tasks onto a compute node - these can be single- or multi-core tasks. It tracks which tasks</td>
</tr>
</tbody>
</table>
have completed successfully, and allows straightforward re-submission of failed or un-run jobs from a task list.

Table 3.1.1: Examples of Workflow Management Systems

The distinction between WCP and WMS is that WCP is a metric and runtime tool to diagnose performance bottlenecks across holistic HPC workflows. It is not responsible for organizing or executing workflow applications. While WMS commonly provide performance metrics to end user, they are often tightly coupled to the system itself. We propose a higher-level abstraction focusing on data state to characterize performance for HPC workflows that take into consideration dataflow patterns and multiple code bases.

In order to effectively test WCP, we developed WMS-like features (e.g. task scheduling, application monitoring, archival data storage) as part of Crux’s workflow simulator. Such features allow us to schedule and simulate workflow tasks. We choose not to operate Crux with a WMS for the main reason that our motivation for Crux comes from HPC applications not developed using a WMS.

3.2 Distributed Systems Tracing Tools

Distributed systems are a long-standing area of research and used throughout industry. However, characterizing and understanding their end-to-end performance is still a challenge [48]. Monitoring tools can measure frequency of anomalies like server crashes but offer poor insight into root causes. Correlating log events across large number of systems and visualizing them effectively is another key challenge [49]. Modern internet services—often comprised of a collection of different applications (i.e. microservices)—may be written in different programming languages and may span many thousands of machines across multiple facilities [50]. To address these challenges, distributed tracing
tools have emerged within the last decade to provide high-level explanations for end-to-end interactions across entire systems.

Distributed systems tracing tools fall into two main categories: black box monitoring schemes and annotation-based schemes [51]. Black box monitoring schemes treat system components as black boxes and attempt to infer causal relationships by using statistical inference methods. For example, a black box monitoring system may observer network traffic among system components and analyze the messaging data to infer causal relationships. Annotation-based (also referred to as application-instrumented) monitoring schemes 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 since there is no need for inference but come with the tradeoff that all system components must be instrumented.

Dapper, developed by Google, is an example tool that uses an annotation-based scheme and is the first to report on a large, production distributed systems tracing framework [50]. In a Dapper trace tree, the tree nodes are basic units of work which 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 trace tree thus can be thought of as a tree of nested RPCs. A span can contain information from multiple hosts and in fact every RPC span contains annotations from both the client and server processes. Dapper also provides an API to simplify access to trace data their repository. Developers at Google use this API to build both general-purpose and application specific analysis tools.
While distributed tracing tools have originated from internet enterprises and target their operations, the scale on which they are deployed at suggests potential use cases for HPC workflows. For example, Twitter’s distributed tracing tool, Zipkin, works with their time series ingestion service which, in a 2016 report, handles more than 2.8 billion write requests per minute, stores 4.5 petabytes of data, and handles 25,000 query requests per minute [52]. As of 2019, Twitter’s services handle approximately 30 petabytes of data and 5 billion writes per minute. Similarly, HPC scientists also work with tens and hundreds of terabytes of data [53]. While HPC clusters typically write to parallel files systems, latency is involved due to multiple hops to get from application to disk, similar to enterprise data centers. The reason for this is compute nodes must send data to gateway nodes which in turn make RPC-like request across commodity network (e.g. interconnect for storage traffic) to storage nodes. These storage nodes then write to an enterprise storage file system [53].

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 [54]. Canopy as several features similar to WCP. Canopy models trace data as DAGs with nodes representing events in time and edges representing causality. 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. Similarly, we also addressed the level of trace granularity to provide with WCP and chose to focus 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. We use data state as nodes in our DAGs and include a node schema that allows end user to identify where data is being processed, the application it came from, and where it is being sent. Canopy also derives the critical path of its trace data and visualizes the critical path to the end user.

3.3 HPC Performance Measurement Tools

Many tools target application performance of sequential and parallel programs in HPC environments. HPCToolkit is an integrated suite of tools for performance analysis of parallel applications [1]. HPCToolkit use of binary analysis to support both measurement (call stack unwinding of unmodified optimized code) and attribution to loops and in-lined functions allows it to analyze multi-lingual programs that include third-party libraries for which source code and symbol information may not be available.

The Tuning and Analysis Utilities (TAU) parallel performance system is another toolkit that offers profiling and tracing of HPC applications [2]. The TAU framework architecture is organized into three layers – instrumentation, measurement, and analysis – where within each layer multiple modules are available and can be configured in a flexible manner under user control. TAU implements a variety of instrumentation techniques as well.

Many of these tools leverage common techniques and other related tools. For example, ProMon leverages Dyninst to insert monitoring probes—small pieces of code that collects and sends data—into applications [55]. The core technique of Dyninst is called dynamic instrumentation [56]. Associated with the Paradyn project [57], Dyninst provides
users an API to insert instrumented code into applications and modify them during runtime. This allows users to obtain performance profiles of unmodified executables. Another example of tools leverage other performance tools is PerfExpert, which employs HPCToolkit to execute a structure sequence of performance counter measurements [58].

### 3.4 Performance Analysis of Scientific Workflows

Perhaps most related in terms of WCP is work related to the performance analysis of scientific workflows. Herold & Williams introduce top-down performance analysis as an approach to better monitor workflow applications [16]. They implemented a measurement infrastructure for HPC workflow tracing that provides summarized performance metrics for workflow, jobs, and job steps. These summaries guide the user to identify inefficiencies and determine the bottleneck causing job step. This work is similar to ours in that it is characterizing HPC workflow performance outside of a workflow management system. However, the emphasis of our work is on defining workflow critical path and developing a tool that can build a workflow PAG to calculate WCP.
Chapter 4: Architecture

In this chapter, we present Crux—a distributed, end-to-end critical path analysis tool for holistic HPC workflows. Crux allows users to calculate and visualize workflow critical path. Crux takes a novel, data-oriented approach to constructing a program activity graph. Instead of representing vertices as computational events and edges as computational dependencies, Crux uses 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. Crux is composed of a user interface (UI), an application programming interface (API), and a database. In the following sections, we describe how Crux can be deployed in an HPC cluster and instrumented in HPC applications.

4.1 Data State

Crux introduces the concept of data state as a modeling object. A data state describes the data source, format, size, origin, and location. For instance, we can say data state A describes a 10MB, csv file located on disk1 of node1. Any operation made on this data state results in its change is said to be a data mutation (see Figure 4.1.1).

Figure 4.1.1: Crux definitions for data state vertex and data mutation edge
A vertex (circle) in Crux has a size, time, origin, location, and label. Size describes the size of the data. Time is its creation timestamp. Origin is the original application that produced the state. Location is the current storage location. Label is a meaningful descriptor for the data state (e.g. file.csv, byte_stream). An edge
(arrow) in Crux has a cost and mutation. Cost is the elapsed time between two connected data states. Mutation is the operation performed on a data state resulting in a state change.

Crux uses six types of data mutations. We classify these mutations based on common data operations observed in scientific applications. Table 4.1.1 describes the mutation types.

<table>
<thead>
<tr>
<th>Mutation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSFER</td>
<td>Transfer of data between one physical location to another (e.g. staging in data from storage to compute node)</td>
</tr>
<tr>
<td>CONVERT</td>
<td>Conversion of data format or schema (e.g. JSON to CSV)</td>
</tr>
<tr>
<td>APPEND</td>
<td>Appending data to existing data (e.g. adding timestamps to data points in a file)</td>
</tr>
<tr>
<td>SPLIT</td>
<td>Splitting of data into multiple locations (e.g. mpi_scatter())</td>
</tr>
<tr>
<td>MERGE</td>
<td>Merging data from different sources (e.g. mpi_gather())</td>
</tr>
<tr>
<td>DELETE</td>
<td>Permanent deletion of data</td>
</tr>
</tbody>
</table>

Table 4.1.1: Types of data mutations in Crux

Having a schema for data state and data mutation allows Crux to describe a workflow in terms of dataflow and how an input set of data evolves over time. Figure 4.1.2 shows a trivial example of an input file undergoing different data mutations as it moves through a workflow.

**Figure 4.1.2: 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.

### 4.2 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. See Chapter 4.6 for deployment of Crux components in an HPC cluster.

### 4.3 Crux API

The Crux API is an HTTP API that follows a representational state transfer (REST) architecture. REST defines 6 guiding principles [59]:

1. Client-server: Client and server are decoupled and evolve independently. Client only needs to know about a resource’s Uniform resource identifier (URI).
2. Stateless: Server does not store client context between requests. Client manages state of the application.
3. Cacheable: If a response is cacheable, the client cache is given the right to reuse that response data for later, equivalent requests.
4. Layered System: REST allows for a layer system design where server components can run on different physical machines.
5. Code-on-demand: Client can extend functionality by downloading and executing code as needed.

6. Uniform Interface: API resources are identified and manipulated following a common pattern.

An API that adheres to REST principles gains scalability (properties 1, 2, 4), performance (properties 3, 4), simplicity (property 6), portability (property 1), modification (properties 1, 5, 6), and visibility (property 6) [60]. We find these benefits suitable for Crux as a distributed tool capable of scaling to fit client, i.e. HPC application, demands.

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 endpoint such as 

/api/datastates?size={size}&time={time}&location={location}...

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, updating an edge, to something more complicated 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. Figure 4.3.1 shows how the Crux API server will interact with clients and backend graph database.
REST-based web services are typically organized into resources. A resource can be any logical object we want to expose to the user. For example, an API might have users as resources. 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 below.

```
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.

The Crux API has two main resources: states and mutations. State represents a data state vertex and mutation represents a data mutation edge (see Table 4.4.1: The Crux API).

### 4.4 Crux Database

The Crux Database is the backend storage for the Crux API. Its main responsibility is to hold PAG data generated during workflow execution. The database must support concurrent control to manage write operations from multiple API instances. The database must also support a dataset large enough to accommodate collected PAG data. To elaborate, the Crux Database must hold 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.

<table>
<thead>
<tr>
<th>HTTP Method</th>
<th>Path</th>
<th>Request Body</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/states</td>
<td>N/A</td>
<td>Returns a list of data states</td>
</tr>
<tr>
<td>GET</td>
<td>/states/{ID}</td>
<td>N/A</td>
<td>Returns a data state with matching ID</td>
</tr>
<tr>
<td>POST</td>
<td>/states</td>
<td>data state:</td>
<td>Creates a new data state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size number</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>time number</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>origin string</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>location string</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>label string</td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td>/mutations</td>
<td>N/A</td>
<td>Returns a list of data mutations</td>
</tr>
<tr>
<td>POST</td>
<td>/mutations/transfer</td>
<td>start state:</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>origin</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>location</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end state:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>origin</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>location</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Creates a TRANSFER data mutation between a start data state vertex and an end data state vertex.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>POST</th>
<th>/mutations/convert</th>
<th>start state:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>origin</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>location</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>end state:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>origin</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>location</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>Creates a CONVERT data mutation between a start data state vertex and an end data state vertex.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>POST</th>
<th>/mutations/split</th>
<th>start state:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>origin</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>location</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>end states:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end state:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>origin</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>location</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>Creates a SPLIT data mutation between a start data state vertex and ending at all end data state vertices.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>POST</th>
<th>/mutations/merge</th>
<th>start states:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>list</td>
<td></td>
</tr>
<tr>
<td></td>
<td>start state:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>origin</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>location</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>end state:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>origin</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>location</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>Creates a MERGE data mutation between all start data state vertices and ending at an end data state vertex.</td>
<td></td>
</tr>
<tr>
<td>POST</td>
<td>/mutations/append</td>
<td>start state:</td>
</tr>
<tr>
<td>-------</td>
<td>------------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>origin string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>location string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>label string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>origin string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>location string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>label string</td>
</tr>
<tr>
<td>POST</td>
<td>/mutations/delete</td>
<td>start state:</td>
</tr>
<tr>
<td>-------</td>
<td>------------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>origin string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>location string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>label string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>origin string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>location string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>label string</td>
</tr>
<tr>
<td>POST</td>
<td>/wcp</td>
<td>start state:</td>
</tr>
<tr>
<td>-------</td>
<td>------------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>id number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>id number</td>
</tr>
</tbody>
</table>

Table 4.4.1: The Crux API

### 4.5 Crux Critical Path Algorithm

In Crux, the critical path represents the longest path of data state mutations based on execution time. Critical path algorithms are typically shortest path algorithms modified to find the path with the longest execution time [10]. 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. The SSSP problem states that given a directed graph with non-negative weights, find the minimal cost path from one chosen node to all other nodes.

There are distributed implementations of Dijkstra’s algorithm that outperform the original
algorithm. For instance, Meyer and Sanders presented Delta-stepping [61] which divides Dijkstra’s algorithm into phases that can be executed in parallel on distributed memory architectures for an average-case time of $O(\log^2 n/\log \log n)$. For Crux, we use a version of Delta-stepping implemented for shared memory architectures described by Kranjčević et al. [62]. The input of the $\Delta$-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 $\Delta$. 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 [62]. The pseudocode for this algorithm is listed below.
4.6 Deployment in HPC clusters

Figure 4.6.1 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.

```
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,∅)
    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
```
- 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.

- Crux database installed on 1 node.

---

**Figure 4.6.1: 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.
4.7 Instrumentation of HPC Applications

The Crux API is designed as a stateless server. It does not track or store data from clients. Clients of Crux must know when and how to 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.

```java
  // 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' })
```

For Crux to be adopted in a practical manner by HPC applications and workflows, we believe a majority of the instrumentation must be automated.
4.8 Other Instrumentation Approaches

Tracing techniques discussed in Chapter 2.5 show that there are many ways to instrument client programs. We list these instrumentation approaches and describe the pros and cons on clients of Crux (Table 4.8.1).

<table>
<thead>
<tr>
<th>Instrumentation Type</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>Extremely portable</td>
<td>Tedious</td>
</tr>
<tr>
<td></td>
<td>High granularity</td>
<td>Error prone</td>
</tr>
<tr>
<td></td>
<td>Can be automated</td>
<td>May be impacted by compiler</td>
</tr>
<tr>
<td>Compiler-based</td>
<td>Highly automatic</td>
<td>More overhead</td>
</tr>
<tr>
<td></td>
<td>Routine and event granularity</td>
<td>Harder to develop</td>
</tr>
<tr>
<td></td>
<td>Less error prone</td>
<td>Less granular than source code</td>
</tr>
<tr>
<td>Binary</td>
<td>Highly automatic</td>
<td>Less granular than source code</td>
</tr>
<tr>
<td></td>
<td>Proportional overhead</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.8.1: Types of instrumentation approaches with pros and cons

Source code instrumentation is portable and can easily extend to support all compilers for a given language. Furthermore, source code instrumentation provides flexibility to control the instrumentation granularity, for instance a nesting level may be specified to place instrumentation of outer or inner loops [39]. Manually instrumenting a program however is tedious and often error prone. With Crux, the chance of error is higher because clients must pass data following the data state schema of Crux, which is strictly defined. Requiring clients to manually track and manage their data states is not a scalable or practical approach. Compiler-based instrumentation is highly automatic since it uses compilers with instrumentation hooks to add API calls. The benefit of this approach is that instrumentation is abstracted from the developer. Clients do not need to manually insert API calls or managing data state information to pass to Crux. However, further development work would be needed for Crux to instrument using different compilers and interpreters. Finally, binary instrumentation offers the possibility of instrumenting a program during runtime. Similar to compiler-based, the binary approach is also highly
automatic and would abstract instrumentation responsibilities from the developer. Therefore, a compiler-based or binary approach to instrumenting Crux API calls should be considered for production use.
Chapter 5: Crux Prototype

To enable further study of WCP, we implemented a prototype of Crux along with the tooling needed to build and deploy. In addition, we implemented a workflow simulator to create common workflow patterns to conveniently experiment with Crux. Finally, we developed a cloud-based environment to test Crux using Amazon Web Services.

5.1 Crux API

There are many ways to implement an HTTP API. Most programming languages offer built-in library support for creating HTTP clients and servers or have external packages that can do so (for example, Python’s requests library or C’s libcurl library). Backend, or server-side, frameworks such as Django, Express, and Ruby on Rails, are popular for building web applications since they offer high-level mechanisms like creating project files, working with HTTP requests, accessing a database, sanitizing user input, and caching. While there are certain advantages to using a “batteries included” backend framework, Crux’s API does not need them all. For instance, we did not foresee the Crux prototype needing to support authentication for thousands of concurrent users but rather just a handful of developers at a time. We identified the following properties as most important when comparing different backend tools and languages for the Crux API prototype.

1. **Rapid development** – Prototyping is a discovery process and we want a language or framework that lets us develop quickly with minimal code. This allows us to pivot, refactor, or share code snippets easily.

2. **High Performance and Asynchrony** – Crux must potentially handle a large scale of API calls coming from applications in an HPC workflow. In addition, the API
calls are expected to be asynchronous in that clients are not waiting for a return response from the server.

3. **Documentation support** – We want the Crux API to follow best practices in API documentation practices. This makes it easier to develop and consume the Crux API as we iterate on prototype implementations.

With these factors in mind, 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 a framework called FastAPI [63], a framework built around Starlette, which is a lightweight ASGI framework. FastAPI is a fast Python framework that integrates with standards for OpenAPI and JSON schema [63].

Crux’s API schema follows the OpenAPI Specification (OAS) overseen by the OpenAPI Initiative, an open-source collaboration project of the Linux Foundation. The goal of OAS is to define “a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection” [64]. FastAPI integrates OAS support and generates OAS documentation. For example, we can view Crux’s OpenAPI schema in raw JSON format or using a browser.
<table>
<thead>
<tr>
<th>Method</th>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/</td>
<td>Root</td>
</tr>
<tr>
<td>POST</td>
<td>/test</td>
<td>Test</td>
</tr>
<tr>
<td>POST</td>
<td>/graph</td>
<td>Create Graph</td>
</tr>
<tr>
<td>POST</td>
<td>/nodes/{label}</td>
<td>Create Node</td>
</tr>
<tr>
<td>GET</td>
<td>/nodes</td>
<td>Get Node</td>
</tr>
<tr>
<td>GET</td>
<td>/alphav1/nodes/</td>
<td>Get Node Alphav1</td>
</tr>
<tr>
<td>POST</td>
<td>/relationships/transfer</td>
<td>Create Transfer Relationship</td>
</tr>
<tr>
<td>POST</td>
<td>/relationships/convert</td>
<td>Create Convert Relationship</td>
</tr>
<tr>
<td>POST</td>
<td>/relationships/split</td>
<td>Create Split Relationship</td>
</tr>
<tr>
<td>POST</td>
<td>/relationships/merge</td>
<td>Create Merge Relationship</td>
</tr>
<tr>
<td>POST</td>
<td>/relationships/append</td>
<td>Create Append Relationship</td>
</tr>
<tr>
<td>POST</td>
<td>/wcp</td>
<td>Calculate Critical Path</td>
</tr>
<tr>
<td>POST</td>
<td>/test/graph</td>
<td>Test Create Graph</td>
</tr>
</tbody>
</table>

**Figure 5.1.1: Crux OpenAPI Schema in browser, Part 1**
A screenshot of Crux’s OpenAPI documentation describing Crux’s API endpoints.
Figure 5.1.2: Crux OpenAPI Schema in browser, Part 2
A screenshot of Crux’s OpenAPI documentation describing a HTTP POST method on the path “/relationships/merge”. Reaching this endpoint requires the client to supply a request body with the “merge” schema.
Figure 5.1.3: Crux OpenAPI Schema in browser, Part 3
A screenshot of Crux’s OpenAPI documentation showing the schema for “merge”. Merge is a JSON object with a start_state (list of states) and an end_state.
5.2 Crux Database

Crux models a workflow using a program activity graph based on data state. To this purpose, we found a graph database as an appropriate choice for backend storage. Graph databases typically provide a query language making it easier to work with graph-related primitives like vertices and edges. Many graph databases also provide integration with visualization tools or come with client library options to do so. Using a graph database may restrict the schema and therefore types of data we can store. However, we believe it is important for Crux to have easy to use, efficient graph query support.

As previously discussed, there are many choices for backend frameworks; there are also many choices for graph databases. For the Crux prototype, we wanted a solution that was well documented, showed strong concurrency use cases, and came with graph algorithm support optimized for that database. A graph database with a well-documented query language meant we could develop prototypes quickly. Strong concurrency use cases support scalability and scenarios where Crux is deployed with multiple API servers. Furthermore, we wanted a graph database that provided graph algorithms optimized for that particular graph database to ensure highest degree of efficiency.

To this end, we chose Neo4j as our graph database. As the most popular graph database [65], Neo4j comes 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:
To integrate Neo4j into Crux, we use a containerized version of Neo4j. 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.

### 5.3 Crux Critical Path Algorithm

Neo4j provides a variety of graph algorithms that span a variety of use cases like determining the importance of distinct nodes in a network, evaluating how a group is clustered or partitioned, or path finding. Given Crux’s program activity graph, the critical path between the start and end vertex is the path that takes the longest time to complete. 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 [10]. Since Crux represents data state as vertices and data mutations as edges, we measure the elapsed time between data states as the edge weight. To visualize workflow critical path, we choose Neo4j’s algo.shortestPath.stream() routine, which is an implementation of Dijkstra’s shortest path algorithm. The following shows the pseudocode for Dijkstra’s algorithm.

```python
dijkstra(Vertex s)
    Vertex v,w
    Initialize s.dist = 0 and set dist of all other vertices to infinity
    while (there exist unknown vertices, find the one b with the smallest distance)
        b.known = true
        for each a adjacent to b
            if (!a.known)
```
if (b.dist + Cost_ba < a.dist)
    decrease(a.dist to= b.dist + Cost_ba)
a.path = b

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. We then run Neo4j’s shortest path algorithm to visualize the critical path by executing the following Cypher query.

```cypher
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
```

One caveat for using Neo4j’s optimized shortest path algorithms on a Crux graph comes with the case when a MERGE data mutation occurs. 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, the edges receive 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 data state vertex or vertices with the smallest elapsed time to the new data state.
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.

**Figure 5.3.1: 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.

While Neo4j’s algo.shortestPath.stream() routine lets us visualize the critical path in the Neo4j Browser, we do not rely on it for the Crux API to use to calculate WCP.
Instead, we rely on Neo4j’s algo.shortestPath.deltaStepping() routine which implements delta-stepping for shared memory architectures described by Kranjčević et al. [62]. Their implementation of delta-stepping outperforms Dijkstra’s algorithm with an overall complexity of $O(|V|^{1+\frac{1}{d}})$ for graphs where $d$ is the maximum vertex degree [62]. We therefore use this routine with the following Cypher query.

```cypher
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
```

5.4 Crux UI

We leverage the Neo4j Browser for our Crux prototype. The Neo4j Browser is a general-purpose UI that lets users query, visualize, administrate and monitor a Neo4j database. Neo4j Browser also comes with user authentication. Figure 5.4.1 shows a screenshot of the Neo4j Browser displaying a Crux program activity graph. The Crux prototype does not include a custom UI or graph visualizer other than the Neo4j Browser. We discuss the requirements for a future Crux UI in Chapter 7:
The Crux prototype uses the Neo4j Browser. Users can view a workflow PAG being constructed during runtime. The Neo4j Browser also lets users submit Cypher queries against the graph database.

5.5 Crux Workflow Simulator

We design a system of distributed applications to simulate a scientific workflow. The goal of this simulator system is to provide developers with a template to create custom workflows, instrument Crux API calls, and experiment Crux features. The simulator system also serves as a lightweight, local testbed to examine Crux’s performance in lieu of using HPC applications running on production HPC clusters. A diagram of the Crux Workflow Simulator system is shown in Figure 5.5.1.
Figure 5.5.1: Crux Workflow Simulator deployed with Crux

Crux Workflow Simulator components (green) shown with Crux core components (blue) running on the same host. Workflow Simulator components are implemented as standalone applications. 1) 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. 2) 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. 3) The Crux API server, implemented in Python, sends Cypher queries via Bolt protocol at bolt://graphdatabase.crux:7687. 4) Web browser opens the Neo4j Browser at http://localhost:7474/browser/

The simulator system is made up of individual applications intended to simulate a certain aspect of an HPC workflow. For instance, one application could simulate a data preprocess stage that involves staging in data from a storage to compute node and performing some preprocessing on that dataset. Another application could simulate an MPI job that splits the dataset to a number of MPI ranks. To make the simulators extensible, we create a common configuration file from which each simulator loads. The configuration 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).

```plaintext
// 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
```

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.
5.6 Local deployment of Crux using Docker

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 see contents assigned to it. Docker is a set of tools for building and deploying containers. A Docker image is a read-only template (defined by a YAML file called a Dockerfile) that is used to build containers. A Docker container is a running instance of a Docker image. Docker also comes with a tool called docker-compose that allows users to conveniently define and run multiple Docker containers at once. 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 Chapter 5.7). Figure 5.6.1 shows an example deployment of Crux using Docker on a local system.
5.6.1 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.

5.7 Cloud deployment of Crux using AWS

In order to test the distributed and scaling properties of Crux, we chose Amazon Web Services (AWS) as platform for cloud deployment. AWS offers a host of services to deploy, manage, host, and scale web applications. One of these services, called Elastic Container Service (ECS), allows users a way to deploy Docker containers without manually provisioning underlying Elastic Cloud Compute (EC2) instances, i.e. VMs. AWS ECS handles Docker installation as well as Docker container orchestration. Running one container on ECS using the AWS console is somewhat trivial since users just follow console prompts to do so. For example, users get prompted to specify what container image they want to deploy, called a Task Definition. A Task represents a running container in ECS. Users then create Services which tell ECS how many instances of a Task Definition to run in an ECS cluster. Services will handle launching Tasks again should any of them
fail. Services also allow users to configure a load balancer to distribute traffic across all Tasks related with the Service. Figure 5.7.1 summaries these main ECS-related terminologies.

![ECS Cluster Diagram]

**Figure 5.7.1: AWS Elastic Cloud Service Terms**
An ECS cluster (white) is made up of ECS container images (orange). Container instances are EC2 instances provisioned by ECS to run Docker images. ECS service (blue) defines long running ECS Tasks (green) that use the same Task Definition. Tasks are running Docker containers defined by a Task Definition.

**Terraform**

While the AWS console offers a convenient UI to deploying containers on ECS, it does not allow us to programmatically define infrastructure details or automate deployment. For example, let’s say we commit a small change to the Crux API, build a new Docker image, and push that new image to ECR (AWS Elastic Container Registry). Using the AWS console, we would have to manually stop and delete the Crux API Task and Service and then redeploy a whole new Task and Service for the API image again. A way around this manual process is to use a tool that enables you to write infrastructure-as-code. Amazon’s solution is called CloudFormation. For Crux, we choose an open source tool called Terraform to provision and manage our ECS environment. Terraform is a cloud-agnostic tool that uses HCL (Hashicorp Configuration Language) rather than YAML to define configuration. Terraform also separates the planning phase from the execution
phase, letting developers see changes being proposed before they occur. Being cloud agnostic, our implementation of ECS infrastructure-as-code using Terraform also gives us flexibility in the future if we wish to deploy Crux on a different cloud platform.

_Service Discovery & Scaling_

When an application made up of multiple independent services get deployed to the cloud, its services must have a way of finding each other. For example, if service A running on one VM needs to send an HTTP request to service B running on a different VM, how does service A find the IP address and port number belonging to service B? Furthermore, let’s say we have multiple instances of service B. How does service A reach a specific instance of service B? This common problem is known as service discovery and there many known solutions. For Crux’s deployment on AWS, we use a service registry approach for mapping service names to IP addresses (e.g. apiserver.local to 10.0.0.2). We leverage AWS’s DNS service, called Route 53, which provides a service registry for creating A and SRV records that map to container IPs.

---

![Figure 5.7.2: Example entries in the service registry used to identify Crux components](image)

---

52
For load balancing, we leveraged a networking solution offered by AWS called an Application Load Balancer (ALB). The ALB functions at the application layer. We configure a listener rule that tells the load balancer to check for connection requests using a protocol and port. The load balancer then route requests to specified targets using a default policy of round robin. For Crux, we specify our target groups as instances of the API and Neo4j. The AWS ALB also provides us with a DNS name accessible outside of our virtual private cloud.

Figure 5.7.3: Configuring an AWS Application Load Balancer to route traffic to Crux components
We configure the ALB to route HTTP traffic on port 7474 to target group graphdatabase. This target group contains an instance of the Crux graph database container. We also configure routing of HTTP traffic on port 8000 to target group apiserver. This target group contains all instances of the Crux API server container.

Figure 5.7.4: Example of target group apiserver
Target group apiserver is shown here with 2 instances of the Crux API server running at IP address 10.0.197 and 10.0.1.235.

Figure 5.7.5: Networking of Crux on AWS

The AWS Internet Gateway (green circle) enables communication between instances in our VPC and the internet. The Application Load Balancer (yellow) is configured to route HTTP traffic on port 8000 to target group apiserver and HTTP traffic on port 7474 to target group graphdatabase. Target group apiserver consists of Crux API server instances (blue left) running on EC2 instances (orange). Target group graphdatabase is consists of one Neo4j instance (blue, labeled crux-graphdatabase). AWS ECS Service Discovery and Route 53 Registry Service provide DNS name resolution to container IPs. For example, crux-api instances can be reached at http://apiserver.crux:8000 and the Neo4j Browser can be reached at http://graphdatabase.crux:7474.
Chapter 6: Experiments

In this chapter, we perform experiments to demonstrate the capabilities and performance of the Crux prototype. First, we configure the Crux workflow simulator to generate and execute a variety of characteristic workflows. The workflow simulator is deployed locally and makes API calls to a Crux instance on AWS. Next, we conduct a series of load tests against our Python API framework and measure its performance relative to other framework candidates. Finally, we conduct a series of experiments to measure the performance of Crux running behind a load balancer on AWS. Table 5.7.1 specifies the tools and systems used in these experiments.

<table>
<thead>
<tr>
<th>Tools</th>
<th>Specifications / Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docker</td>
<td>19.03.5</td>
</tr>
<tr>
<td>Docker-compose</td>
<td>docker-compose version 1.24.1</td>
</tr>
<tr>
<td>Terraform</td>
<td>v0.12.5</td>
</tr>
<tr>
<td>Python</td>
<td>3.7.6</td>
</tr>
<tr>
<td>Neo4j</td>
<td>3.5</td>
</tr>
<tr>
<td>Neo4j Graph Algorithm Library</td>
<td>3.5.4.0</td>
</tr>
<tr>
<td>AWS ECS Agent</td>
<td>1.32.0</td>
</tr>
<tr>
<td>AWS EC2 Image</td>
<td>Amazon Linux AMI 2018.03.y x86_64 ECS HVM GP2 t2.medium (2 vCPU, 4GB RAM)</td>
</tr>
<tr>
<td>Local System</td>
<td>macOS Mojave (10.14.6) 2.9 GHz Intel Core i9 (12 core) 32 GB 2400 MHz DDR4</td>
</tr>
</tbody>
</table>

Table 5.7.1: Tools and their specifications used in Crux experiments

6.1 Capability

In this section, we present 5 workflow cases that demonstrate the effectiveness of Crux. In each case, we configure Crux’s workflow simulator to generate a workflow that exhibits a characteristic element or pattern we have observed in HPC workflows and intend to test. We deploy the workflow simulator locally using a tool called Docker-Compose.
which launches all simulator components as containers. These containers are accessible at the following endpoints.

```python
SIMULATOR_URLS = [
    'http://simulatorstagein.crux:8001',
    'http://simulatorpreprocess.crux:8002',
    'http://simulatormpi.crux:8003',
    'http://simulatorpostprocess.crux:8004',
    'http://simulatorviz.crux:8005',
    'http://simulatorcheckpointout.crux:8006',
    'http://simulatorcheckpointin.crux:8007',
    'http://simulatorload.crux:8008',
    'http://simulatorfilecreate.crux:8009',
]
```

The simulator containers make API calls to a remote deployment of Crux on AWS described in Section 5.7. Crux on AWS has a public endpoint where simulators can reach the API server at demo-156045408.us-west-2.elb.amazonaws.com/8000. 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 collect screenshots of the visualized workflow PAG and workflow critical path by logging onto Crux’s Neo4j Browser available at demo-156045408.us-west-2.elb.amazonaws.com:7474 and executing the following Cypher queries against the Neo4j database.

```cypher
// Visualizes the workflow PAG
MATCH (n) return n;

// Visualizes the workflow critical path
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;
```

**Case 1: Generic Workflow**

<table>
<thead>
<tr>
<th>Objective</th>
<th>Simulate workflow with generic jobs as described by APEX [46] from Figure 3.1.1. These jobs include staging in data, preprocessing data, MPI, postprocessing data, and visualizing data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivation</td>
<td>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.</td>
</tr>
</tbody>
</table>
| Configuration | SIMULATORS = {  
    'Stagein': 'http://simulatorstagein.crux:8001',  
    'Preprocess': 'http://simulatorpreprocess.crux:8002',  
} |
Result

Critical Path = ['uuid': '75dedfd1-3fc8-42df-897d-3b4e4c2ddf65',
'uuid': '21a6462f-8676-4985-9af6-d52516d0e4a0',
'uuid': 'b04bd907-89c8-440c-a968-27f878c860f3',
'uuid': 'f64a6412-b4d6-4e78-9032-82759b919f11',
'uuid': '196b7ee2-ae0a-4db3-91de-007f33803f35',
'uuid': 'e08fe039-0fea-4d75-9f67-2c45361a5cb',
'uuid': '9ff6023e-c4f8-4820-87b3-e927538fbaea',
'uuid': 'e335d51c-197e-48be-8381-3e60312a699d']
Cost = 9.492
Total elapsed time = 6.584s

Figure 6.1.1: Generic workflow PAG
Top image shows 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 enlarged MPI portion of PAG.

Figure 6.1.2: Generic workflow critical path.
The critical path in this execution is the path with longest elapsed time through the MPI job. Cost=9.492

**Case 2: Multiple data splits with MPI**

<table>
<thead>
<tr>
<th>Objective</th>
<th>Simulate 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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivation</td>
<td>Heavy scientific computation typically involves distributing data across compute nodes, performing the computations, and merging the results.</td>
</tr>
</tbody>
</table>
| Configuration | SIMULATORS = {  
'Stagein': 'http://simulatorstagein.crux:8001',  
'Preprocess': 'http://simulatorpreprocess.crux:8002',  
'MPI': 'http://simulatormpi.crux:8003',  
'Postprocess': 'http://simulatorpostprocess.crux:8004',  
'Viz': 'http://simulatorviz.crux:8005',  
}  
TOTAL_DATA_SPLITS = 3  
TOTAL_MPI_RANKS = 150 |
| Result | Critical Path = [  
'uuid': '41101c0c-fbf8-420e-8058-0a438a7d3f64',  
'uuid': '75de336c-3bc2-458e-a04d-a66dd9debbf4',  
'uuid': '5c8712b3-ccad-4698-8029-3348f09c3c3c',  
'uuid': '99b0a99-ad24-41f2-89c7-720a52f0ca03',  
'uuid': 'aa6d44f7-830d-497a-a796-590349e3ee9',  
'uuid': 'ceef28303-d31b-458e-8584-88f3c7cd4d46',  
'uuid': '47033880-13d8-4ab8-b4f8-b542a323d3bd',  
'uuid': '391283a6-a4bb-401e-a056-4fd913e47827',  
'uuid': '2f48eeb8-d8c7-4040-ad7b-cb34d7e9171f',  
'uuid': 'b3a06bf3-99a6-4bf9-b8aa-ac01e8832af4',  
'uuid': '448fdaaf4-4280-42bd-b26c-ee2e63558605'  
]  
Cost = 22.159  
Total elapsed time = 57.590s |
Figure 6.1.3: Crux PAG showing multiple data splits with 150 MPI ranks
Top image shows entire PAG with 5 jobs: stage in (purple), preprocess (blue), MPI (yellow), postprocess (teal), and visualization (grey). All Crux PAGs begin with a null vertex (red) created during database initialization. Bottom image shows enlarged MPI portion of PAG.
**Figure 6.1.4: Workflow critical path for PAG showing multiple data splits with 150 MPI ranks**

The critical path in this execution is the path with longest elapsed time through the MPI job. Cost=22.159

---

**Case 3: Multiple scientific runs with checkpoint files**

<table>
<thead>
<tr>
<th>Objective</th>
<th>Simulate 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 should be configurable so that we can intentionally set it as the critical path for testing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivation</td>
<td>Workflows involving a scientific simulator as described in APEX [46] from Figure 3.1.1 often include multiple runs of that simulator with checkpoint files created in between. These files can be kept for the duration of the workflow or persist between workflow runs in storage. We simulate checkpoint files being written to storage between runs of parallel tasks representing the scientific simulator.</td>
</tr>
</tbody>
</table>
| Configuration| SIMULATORS = {
    'Stagein': 'http://simulatorstagein.crux:8001',
    'Preprocess': 'http://simulatorpreprocess.crux:8002',
    'MPI': 'http://simulatormpi.crux:8003',
    'Checkpointout': 'http://simulatorcheckpointout.crux:8006',
    'Checkpointin': 'http://simulatorcheckpointin.crux:8007',
    'MPI2': 'http://simulatormpi.crux:8003',
    'Postprocess2': 'http://simulatorpostprocess.crux:8004',
    'Viz': 'http://simulatorviz.crux:8005',
}

CHECKPOINT_FILE_TRANSFER_TIME = 2

| Result | Critical Path = ['uuid': 'c8c2a97d-bbdf-4ccc-8ec2-757ceed3782f', 'uuid': '35f18ae2-bfe2-4b0c-a11c-0160608a785f', 'uuid': '926e2344-95ca-4ed6-b3ff-767bf1264849', 'uuid': 'c383f3d1-c238-47fa-9877-f180e09d65a5', 'uuid': '1afadb84-43d9-47fd-9917-280dca2840f6', 'uuid': '9963b882-3f6f-41b2-803f-cd9a55cd8364', 'uuid': '44aab3ef-b510-4afe-942f-c3bc80865fc7', 'uuid': 'c6189b74-8cad-4a44-ae11-fe3be9dbf35', 'uuid': 'f17622e7-8387-4777-a9b7-e50ccc5f8e0', 'uuid': 'd39bc486-d352-43f3-b458-21d4324a8c6', 'uuid': '2e3d605e-8841-4391-8e88-218838d191ad', 'uuid': 'eea86e5e-065e-4a51-8615-934e6553598e', 'uuid': 'd80c2023-26c8-4e18-8011-9ffadfe4ae15', 'uuid': 'eea86e5e-065e-4a51-8615-934e6553598e', 'uuid': 'd80c2023-26c8-4e18-8011-9ffadfe4ae15',] |

---
Figure 6.1.5: Crux PAG showing checkpoint files
Top image shows entire PAG with 8 jobs: stage in (purple), preprocess (blue), 2 MPI jobs (yellow), checkin and checkout file (grey, center), postprocess (teal), and visualization (grey, right). All Crux PAGs begin with a null vertex (red) created during database initialization. Bottom image shows enlarged checkout and checkin portion of PAG. Highlighted is the TRANSFER data mutation which we configured to take 2 seconds as to make it the critical path for testing.
Figure 6.1.6: Workflow critical path of PAG showing checkpoint files
The critical path in this execution is the path involving the checkout and checkout data vertices (grey, center). Cost = 74.854. Note the Neo4j Browser inaccurately includes the edge (highlighted yellow) from a previous MPI data state vertex (yellow) and the checkin file data state vertex (gray, center).

Case 4: Multiple data sources

| Objective | Simulate workflow that involves loading more than one source of data. The loading should take place between workflow jobs. |
| Motivation | Loading additional sources of data between jobs is another common workflow pattern. |
| Configuration | SIMULATORS = {
  'Stagein': 'http://simulatorstagein.crux:8001',
  'Preprocess': 'http://simulatorpreprocess.crux:8002',
  'MPI': 'http://simulatormpi.crux:8003',
  'Postprocess': 'http://simulatorpostprocess.crux:8004',
  'Load': 'http://simulatorload.crux:8008',
  'MPI2': 'http://simulatormpi.crux:8003',
  'Viz': 'http://simulatorviz.crux:8005'
} |
| ADDITIONAL_DATA_SOURCE_COUNT = 1 |
| Result | Critical Path = [
  'uuid': 'd794b404-9d67-4aac-b3ab-a77710617bdd',
  'uuid': '6e75a7d1-2f0a-4c4c-9ce3-c6f7-7be89bf8',
  'uuid': '78a59bc4-03b6-4ac9-a4f6-b412a24b0533',
  'uuid': '9c63bf13-c6f7-4941-bbaa-4b4d9b05df',
  'uuid': 'e059d2ee-869d-4b0f-ab01-dd994889e28b',
  'uuid': '3c5a47bb-96a7-4939-af98-5b4df57241bc',
  'uuid': 'd1363e36-20ad-4245-b89-7057fa3871de',
  'uuid': '7de5551-62ba-4879-81a4-cc7f5c6bca6',
  'uuid': '5ce0ac3de-ec2d-4e7e-99df-b7c601651377',
  'uuid': 'cdaddb22-48ed-46c6-aa5c-c53a6c384f16',
  'uuid': '85cc42db-c71d-4986-8d1d-1c5e0949da5b',
  'uuid': '30eca3eb-7efa-4d81-873a-91bf02450070'] |
Cost = 67.242
Total elapsed time = 6.5413s
Figure 6.1.7: Crux PAG showing two input data sources
Top image shows entire PAG with 8 jobs: stage in (purple and grey, left), preprocess (blue), 2 MPI jobs (yellow), postprocess (teal), load (grey, center) and visualization (grey, right). All Crux PAGs begin with a null vertex (red) created during database initialization. Middle image shows enlarged stage in job where a TRANSFER data mutation represents the loading of two input data states. Bottom image shows enlarged portion where a MERGE data mutation merges the postprocessed output of a first MPI job (teal) with the second data source (grey left) to create a new data state (grey right).

Figure 6.1.8: Workflow critical path of Crux PAG showing two input data sources
The critical path in this execution is the path involving first and second MPI job. Cost= 67.241.
Case 5: File creation and deletion

<table>
<thead>
<tr>
<th>Objective</th>
<th>Simulate workflow that involves creating temporary files and deleting them between runs of a scientific simulator.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivation</td>
<td>This pattern of file creation and deletion between runs of a scientific simulator was observed in DroughtHPC (Section 1.1.1 Case: DroughtHPC and VIC)</td>
</tr>
<tr>
<td>Configuration</td>
<td>SIMULATORS = {</td>
</tr>
<tr>
<td></td>
<td>'Stagein': '<a href="http://simulatorstagein.crux:8001">http://simulatorstagein.crux:8001</a>',</td>
</tr>
<tr>
<td></td>
<td>'Preprocess': '<a href="http://simulatorpreprocess.crux:8002">http://simulatorpreprocess.crux:8002</a>',</td>
</tr>
<tr>
<td></td>
<td>'MPI': '<a href="http://simulatormpi.crux:8003">http://simulatormpi.crux:8003</a>',</td>
</tr>
<tr>
<td></td>
<td>'Filecreate': '<a href="http://simulatorfilecreate.crux:8009">http://simulatorfilecreate.crux:8009</a>',</td>
</tr>
<tr>
<td></td>
<td>'Postprocess': '<a href="http://simulatorpostprocess.crux:8004">http://simulatorpostprocess.crux:8004</a>',</td>
</tr>
<tr>
<td></td>
<td>'MPI2': '<a href="http://simulatormpi.crux:8003">http://simulatormpi.crux:8003</a>',</td>
</tr>
<tr>
<td></td>
<td>'Viz': '<a href="http://simulatorviz.crux:8005">http://simulatorviz.crux:8005</a>',</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Result</td>
<td>Critical Path = [</td>
</tr>
<tr>
<td></td>
<td>'uuid': 'bd753f1c-f3a8-4e97-9b44-92ac44293049',</td>
</tr>
<tr>
<td></td>
<td>'uuid': '8667f1dc-0e0a-43cd-8a06-0d374f58136f',</td>
</tr>
<tr>
<td></td>
<td>'uuid': '349db87d-f737-44f5-9d95-a922b02eb2c2',</td>
</tr>
<tr>
<td></td>
<td>'uuid': '2f6f3eeb-06d1-467b-9b4b-4e19b9b1a76d',</td>
</tr>
<tr>
<td></td>
<td>'uuid': '65dfca3d-336e-4e70-aba4-2d0d7df8faec',</td>
</tr>
<tr>
<td></td>
<td>'uuid': '8b186061-6a0c-4feb-a5d1-1e5e85eba3adb',</td>
</tr>
<tr>
<td></td>
<td>'uuid': 'a616419e-937e-4e7f-8ef4-b01ad62edff7',</td>
</tr>
<tr>
<td></td>
<td>'uuid': '42b035f0-22e6-46d2-95f3-6dba5e3c690f',</td>
</tr>
<tr>
<td></td>
<td>'uuid': '6a102ca8-8689-48a1-8978-4de1ebd23a51',</td>
</tr>
<tr>
<td></td>
<td>'uuid': '9695b526-0ce9-43b0-ae18-eb093193ac5d',</td>
</tr>
<tr>
<td></td>
<td>'uuid': '7ecc3437-94d3-45a2-81dc-39a3e7a333bf',</td>
</tr>
<tr>
<td></td>
<td>'uuid': '00e033be-08c7-47c8-837f-4d2ce8f7e91b'</td>
</tr>
<tr>
<td></td>
<td>]</td>
</tr>
<tr>
<td></td>
<td>Cost = 74.868</td>
</tr>
<tr>
<td></td>
<td>Total elapsed time = 7.2561s</td>
</tr>
</tbody>
</table>
Figure 6.1.9: Crux PAG showing file creation and deletion
Top image shows entire PAG with 8 jobs: stage in (purple), preprocess (blue), 2 MPI jobs (yellow), file creation and deletion (grey, center), postprocess (blue), and visualization (grey, right). All Crux PAGs begin with a null vertex (red) created during database initialization. Middle image shows enlarged stage in job where a TRANSFER data mutation represents the loading of two input data states. Bottom image shows enlarged portion where a MERGE data mutation merges the postprocessed output of a first MPI job (teal) with the second data source (grey left) to create a new data state (grey right).
The critical path in this execution is the path involving first and second MPI job with the file creation between them. Cost= 74.868.

6.2 Performance

In this section we perform experiments to characterize the performance of the Crux prototype. The purpose of experiments in this section is to characterize the performance of Crux.

API Load Testing Comparison

Web API servers are often evaluated using a technique called API load testing. The purpose of API load testing is to evaluate how many requests an API can handle before failing. There are many load testing tools available to choose from. For Crux, we used a tool called Vegeta [66] written in Go. Vegeta’s CLI allowed us to easily control over variables of interesting such as request rate, concurrency, and time. We compare Crux’s underlying Python API framework against two others: Flask and Go. Flask is a popular
micro web framework written in Python. We implement a simple HTTP server using Flask as a “baseline” Python framework to test against our FastAPI Python framework. We implement a second HTTP server using Go’s HTTP library. As a compiled language developed for web services, we expect Go to have the better performance compared to either Python frameworks. First, we used Vegeta to measure the mean latency of all 3 servers deployed locally running 50 requests per second for 5 seconds (Figure 6.2.1). Second, we deployed the same 3 servers in individual containers and measured the latency running 50 requests per second for 5 seconds (Figure 6.2.2). Lastly, we measured each HTTP server’s request rate with various numbers of max workers over 5 seconds (Figure 6.2.3). Max workers specifies the maximum number of concurrent workers able to make a request.

![Mean latency of API Servers at 50 requests per second for 5 seconds](Image)

**Figure 6.2.1: Mean latency of HTTP servers at 50 requests per second**
Comparing HTTP servers written using Flask, FastAPI, and Go deployed locally on port 8081. Vegeta command used: echo "GET http://localhost:8081" | vegeta attack -duration=5s | tee results.bin | vegeta report
Figure 6.2.2: Mean latency of containerized HTTP servers at 50 requests per second
Comparing HTTP servers written using Flask, FastAPI, and Go deployed in Docker containers locally exposed on port 8081. The Docker Crux FastAPI is the actual Crux prototype. Vegeta command used: echo "GET http://localhost:8081" | vegeta attack -duration=5s | tee results.bin | vegeta report

<table>
<thead>
<tr>
<th>Max Workers</th>
<th>Flask</th>
<th>FastAPI</th>
<th>Go</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1388.32</td>
<td>3902.70</td>
<td>8786.40</td>
</tr>
<tr>
<td>10</td>
<td>1461.61</td>
<td>5872.98</td>
<td>29531.37</td>
</tr>
<tr>
<td>100</td>
<td>1456.30</td>
<td>6175.92</td>
<td>27950.27</td>
</tr>
<tr>
<td>200</td>
<td>1589.96</td>
<td>6090.79</td>
<td>28523.28</td>
</tr>
<tr>
<td>500</td>
<td>49810.04</td>
<td>59108.39</td>
<td>37269.73</td>
</tr>
</tbody>
</table>

Success rate at 500 workers:
- Flask: 1.74%
- FastAPI: 2.68%
- Go: 86.00%

Figure 6.2.3: Request rate of HTTP servers against number of max workers
Request rate is the number of requests per second. Max workers specifies the maximum number of concurrent workers able to send a request during a test. Request rates with less than 100% responses with success codes are indicated in red. FastAPI outperformed Flask at all levels of workers but both failed to have 100% responses with success codes at 200 max workers. Go outperformed both Python frameworks and sustained 100% responses with success codes up to 200 max workers. Vegeta command used: echo "GET http://localhost:8081/" | vegeta attack -max-workers 100 -rate -0 -duration 5s | tee results.bin | vegeta report

**Performance of local and remote deployments of Crux**

We measure the time required to create data state vertices on local and remote deployments of Crux. We create and run a local Python script that makes parallel HTTP requests to a local instance of Crux. We measure the time for the requests to reach Crux’s /node endpoint to create 100, 1000, and 10,000 data state vertices. We perform the same measurement for requests against Crux deployed on AWS (Figure 6.2.4). Next, we scale out the instances of Crux’s API server to 1, 2, and 3 instances. We run the same Python script and perform the same time measurement (Figure 6.2.5).

![Time to add vertices to local and remote deployment of Crux](image)

**Figure 6.2.4: 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 \( (R^2=1) \).

![Figure 6.2.5: Time to add vertices against number of Crux API instances](image)

**Figure 6.2.5: 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 = 1.3792s). 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 \( (R^2=1) \).

**AWS Application Load Balancer Performance**

We observe the performance of the AWS Application Load Balancer (ALB) used in remote deployment of Crux. The ALB is responsible for distributing traffic to Crux API server instances in round robin fashion. For this experiment, we executed the following steps:

1. Create a local python script that makes \( N \) number of POST requests to the Crux API test endpoint which will create a data state vertex in Crux’s graph database.
2. Launch 3 instances of the Crux API server on AWS registered to the ALB. Run
the python script with N=10000.

3. Scale the API server down to 1 instance. Run python script with N=10000.

4. Run python script with N=1000

5. Scale the API server up to 4 instances. Run python script with N=10000.

6. Observe the following metrics: requests (Figure 6.2.6), target response time
   (Figure 6.2.7), request count per target (Figure 6.2.8), and utilization of CPU and
   memory (Figure 6.2.9).

![Figure 6.2.6: Request count to Crux measured by AWS ALB](app/demo/11e05b35fe1c2d52)

Shows the request count from launching 10000 requests to 3 API server instances (21:15 to 21:20), 10000 requests to 1 API server instance (21:35 to 21:40), 1000 requests to 1 API server instance (21:45 to 21:50), and finally 10000 requests to 4 API server instances (22:05 to 22:10).
**Figure 6.2.7: Target response time of Crux measured by AWS ALB**

Shows the target response time of launching 10000 requests to 3 API server instances (21:15 to 21:20), 10000 requests to 1 API server instance (21:35 to 21:40), 1000 requests to 1 API server instance (21:45 to 21:50), and finally 10000 requests to 4 API server instances (22:05 to 22:10). Target response in all cases did not exceed over 35ms.

**Figure 6.2.8: Request count per target measured by AWS ALB**

The request count per target is the proportion of response to a single Crux API server instance. We use this metric to ensure that the load balancer is distributing traffic to all Crux API server instances. We can see that from 21:30 to 21:45, only one API instance was used since the request count per target peaks at 100 percent. At 21:15, we can tell 3 API instances were used since the request count per target peaks at 0.33 percent. Finally, the request count per target after 22:00 peaks at 0.25 which corresponds to 4 API instances used.
Figure 6.2.9: CPU and memory utilization for API server and graph database instances on AWS
Shows CPU and memory utilization from launching 10000 requests to 3 API server instances (21:15 to 21:20), 10000 requests to 1 API server instance (21:35 to 21:40), 1000 requests to 1 API server instance (21:45 to 21:50), and finally 10000 requests to 4 API server instances (22:05 to 22:10).
Chapter 7: Discussion

In this chapter, we discuss the experimental results of the Crux prototype. We discuss factors that contribute to the overhead of Crux and identify future areas of work for Crux.

7.1 Capability and Performance

The main goal of our experiments was to demonstrate the workflow patterns that Crux could solve workflow critical path for and to provide evidence supporting the scale at which Crux can reasonably operate at. We also wanted to observe the capability of Neo4j and the Neo4j Browser. While the 5 cases we demonstrated using the workflow simulator does not cover every possible workflow pattern, we believe it covers key patterns from workflows we reviewed such as described in APEX [46] and DroughtHPC [11]. For instance, cases 1, 2, and 3 show common jobs like MPI and checkpoint file creating. Case 4 and 5 showed multiple data sources and file creation and deletion. While it would be possible to create longer workflows with more vertices (for instances we use Crux API calls to create 10,000 vertices in several of the performance experiments), we kept runtimes short with modest total vertices in order to better provide screenshots of the entire PAG and workflow critical path.

We observed limitations with the capability of Neo4j Browser for Crux. One of the most obvious limitations was the inability of Neo4j Browser to highlight a path within a graph. This would be a helpful feature for users to be able to view workflow critical path in the context of a fully visualized Crux PAG. Another limitation was demonstrated in Case 5 where we simulated checkpoint files. While the Neo4j SSSP algorithm correctly returned
the cost of the critical path, the path visualized by Neo4j Browser incorrect displayed an edge did not belong in the critical path (Figure 6.1.6). To verify that this was the browser’s limitation, we ran the same workflow but with an added null vertex to the non-critical path (between the merged MPI job vertex and checkpoint out vertex). This resulted in Neo4j Browser displaying the correct critical path edges. Viewing large number of vertices during our performance experiments was challenging for the Neo4j Browser. In both cases of Crux deployed locally and remotely on AWS, we observe the browser lagging when attempting to view 2000+ vertices. At 10,000 vertices, the browser was not practically usable (Figure 7.1.1). Lastly, Neo4j Browser did not offer ways for us to visualize results from previous workflow runs or easily export PAG data from the UI.

![Figure 7.1.1: Neo4j Browser attempting to display 10000 data state vertices](image)

### 7.2 Overhead

Like other performance tools, the main overhead of Crux will involve the number of instrumented API calls required to create a full program activity graph. However, Crux receives API calls in the form of HTTP requests to the Crux API server running remotely. Our performance experiments suggest that network proximity of Crux to application clients...
improves Crux’s performance. This is consistent with our original expectations when we described Crux’s architecture (Chapter 4.6 ). Requests that must travel from the client through the internet to reach Crux will take longer than requests that can reach Crux on the same network. More 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 the load balancer responsible for distributing traffic to the API instances. If the load balancer can only process a limited number of requests per second, adding more API instances will not improve Crux’s performance. Another limiting factor could be the performance of Crux’s database. Like most popular databases, Neo4j implements concurrency control to avoid read-write conflicts. However, having multiple API instances would have diminishing returns if Crux’s Neo4j instance is unable to process more requests than it receives. Lastly, we consider the system overhead required to deploy and run Crux. 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.

7.3 Instrumenting applications with Crux API calls

After developing the Crux prototype and workflow simulator, we noticed the need to develop rules or guidelines for mapping Crux API calls to actual application routines. We started identifying API-related events in our workflow simulator applications and
describing their implications for HPC applications. We provide a few examples in Table 7.3.1.

<table>
<thead>
<tr>
<th>Workflow Simulator Application</th>
<th>HPC Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relies on simulator manager to track previously created data states</td>
<td>Must have knowledge of previously created data state and query for its reference using Crux API</td>
</tr>
<tr>
<td>Uses configured values for data state parameters when calling API to create new data state</td>
<td>Must compute data state parameters before calling API to create new data state</td>
</tr>
<tr>
<td>Assumes high-level granularity and no intermediary states when creating new data states. For example, writing to a new file was just represented as an APPEND data mutation from a previous data state to a new data state.</td>
<td>Granularity level may depend on programming language. For instance, writing to a new file may first involving creating an empty file and then writing to that file line by line. Number of data states created can thus vary.</td>
</tr>
<tr>
<td>Access to common routines for preparing JSON data for API calls.</td>
<td>Application is responsible for preparing data accurately in API calls</td>
</tr>
</tbody>
</table>

Table 7.3.1: Crux API events in workflow simulator applications and implication for HPC applications

To better illustrate these examples, 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 (Figure 7.3.1 and Figure 7.3.2). In the Python application, we included our custom Python library called crux to access utility functions that help create and manage Crux data states. We do the similar in C but also include utility functions that wrap around the libcurl library to help execute HTTP requests. Lastly, we show the resulting Crux PAG in Neo4j after running the Python and C applications (Figure 7.3.3).
```python
if __name__ == '__main__':
    # Client must establish connection to Crux and receive starting vertex
    start_vertex = crux.connect()
    f = open('input.txt', 'r')
data = f.read()

    # API call to create new data state for loading a input file into memory
    input_vertex = crux.new_data_state(location='input.txt')
    requests.post(crux.DATA_VERTEX_ENDPOINT + '/myapp', json=input_vertex)
    requests.post(crux.DATA_VERTEX_ENDPOINT + '/transfer', json=crux.new_transfer_bilateral(start_vertex, input_vertex))
    f.close()
    processed_data = sample_computation(data)

    # API call to create new data state to show data converted after computation in memory
    new_data_vertex = crux.new_data_state(size=len(processed_data))
    requests.post(crux.DATA_VERTEX_ENDPOINT + '/myapp', json=new_data_vertex)
    requests.post(crux.DATA_VERTEX_ENDPOINT + '/convert', json=crux.new_convert_bilateral(input_vertex, new_data_vertex))
    f = open('output.txt', 'w')
f.write(processed_data)

    # API call to create new data state for writing to new file, output.txt
    output_vertex = crux.new_data_state(location='output.txt')
    requests.post(crux.DATA_VERTEX_ENDPOINT + '/myapp', json=output_vertex)
    requests.post(crux.DATA_VERTEX_ENDPOINT + '/transfer', json=crux.new_transfer_bilateral(new_data_vertex, output_vertex))
    f.close()
```

Figure 7.3.1: Python application instrumented with 6 Crux API calls
Crux API calls inserted on lines 34, 35, 43, 44, 52, and 53.
Figure 7.3.2: C application instrumented with 6 Crux API calls

Crux API calls inserted on lines 43, 44, 51, 52, 59 and 60
Figure 7.3.3: Crux PAG from Python and C application instrumented with 6 Crux API calls

Crux PAG from running Python (top) and C (bottom) applications instrumented with Crux API. Since both programs perform the same I/O and data operations, we expect the PAGs to be identical. The text on each node corresponds to the data state’s location. The null vertex (red) represents a vertex from a previous application.
In Chapter 1.1 we discussed DroughtHPC as a motivating case for Crux. To approximate the number of Crux API calls needed for a smaller scale HPC application such as DroughtHPC, we estimated the number of Crux data states that would have to be created. We performed an inventory of files accessed by DroughtHPC and VIC as reported by Suriyakumar et al. [11] and estimated that a minimum of 312,000 Crux API calls would be needed to build the Crux PAG.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td># API calls for loading a file</td>
<td>2</td>
</tr>
<tr>
<td># API calls for writing a file</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DroughtHPC &amp; VIC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># input files per run of VIC</td>
<td>25</td>
</tr>
<tr>
<td># output files per run of VIC</td>
<td>25</td>
</tr>
<tr>
<td># files accessed for data assimilation per simulated day</td>
<td>200</td>
</tr>
<tr>
<td># samples</td>
<td>50</td>
</tr>
<tr>
<td># days</td>
<td>30</td>
</tr>
</tbody>
</table>

**Estimated number of Crux API calls for DroughtHPC**

<table>
<thead>
<tr>
<th></th>
<th>Total data states related to VIC + total data states related to data assimilation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300k + 12k</td>
</tr>
<tr>
<td></td>
<td>312k</td>
</tr>
</tbody>
</table>

Table 7.3.2: Estimating number of Crux API calls needed for DroughtHPC

Estimating the number of Crux API calls requires estimating the number of data states in the Crux PAG. We map data state creating to number of files accessed in each call to VIC and during data assimilation. Two Crux API calls are needed for loading and writing a file (1 for creating a new data state vertex and 1 for linking that vertex to a previous vertex).

### 7.4 Future Work

Our API load tests suggest that Crux’s API server could see considerable performance increase from using a statically typed, compiled language like Go instead of a dynamically typed language like Python. Running a REST API server involves many
operations to serialize and deserialize JSON strings, which our Python code handles by converting JSON strings to dictionary objects when receiving data and then converting dictionary objects to JSON strings when it sends data. Furthermore, our Python code runs as a single threaded process. Therefore, switching our implementation language to a compiled language with strong concurrency support would likely increase the performance of the Crux API. Our performance experiment results also suggested that the AWS Application Load Balancer may be a bottleneck to Crux API instances on ECS. We consider as future work using the AWS Network Load Balancer instead which would the direct TCP connections from clients to Crux API instances. The AWS NLB is capable of handling millions of requests per second while maintaining ultra-low latencies [67].

We consider as future work a custom Crux GUI. The custom Crux GUI should address the limitation we observed with the Neo4j Browser, particularly in highlighting the critical path against a visualized PAG. There are many graph visualization libraries to potentially leverage for developing a Crux GUI such as Neoviz.js, D3, or Graphviz. Additionally, the future Crux GUI should easily allow a user to save workflow critical path results from multiple runs.

Adoption of Crux for use in production science environments will require methods for Crux users to instrument their workflow applications. We outlined at a high-level way that this could happen in Chapter 4.7. Future work could consider automated approaches such as compiler or binary instrumentation to avoid manual insertion of API calls. Furthermore, we recommend instrumenting sample HPC applications with workflows matching our simulated workflow cases as next step work.
Chapter 8: Conclusion

In this thesis, we introduced Workflow Critical Path and showed how it can be used to determine the critical path for Holistic HPC Workflows. We implemented a prototype tool called Crux for calculating WCP and provided evidence of its capability and performance using a workflow simulator to generate common workflow patterns. The key contributions of this thesis are:

1. *Defined a novel performance metric called workflow critical path:*
   
   We defined a new metric called workflow critical path for holistic HPC workflows. It does this by defining a program activity graph (PAG) where vertices represent data state and edges represent data mutations.

2. *Developed a prototype tool called Crux for calculating WCP:*
   
   We implemented Crux, a distributed, critical path analysis tool for holistic HPC workflows. The Crux prototype consisted of a scalable API server, a graph database for storing PAGs, and UI for visualizing WCP. We defined a data state schema and REST API endpoints for HPC client applications to interact with Crux.

3. *Developed a workflow simulator for generating HPC workflow patterns:*
   
   We developed a system of simulators to simulate HPC workflows and workflow patterns.

4. *Designed a cloud-based, test environment for Crux:*
   
   We provide tools to deploy Crux on AWS and allow users to replicate our test environment for Crux. We provide ways to deploy Crux locally as well.

We conclude that web- and cloud-based technologies can be used to efficiently calculate Workflow Critical Path, a data-oriented critical path metric for Holistic HPC Workflows.
References


