Future Generation Computer Systems 124 (2021) 315–329

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Comparing workflow application designs for high resolution satellite image analysis

Aymen Al-Saadi[a], Ioannis Paraskevakos[a,*], Bento Collares Gonçalves[b], Heather J. Lynch[b], Shantenu Jha[a,c], Matteo Turilli[a]

[a] Department of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ 08854, United States of America
[b] Department of Ecology and Evolution, Stony Brook, NY 11777, United States of America
[c] Brookhaven National Laboratory, United States of America

ARTICLE INFO

Article history:
Received 7 April 2020
Received in revised form 12 March 2021
Accepted 27 April 2021
Available online 4 May 2021

Keywords:
Image analysis
Task-parallel
Scientific workflows
Runtime
Computational modeling

ABSTRACT

Very High Resolution satellite and aerial imagery are used to monitor and conduct large scale surveys of ecological systems. Convolutional Neural Networks have successfully been employed to analyze such imagery to detect large animals and salient features. As the datasets increase in volume and number of images, utilizing High Performance Computing resources becomes necessary. In this paper, we investigate three task-parallel, data-driven workflow designs to support imagery analysis pipelines with heterogeneous tasks on high performance computing platforms. We analyze the capabilities of each design when processing 3097 and 1575 images for two distinct use cases, for a total of 4,672 satellite and aerial images and 8.35 TB of data. We experimentally model the execution time of the tasks of the image processing pipelines. We perform experiments to characterize resource utilization, total time to completion and overheads of each design. Our analysis shows which design is best suited to scientific pipelines with similar characteristics.

1. Introduction

A growing number of scientific domains are adopting workflows that use multiple analysis algorithms to process a large number of images. The volume and scale of data processing justifies the use of parallelism, tailored programming models and high performance computing (HPC) resources. While these features create a large design space, the lack of architectural and performance analyses makes it difficult to chose among functionally equivalent implementations.

In this paper, we focus on the design of computing frameworks that support the execution of heterogeneous tasks on HPC resources to process large imagery datasets. These tasks may require one or more CPUs and GPUs, implement diverse functionalities and execute for different amounts of time. Typically, tasks have data dependences and are therefore organized into workflows. Due to task heterogeneity, executing workflows poses challenges of effective scheduling, correct resource binding and efficient data management. HPC infrastructures exacerbate these challenges by privileging the execution of single, long-running tasks.

From a design perspective, a promising approach to address those challenges is isolating tasks from execution management. Tasks are assumed to be self-contained programs which are executed in the operating system (OS) environment of HPC compute nodes. Programs implement the domain-specific functionalities required by use cases while computing frameworks implement resource acquisition, task scheduling, resource binding, and data management.

Compared to approaches in which tasks are functions or methods, a program-based approach offers several benefits as, for example, simplified implementation of execution management, support of general purpose programming models, and separate programming of management and domain-specific functionalities. Nonetheless, program-based designs impose performance limitations, including OS-mediated intertask communication and task spawning overheads, as programs execute as OS processes and do not share a memory space.

Due to their performance limitations, program-based designs of computing frameworks are best suited to execute compute-intensive workflows in which each task requires a certain amount of parallelism and runs from several minutes to hours. The use of modern HPC infrastructures with large numbers of CPUs/GPUs presents new challenges to the design of program-based workflows that require heterogeneous, compute-intensive tasks that process large amounts of data.

We use two paradigmatic use cases from the polar science domain to evaluate three alternative designs of computing frameworks for executing program-based tasks, and experimentally

* Corresponding author.
E-mail address: i.paraskev@rutgers.edu (I. Paraskevakos).

https://doi.org/10.1016/j.future.2021.04.023
0167-739X/© 2021 Elsevier B.V. All rights reserved.

characterize and compare their performance. The first use case requires us to detect pack-ice seals by analyzing satellite images of Antarctica taken across a whole calendar year. The resulting dataset consists of 3097 images for a total of 4 TB. This use case requires us to repeatedly process these images, running tasks on both CPUs and GPUs that exchange several GB of data. The second use case requires us to match paired images of penguin colonies from Antarctica and estimate the approximate location where images were taken. The dataset contains 1575 images for a total of 1 TB.

The first design uses a pipeline to independently process each image, while the second and third designs use the same pipeline to process a series of images with differences in how images are bound to available compute nodes.

Leveraging and extending the results presented in Ref. [1], this paper offers four main contributions: (1) a GPU implementation of the Scale Invariant Fast Transformation (SIFT) algorithm to serve the purpose of geolocating satellite imagery; (2) an indication of how to further the implementation of our workflow engine so as to support the class of use cases we considered, while minimizing workflow time to completion and maximizing resource utilization; (3) specific design guidelines for supporting data-driven, compute-intensive workflows on high-performance computing resources with a task-based computing framework; and (4) an experiment-based methodology to compare performance of alternative designs that does not depend on the use case and computing framework presented in this paper.

The paper is organized as follows. Section 2 provides a survey of the state of the art. Section 3 presents the use cases in more detail and discusses their computational requirements as well as the individual stages of the pipelines. Section 4 describes and discusses the novel implementation of SIFT and its performance. Section 5 discusses the three program-based designs in detail. Section 6 details our performance evaluation, discussing the results of our experiments. In Section 7, we summarize the contributions of this paper and identify some new lines of research that it opens.

2. Related work

Several tools and frameworks are available for image analysis based on diverse designs and programming paradigms, and implemented for specific resources. Numerous image analytics frameworks for medical, astronomical, and other domain specific imagery provide MapReduce [2] implementations. MaReIA [3], built for medical image analysis, is based on Hadoop and Spark [4]. Kira [5], built for astronomical image analysis, also uses Spark and pySpark, allowing users to define custom analysis applications. Further, Ref. [6] proposes a Hadoop-based cloud Platform as a Service, utilizing Hadoop's streaming capabilities to reduce filesystem reads and writes. These frameworks support clouds and/or commodity clusters for execution.

BIGS [7] is a framework for image processing and analysis. BIGS is based on the master–worker model and supports heterogeneous resources, such as clouds, grids and clusters. BIGS deploys a number of workers to resources, which query its scheduler for jobs. When a worker can satisfy the data dependencies of a job, it becomes responsible to execute it. BIGS workers can be deployed on any type of supported resource. The user is responsible for defining the input, processing pipeline and launching BIGS workers. As soon as a worker is available, execution starts. In addition, BIGS offers a diverse set of APIs for developers. BIGS approach is very close to Design 1 we described in Section 5.1.

LandLab [8] is a framework for building, coupling and exploring two-dimensional numerical models for Earth-surface dynamics. LandLab provides a library of processing constructs. Each construct is a numerical representation of a geological process. Multiple components are used together, allowing the simulation of multiple processes acting on a grid. The design of each component is intended to work in a plug-and-play fashion. Components couple simply and quickly but parallelizing Landlab components is left to the developer.

The High Performance Computing (HPC)/ High Throughput Computing (HTC) Software Infrastructure for the Synthesis and Analysis of Cosmic Microwave Background (CMB) Datasets [9] is a project to enable CMB experiments to seamlessly use both HPC and HTC systems for their simulation and processing needs. Specifically, this project develops compatible data models to enable bidirectional data flow among pipeline components, concurrently executing on HPC, HTC and hybrid infrastructures. The project extends the Time Ordered Astrophysics Scalable Tools (TOAST) to support data translation and unification across these infrastructures.

The Sea Ice High Resolution Image Analytics (ArcCI) [6] is a framework that uses cloud computing for big data management and visualization. ArcCI is implemented as a set of web services to collect, search, explore, visualize, organize, analyze and share collections of high spatial resolution Arctic sea ice imagery. Currently, ArcCI supports 35 datasets for a total of 1.96 TB of data.

The Large-scale IMage Processing Infrastructure Development (LIMPID) [10] project developed the Bio-Image Semantic Query User Environment (BisQue) for managing, analyzing and sharing images and metadata for large-scale problems. The main goal of BisQue is to enable reproducible image data science on cloud platforms, supporting multiple imaging modalities such as photographs, satellites and microscopes.

Image analysis libraries, frameworks and applications have been proposed for HPC resources. PIMA(GE)^2 Library [11] offers a low-level API for parallel image processing using MPI and CUDA. SIBIA [12] is a framework for coupling biophysical models with medical image analysis, providing users parallel computational kernels through MPI and vectorization. Ref. [13] proposes a scalable medical image analysis service. This service uses DAX [14] as an engine to create and execute image analysis pipelines. Tomosaic [15] is a Python framework, used for medical imaging, employing MPI4py to parallelize different parts of the workflow.

Petruzza et al. [16] describe a scalable image analysis library. Their approach defines pipelines as data-flow graphs, with user defined functions as tasks. Charm++ is used as the workflow management layer, by abstracting the execution level details, allowing execution on local workstations and HPC resources. Teodoro et al. [17] define a master–worker framework supporting image analysis pipelines on heterogeneous resources. The user defines an abstract dataflow and the framework is responsible for scheduling tasks on CPU or GPUs. Data communication and coordination is done via MPI. Ref. [18] proposes the use of UNICORE [19] to define image analysis workflows on HPCs.

Image classification is an existing problem of interest for computer vision scientists. The most common approaches are using scene and object recognition technology. These approaches identify a set of images and classify them based on the scene of interest, e.g., building, mountain, or lakes. A disadvantage of these approaches is that they do not estimate the approximate geographical location of images.

Another approach in computer vision is geolocating satellite and ground-level imagery. Ghouaiel and Lefèvre [20] proposed an automatic translation for ground photos into aerial viewpoint, the technique specifically supports only wide panoramic photos with an accuracy of 54%.

In the large scale image geolocalization, the approach is based on using the "IM2GPS" algorithm [21]. IM2GPS uses a convolutional neural network (CNN) to geolocalize images against a database of geotagged Internet photographs, used as training data. The approach reaches an accuracy of 25% for the 237 photos in their dataset.

We introduce another image geolocating approach based on the image matching technique to extract the similarity level between two images and estimate the approximate location as values of longitude and latitude. We focus on geolocating a set of historic aerial photo imagery using satellite imagery as a basemap.

Our workflow approach proposes designs for image analysis pipelines that are domain independent, i.e., not specific to medical, astronomical, or other domain imagery. Both the workflow and runtime systems we use allow execution on multiple HPC resources with no change in our approach, independent from the types, durations and sizes of task that workflows require to execute. Furthermore, in one of the proposed designs, parallelization is inferred, allowing correct execution regardless of the multi-core or multi-GPU capabilities of the used resource.

All the above, except Ref. [5], focus on characterizing the performance of the proposed solution. Ref. [5] compares different implementations, one with Spark, one with pySpark, and an MPI C-based implementation. This comparison is based on the weak and strong scaling properties of the approaches. Our approach offers a well-defined methodology to compare different designs for task-based and data-driven pipelines with heterogeneous tasks.

3. Satellite imagery analysis use cases

In this paper we developed and characterized computational workflows that satisfy the requirements of two earth science use cases. The first use case, labeled as UC1, requires to process imagery to find Antarctic pack-ice seals. The second use case, labeled as UC2, geolocates an aerial image using a satellite image as a basemap. These use cases require to develop application workflows in which images are processed and analyzed in multiple stages in order to find some relevant properties. This application pattern is used in many scientific domains and, as such, our two use cases are paradigmatic of a common set of computing requirements.

3.1. Seals use case (UC1)

The imagery employed by ecologists as a tool to survey populations and ecosystems come from a wide range of sensors, e.g., camera-trap surveys [22] and aerial imagery transects [23]. However, most traditional methods can be prohibitively labor-intensive when employed at large scales or in remote regions. Very High Resolution (VHR) satellite imagery provides an effective alternative to perform large scale surveys at locations with poor accessibility such as surveying Antarctic fauna [24]. To take full advantage of increasingly large VHR imagery, and reach the spatial and temporal breadths required to answer ecological questions, it is paramount to automate image processing and labeling.

Convolutional Neural Networks (CNN) represent the state-of-the-art for nearly every computer vision routine. For instance, ecologists have successfully employed CNNs to detect large mammals in airborne imagery [25,26] and camera-trap survey imagery [27]. We use a CNN to survey Antarctic pack-ice seals in VHR imagery. Pack-ice seals are a main component of the Antarctic food web [28]; estimating the size and trends of their populations is key to understanding how the Southern Ocean ecosystem copes with climate change [29] and fisheries [30].

For this use case, we process WorldView 3 (WV03) panchromatic imagery as provided by DigitalGlobe Inc. This dataset has the highest available resolution for commercial satellite imagery. We refrain from using imagery from other sensors because pack-ice seals are not clearly visible at lower resolutions. For our CNN architecture, we use a U-Net [31] variant that counts seals with an added regression branch and locates them using a seal intensity heat map. To train our CNN, we use a training set of 53 WV03 images, with 88,000 hand-labeled tiles, where every tile has a correspondent seal count and a class label (i.e., seal vs. non-seal). For hyper-parameter search, we train CNN variants for 75 epochs (i.e., 75 complete runs through the training set) using an Adam optimizer [32] with a learning rate of 10^{-3} and tested against a validation set. The validation set consists of 10% of the training set. In addition, we randomized the WV03 image selection so that a validation tile does not overlap with a training one. Testing was performed on 5 WV03 images. Double observer seal counting was performed and model detection results were compared to observer consensus detections. Furthermore, we avoided double counting by setting a minimum distance boundary between neighboring seals and keeping those where the model was more confident.

We use the best performing model on an archive of over 3097 WV03 images, with a total dataset size of 4 TB. Due to limitations on GPU memory, it is necessary to tile WV03 images into smaller patches before sending input imagery through the seal detection CNN. Taking tiled imagery as input, the CNN outputs the latitude and longitude of each detected seal. While the raw model output still requires statistical treatment, such "mock-run" emulates the scale necessary to perform a comprehensive pack-ice seal census. We order the tiling and seal detection stages into a pipeline that can be re-run whenever new imagery is obtained. This allows domain scientists to create seal abundance time series that can aid in Southern Ocean monitoring.

3.2. Image geolocation use case (UC2)

We introduce the image geolocating use case for two main reasons: (i) Image geolocation can help domain scientists to assess the impact of global warming on climate change; and (ii) a second use case with different computational requirements helps to validate our design.

Image geolocating or geotagging is the process of appending geographical identification metadata to images. Each image is paired to other images and specialized algorithms are used to extract, compare and match relevant image features. In this way, different images of the same geographical location can be matched. In earth science, geolocating can be useful to match datasets of geographical areas recorded at different points in time, by different instruments, with different camera viewpoint, orientation, resolution and brightness.

For this use case, we process aerial and satellite panchromatic imagery. The aerial imagery was taken in 2000 and provided by the U.S. Antarctic Resource Center (USARC). The satellite imagery of the same area was taken in 2017 by WorldView 2 (WV02) and provided by DigitalGlobe Inc. The geolocating process involves two main operations: image matching and rectifying of false positive and false negative matching. For the former we used the scale-invariant feature transform (SIFT) algorithm [33] and for the latter the random sampling consensus (RANSAC) algorithm [34].

SIFT is a feature detection algorithm developed for computer vision to detect commonalities among different images with a stated degree of accuracy and number of probable false matches. Importantly, SIFT results are invariant to image resizing and rotation, and partially invariant to changes in brightness and camera viewpoint. RANSAC is an iterative method to detect outliers in a provided dataset. It is a "learning" algorithm because it fits a model to multiple random samples of the dataset and returns the model that best fits a subset of the data. In this context, it is used to evaluate the set of matched features produced by SIFT and to separate false positive matches.

For our use case, image matching required us to first divide every satellite image into smaller rectangular tiles of the same size. This process created a set of tiles between 2000^2 px and 5000^2 px, discarding tiles that were at the edge of an image. Every tile from one satellite image was then matched against all the aerial images to find overlapping keypoints, i.e., common features such as edges, corners, blobs/regions, and ridges. The similarity between source and target images was measured as the total number of matched keypoints, as extracted by SIFT.

4. GPU-SIFT implementation and performance characterization

Currently, two main implementations of SIFT are freely available: CPU-SIFT [35] and CUDA-SIFT [36]. As required by the Geolocation use case described in Section 3.2, CPU-SIFT supports raw GeoTIFF satellite imagery and can process image tiles up to 5000^2 px or more. Unfortunately, CPU-SIFT is memory inefficient, especially with large tiles, and cannot use GPUs. CUDA-SIFT supports GPUs but does not support raw GeoTIFF satellite imagery and can process tiles only up to 1920×1080 px.

To address these challenges, we extended CUDA-SIFT [36] and developed GPU-SIFT [37]. We extended CUDA-SIFT instead of CPU-SIFT because GPUs offer shorter execution times compared to CPUs. That allows us to better support use cases in which the size of datasets grows over time, requiring increasing amount of computing resources and execution time.

GPU-SIFT offers the following functionalities: (i) reading dual-band 8 and 16 bit GeoTIFF satellite imagery; (ii) reading GeoTIFF images larger than 1920×1080 px; (iii) enabling CUDA to allocate up to 4 GB GPU memory per image; (iv) implementing CUDA Multi-Process Service (MPS) technology to run 2 CUDA kernels on a single GPU device; and (v) implementing adaptive contrast enhancer.

We characterize and compare the performance of CPU-SIFT, CUDA-SIFT, and GPU-SIFT based on three metrics: throughput, memory consumption and matching accuracy. We match single pairs of increasingly large GeoTIFF images (source and target), measuring how many MB are processed by the CPU or GPU per second, how much memory the matching of the images required and how accurate such a matching was. Note that, in this context, throughput refers to the volume of data processed per unit of time (MB/s).

We performed all our experiments on the XSEDE Bridges supercomputer [38]. Bridges offers 32 nodes with 2 T P100 GPUs and 32 GB of GPU-dedicated memory, and 2 16-cores Intel Broadwell E5-2683 CPUs with 128 GB of RAM. We used RADICAL-Pilot [39] to manage the execution of our experiment workloads on Bridges. For all the experiments, we use a single GeoTIFF image of 845 MB, tiled into four predefined sizes: 90, 180, 360, and 720 MB. We then match two copies of the same tile for each size, using each SIFT implementation.

4.1. Throughput

We measure the throughput of CPU-SIFT, CUDA-SIFT, and GPU-SIFT in MB/s when processing the same pair of 90, 180, 360 and 720 MB images. CPU-SIFT and CUDA-SIFT can analyze one pair of images per CPU/GPU, while GPU-SIFT can use CUDA MPS to analyze two pairs of images per GPU. Consistently, in our experiments we concurrently execute two CPU-SIFT on two CPUs, two CUDA-SIFT on two GPUs and one GPU-SIFT on one GPU.

Fig. 1(a) shows that CPU-SIFT (red) has the lowest throughput with a value of 54.38 MB/s. This is due to CPU-SIFT programming model: CPU-SIFT uses one core per CPU to process and match two images. As a result, CPU-SIFT cannot leverage the parallelism supported by multi-core architecture to increase throughput.

CUDA-SIFT (dark green) has a throughput almost six times higher than CPU-SIFT, with a value of 244.44 MB/sec. In contrast to CPU-SIFT, CUDA-SIFT uses the CUDA framework to load balance calculations across the cores of the GPU device. Nonetheless, note that with the given images, CUDA-SIFT uses only up to 1658 of the 3584 CUDA cores available on the Nvidia P100 GPU devices of our experiments. There are not enough data to process to saturate the GPU cores.

GPU-SIFT (light green) has the highest throughput among the three SIFT implementations, with value of 478.25 MB/s. GPU-SIFT uses 3400 of the 3584 available CUDA cores, running 2 pair image comparisons concurrently. Note that GPU-SIFT throughput is almost double that of the CUDA-SIFT. This shows that the overheads imposed by the CUDA Multi-Process Service (MPS) are negligible for our implementation.

Fig. 1(a) shows that GPU throughput is invariant to image size. Increasing the image size increases the amount of data processed by the GPU or CPU per second, bounded by the available GPU or CPU bandwidth. Note that the error bars for the GPU implementations are smaller than those of the CPU implementation. This is likely due to the efficiency of the CUDA framework and the absence of competing processes on the GPU subsystem.

4.2. Memory consumption

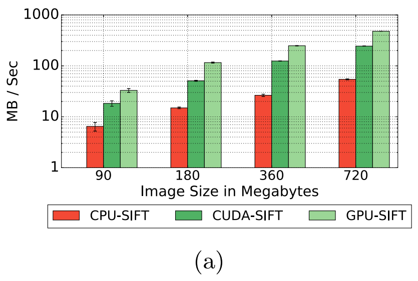
We measure the memory consumption of CPU-SIFT, CUDA-SIFT and GPU-SIFT as the amount of physical memory a particular program utilizes at runtime. Fig. 1(b) shows the total memory consumption of CPU-SIFT, GPU-SIFT and CUDA-SIFT for pairs of 2000^2 px, 3000^2 px, 4000^2 px and 5000^2 px tiles. The memory usage includes tile reading and the SIFT detecting, extracting, and matching of features.

For the largest tile of 5000^2 px, CPU-SIFT consumes 24.67 GB of memory, almost five times higher than CUDA-SIFT and GPU-SIFT which consume 8.05 GB and 8.08 GB respectively. The minimal difference in memory consumption of GPU-SIFT compared to CUDA-SIFT account for the acquisition of GeoTIFF images and the use of CUDA MPS. This shows the minimal memory overhead that these features require and therefore the minimal cost of doubling the throughput of GPU-SIFT.

4.3. Numbers of matched points

We compare the efficiency of each SIFT implementation in terms of the number of matching points between a pair of images. We picked a pair of images for which 16,850 matches have been previously identified and validated by the domain scientists. We applied CPU-SIFT, GPU-SIFT and CUDA-SIFT on the pair of images with different tile sizes to: (1) measure the number of matches that the three implementations can detect; and (2) validate the assumption that the tile size can affect the number of matches.

We repeated the experiments 75 times to measure the accuracy of both implementations, using the same satellite image with tile sizes of 2000^2 px, 3000^2 px, 4000^2 px and 5000^2 px. Fig. 2(a) shows that the number of matched points detected by CPU-SIFT is more than those detected by GPU-SIFT and CUDA-SIFT. For 5000^2 px tiles, CPU-SIFT returns 13,500 matches out of the existing 16,850 with an accuracy of 80.11%. This is about 5.92% more than GPU-SIFT, and 8.97% more than CUDA-SIFT. GPU-SIFT, plotted in green, returns 12,500 matches with an accuracy of 74.18%, while CUDA-SIFT, plotted in orange, returns 9500 matches with an accuracy of 56.81%.



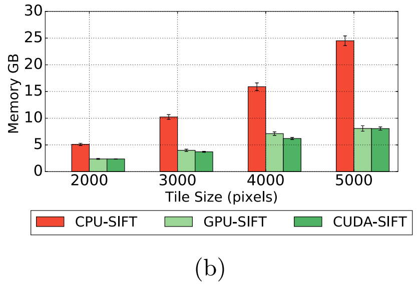
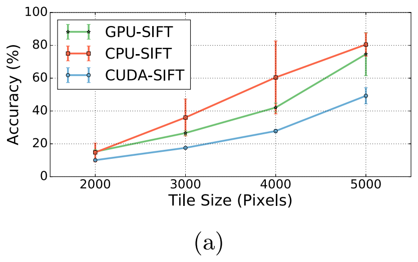


Fig. 1. (a) The throughput of CPU-SIFT, CUDA-SIFT and GPU-SIFT as a function of image size in Megabytes. (b) Memory consumption of CPU-SIFT, CUDA-SIFT and GPU-SIFT as a function of tile size.



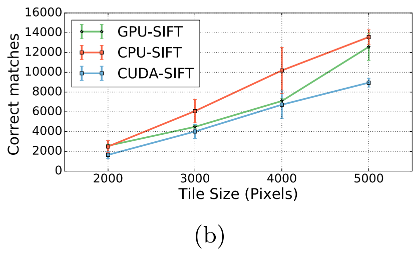


Fig. 2. Shows the matching accuracy (a) and the total number of correct correspondences (b) found with GPU-SIFT, CUDA-SIFT and CPU-SIFT for well-known satellite image as a function of tile size.

CPU-SIFT variability shown in Fig. 2(a) are larger than those of CUDA-SIFT and GPU-SIFT and it remains unclear why CPU-SIFT accuracy is inconsistent. In absolute terms, CPU-SIFT is the implementation that can reach the highest number of matches but, on average, GPU-SIFT offer a more reliable performance.

CPU-SIFT and GPU-SIFT apply contrast enhancement to both source and target images before processing them, while CUDA-SIFT does not apply any contrast enhancement. Enhancing the level of contrast of an image can increase the numbers of detected features and as a result, it directly increases the number of matches between both images [40].

We distinguish between validation of the results produced by SIFT and validation of GPU-SIFT against CPU-SIFT. The former pertains to the accuracy of the matches produced by SIFT, including false positives; the latter to the consistency between the matches produced by GPU-SIFT and CPU-SIFT. GPU-SIFT and CPU-SIFT produce comparable results, with variations introduced by optimizations specific to CPU-SIFT that still have to be ported to GPU-SIFT.

For the domain scientists interested in this use case, the main goal is to obtain effective matches of specific features of an image. So far, SIFT proved relatively inefficient for this specific task. Our GPU-SIFT implementation opens the possibility to investigate the role that ML-driven algorithms requiring GPU support could play to improve the accuracy of SIFT matches. A recent application of machine learning image matching [41,42], shows promising speed improvements, further shifting the focus of the domain scientists from validating the accuracy of GPU-SIFT to study its accuracy when augmented with ML algorithms.

5. Workflow design and implementation

Computationally, the use cases described in Section 3 present three main challenges: heterogeneity, scale and reusability. The images of the use cases' datasets have a wide distribution in size. Each image requires a series of tasks to get the aggregated result. These tasks are memory and computational intensive, requiring CPU and GPU implementations. Whenever the image dataset is updated, it needs to be reprocessed.

We address these challenges by codifying image analyses into workflows. We then execute these workflows on HPC resources, leveraging the concurrency, storage systems and compute speed they offer to reduce time to completion. Typically, the workflows of our use cases consist of a sequence (i.e., pipeline) of tasks, each performing part of the end-to-end analysis on one or more images. We compare two common designs for the execution of these workflows: one in which each image is processed independently by a dedicated pipeline, and the other in which a single pipeline processes multiple images.

Note that both designs separate the functionalities required to process each image from the functionalities used to coordinate the processing of multiple images. This is consistent with moving away from vertical, end-to-end single-point solutions, favoring designs and implementations that satisfy multiple use cases, possibly across diverse domains. Accordingly, the designs we implement and characterize, employ two tasks (i.e., standalone executable programs) to provide the functionalities required by the use cases.

The designs are functionally equivalent, in that they both enable the analysis of the given image datasets. Nonetheless, each design leads to different concurrency, resource utilization and overheads, depending on compute-data affinity, scheduling algorithms, and coordination between CPU and GPU computations. We analyze the performance of these designs using the common metrics of total execution time, resource utilization, and middleware overheads.

Consistent with HPC resources currently available and our use cases, we make three assumptions: (1) each compute node has c CPUs; (2) each compute node has g GPUs where $g \leq c$; and (3) each compute node has enough memory to enable concurrent execution of a certain number of tasks. As a result, at any given point in time there are $C = n \times c$ CPUs and $G = n \times g$ GPUs available, where n is the number of compute nodes.

5.1. *Design 1: One image per pipeline*

We specify the workflow for either counting the number of seals in a set of images, or geolocating pairs of images as a set

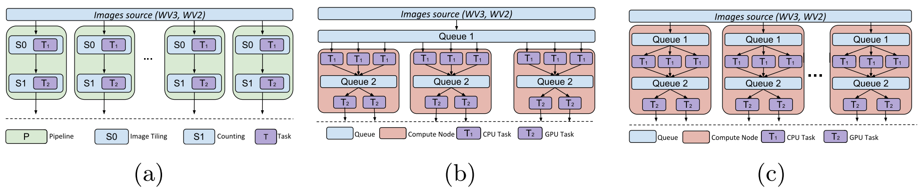


Fig. 3. Design approaches to implement the workflow required for the use cases of Section 3. 3(a)–**Design 1**: Pipeline, stage and task based design. 3(b)–**Design 2**: Queue based design with a single queue for all the tasks of the pipeline. 3(c)–**Design 2.A**: Queue based design with multiple queues for the tasks of the pipeline.

of pipelines. Each pipeline is composed of two stages, each with multiple instances of one type of task. In the Seals use case (UC1), we specify the workflow for counting the number of seals in a set of images. In the Geolocation use case (UC2), we specify the workflow to match a pair of images in two sets of aerial and satellite images.

In UC1, the task of the first stage gets an image as input and generates tiles of that image based on the tile size as output. In UC2, the task of the first stage gets a pair of images as input and produces a set of matches between these two images as output. The task of the second stage gets the output of the first stage – generated tiles or matches – and outputs the number of seals per image for UC1 and a set of matches with reduced false positives for UC2.

Formally, we define two types of tasks in UC1 and UC2:

- $T_1^{UC1} = \langle I, f_I, t \rangle$, where I is an image or a pair of images, f_I is a stage 1 type function and t is a set of tiles or matches that correspond to I.
- $T_2^{UC1} = \langle t, f_A, S \rangle$, where f_A is a stage 2 type function and S is either the number of seals or final matches.
- $T_1^{UC2} = \langle I_p, k_I, m \rangle$, where I_p is a pair of images, k_I is a matching function and m is a set of matches as a file that correspond to I_p.
- $T_2^{UC2} = \langle m, f_R, t \rangle$, where f_R is a function that eliminates the undesired matches and t is the output file of filtered matches.

Tiling in T_1^{UC1} is implemented with OpenCV [43] and Rasterio [44] in Python. Rasterio allows us to open and convert a GeoTIFF WV3 image to an array. The array is then partitioned to sub-arrays based on a user-specified scaling factor. Each sub-array is converted to an compressed image via OpenCV routines and saved to the filesystem.

Seal counting in T_2^{UC1} is performed via a Convolutional Neural Network (CNN) implemented with PyTorch [45]. The CNN counts the number of seals for each tile of an input image. When all tiles are processed, the coordinates of the tiles are converted to the geographical coordinates of the image and saved in a file, along with the number of counted seals. Note that the number of seals in a tile does not affect the execution of the network, i.e., the same number of operations will be executed.

Matching in T_1^{UC2} is implemented by GPU-SIFT as described in Section 4, while filtering in T_2^{UC2} is implemented by RANSAC as described in 3.2.

All task implementations for UC1 and UC2 are invariant across the alternative designs we consider. This is consistent with the designs being task-based, i.e., each task exclusively encapsulates the capabilities required to perform a specific operation over an image, pair of images or tile. Thus, tasks are independent from the capabilities required to coordinate their execution, whether each task processes a single image or pair of images, or a sequence of images or pairs of images.

We implemented Design 1 via EnTK, a workflow engine which exposes an API based on pipelines, stages, and tasks [46]. The user can define a set of pipelines, where each pipeline has a sequence of stages, and each stage has a set of tasks. Stages are executed respecting their order in the pipeline while the tasks in each stage can execute concurrently, depending on resource availability.

For our use cases, EnTK has three main advantages compared to other workflow engines: (1) it exposes pipelines and tasks as first-order abstractions implemented in Python; (2) it is specifically designed for concurrent management of up to 10^5 pipelines; and (3) it supports RADICAL-Pilot, a pilot-based runtime system designed to execute heterogeneous bag of tasks on HPC machines [39]. Together, these features address the challenges of heterogeneity, scale and reusability: users can encode multiple pipelines, each with different types of tasks, executing them at scale on HPC machines without explicitly coding parallelism and resource management.

When implemented in EnTK, the workflows of our use cases map to a set of pipelines, each with two stages St_1, St_2. Each stage has a task of type T_1^{UC1-2} and T_2^{UC1-2} respectively. Each pipeline is defined as $P = (St_1, St_2)$. For UC1, the workflow consists of N pipelines and for UC2 the workflow consists of $N \times (N-1)$, where N is the number of images.

Fig. 3(a) shows the abstract workflow for both use cases. For each pipeline, EnTK submits the task of stage St_1 to the runtime system (RTS). As soon as this task finishes, the task of stage St_2 is submitted for execution. This design allows concurrent execution of pipelines and, as a result, concurrent analysis of single images or pair of images, one for each pipeline. Since pipelines execute independently and concurrently, there are instances where St_1 of a pipeline executes at the same time as St_2 of another pipeline.

Design 1 has the potential to increase utilization of available resources as each compute node of the target HPC machine has multiple CPUs and GPUs. Importantly, computing concurrency comes with the price of multiple reads and writes to the filesystem on which the dataset is stored. This can cause I/O bottlenecks, especially if each task of each pipeline reads from and writes to the same filesystem, possibly over a network connection.

For UC1, we used a tagged scheduler for EnTK's RTS to avoid I/O bottlenecks. This scheduler schedules T_1 of each pipeline on the first available compute node, and guarantees that the corresponding T_2 is scheduled on the same compute node. As a result, compute-data affinity is guaranteed among co-located T_1 and T_2. This design reduces I/O bottlenecks but it may also reduce concurrency when the performance of the compute nodes and/or the tasks is heterogeneous: T_2 may have to wait to execute on a specific compute node while another node is free.

5.2. Design 2: Multiple images per pipeline

Design 2 implements a queue-based approach. We introduce two tasks (T_1- T_2) for both UC1 and UC2 as defined in Section 5.1. In contrast to Design 1, these tasks are started and then executed for as long as resources are available, processing input images at the rate taken to process each image or pair of images. For both use cases, the number of concurrent T_1 and T_2 depends on available resources, including CPUs, GPUs, and RAM.

For the implementation of Design 2, we do not need EnTK, as we submit a bag of T_1 and T_2 tasks via the RADICAL-Pilot RTS, and manage the data movement between tasks via queues. As shown in Fig. 3(b), Design 2 uses one queue (Queue 1) for the dataset, and another queue (Queue 2) for each compute node. For each compute node, each T_1 pulls an image or pair of images from Queue 1, generates tiles or matches, and then queues the results to Queue 2. The first available T_2 on that compute node, pulls those tiles or matches from Queue 2, and counts the seals or filters false positive matches.

To communicate data and control signals between queues and tasks, we defined a communication protocol with three entities: Sender, Receiver, and Queue. Sender connects to Queue and pushes data. When done, Sender informs Queue and disconnects. Receiver connects to Queue and pulls data. If there are no data in Queue but Sender is connected, Receiver pulls a "wait" message, waits, and pulls again after a second. When there are no data in Queue or Sender is not connected to Queue, Receiver pulls an "empty" message, upon which it disconnects and terminates. This ensures that tasks are executing, even if starving, and that all tasks are gracefully terminating when all images are processed.

Note that Design 2 load balances T_1 tasks across compute nodes but balances T_2 tasks only within each node. For example, suppose that T_1 on compute node A runs two times faster than T_1 on compute node B. Since both tasks are pulling images from the same queue, T_1 of A will process twice as many images as T_1 of B. Both T_1 of A and B will execute for around the same amount of time until Queue 1 is empty, but Queue 2 of A will be twice as large as Queue 2 of B. T_2 tasks executing on B will process half as many images as T_2 tasks on A, possibly running for a shorter period of time, depending on the time taken to process each image.

In principle, Design 2 can be modified to load balance also across Queue 2 but in practice, as discussed in Section 5.1, this would produce I/O bottlenecks. Load balancing across T_2 tasks would require for all tiles produced by T_1 tasks to be written to and read from a filesystem shared across multiple compute nodes. Keeping Queue 2 local to each compute node enables using the filesystem local to each compute node.

5.2.1. *Design 2.A: Uniform image dataset per pipeline*

The lack of load balancing of T_2 tasks in Design 2 can be mitigated by introducing a queue in each node from where T_1 tasks pull data. This allows early binding of images to compute nodes, i.e., deciding the distribution of input images per node before executing T_1 and T_2. As a result, the execution can be load balanced among all available nodes, depending on the correlation between image properties and image execution time.

Fig. 3(c) shows variation 2.A of Design 2. The early binding of images to compute nodes introduces an overhead compared to using late binding via a single queue as in Design 2. Nonetheless, depending on the strength of the correlation between image properties and execution time, design 2.A offers the opportunity to improve resource utilization. While in Design 2 some node may end up waiting for another node to process a much larger Queue 2, in design 2.A this is avoided by guaranteeing that each compute node has an analogous payload to process.

6. Experiments and discussion

We executed three experiments using the GPU compute nodes of the XSEDE Bridges supercomputer. These nodes offer 32 cores, 128 GB of RAM and two P100 T GPUs. We stored the dataset of the experiments and the output files on Bridges' Pylon5 Lustre filesystem. Specifically, for the Seals use case (UC1 onwards), we stored the tiles produced by the tiling tasks on the local filesystem of the compute nodes. This way, we avoided a potential performance bottleneck from millions of reads and writes of ≈700 KB on Pylon5. The Geolocation use case (UC2 onwards) did not require the use of the node local filesystem since it writes a single file of few MBs per task. We submitted jobs requesting 4 compute nodes to keep the average queue time within a couple of days. Requesting more nodes produced queue times in excess of a week.

The dataset of UC1 consists of 3097 images, ranging from 50 to 2770 MB, for a total of 4 TB of data. The dataset of UC2 consists of 1575 aerial and satellite images, ranging from 1.5 to 5.5 MB for a total of 4.35 GB. We generated 11,552 image pairs to cross-match all aerial images to all satellite images. Those datasets are posed to grow overtime, both in terms of number of images and the size of each image. In turn, that increases the number and size of the generated image pairs. Consequently, in UC2, we used a GPU implementation to: (i) account for the growth of the dataset's number of images and the size of individual images; and (ii) improve the scale and time to completion of the analysis as the GPU implementation is faster than the CPU one.

The image size of both datasets follows a normal distribution. The UC1 dataset has a mean value of 1,304.85 MB and standard deviation of 512.68 MB. The dataset of UC2 has a mean value of 6.13 MB and standard deviation of 1.79 MB.

For Design 1, 2 and 2.A described in Section 5, Experiment 1 models the execution time of the two tasks of our use cases as a function of the image size—the only property of the images for which we found a correlation with execution time; Experiment 2 measures resource utilization for each design; and Experiment 3 characterizes the overheads of the middleware implementing each design. These experiments enable performance comparison across designs, allowing us to draw conclusions about the performance of heterogeneous task-based execution of data-driven workflows on HPC resources.

As already done in Section 5.1, we use T_1^{UC1} and T_2^{UC1} to indicate the first and second type of task for the Seals use case; and T_1^{UC2} and T_2^{UC2} to indicate first and second type of task for the Geolocation use case.

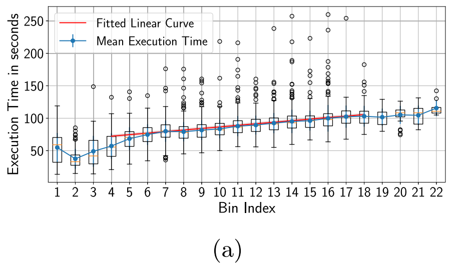
6.1. *Experiment 1, Design 1*

Fig. 4(a) shows the execution time of the tiling task T_1^{UC1} as a function of the image size. We partition the set of images based on image size, obtaining 22 bins with binsize 125 MB each starting from 50 MB up to 2800 MB. The average time to tile an image in each bin tends to increase with the image size. The box-plots show some positive skew of the data with a number of data points falling outside the assumed normal distribution. Thus, there is a weak correlation between task execution time and image size with a large spread across all the image sizes.

There are also large standard deviations (*STD*—blue line) in most of the bins. We explored the causes of the observed values by measuring how it varies in relation to the number of T_1^{UC1} concurrently executing on the same node. The *STD* observed was consistent across degrees of task concurrency, allowing us to conclude that it depends on fluctuation in the node performance [1].

Fig. 5(a) shows the execution time of the image matching task T_1^{UC2} as a function of the size of an image pair. In this use case, we partitioned the data based on the total size of the image pair as each task processes two images at the same time. Fig. 5(a) shows 22 bins of 187 KB, each in a range of [1.0, 5.5] MB.

Fig. 4(a) indicates that the execution time is a linear function of the image size between bins 4 and 18. Bins 1–3 and 19–23 are not representative as the head and tail of the image sizes distribution contain less than 5% of the image dataset. Similarly, in Fig. 5(a) bins 5–19 show linear behavior and bins 1–4 and 20–23 are omitted from the analysis as they contain less than 4% of



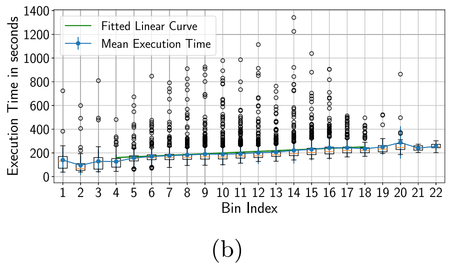
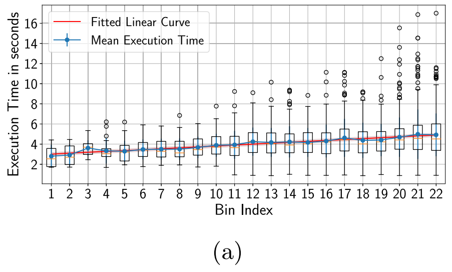


Fig. 4. Experiment 1, Design 1, UC1: Box-plots of (a) T_1^{UC1} and (b) T_2^{UC1} execution times, means and standard deviations (*STDs*) for 125 MB image size bins. Red line shows fitted linear function for T_1^{UC1}, green line for T_2^{UC1}. Red shadow shows confidence interval for T_1^{UC1}, green shadow for T_2^{UC1}.



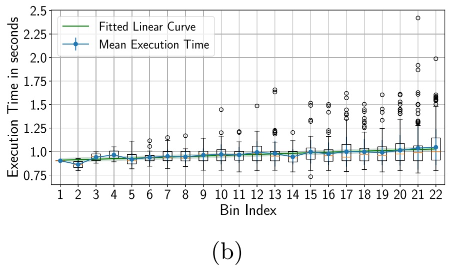


Fig. 5. Experiment 1, Design 1, UC2: Box-plots of (a) T_1^{UC2} and (b) T_2^{UC2} execution times, means and standard deviations (*STDs*) for 187 MB image size bins. Red line shows fitted linear function for T_1^{UC2}, green line for T_2^{UC2}. Red shadow shows confidence interval for T_1^{UC2}, green shadow for T_2^{UC2}.

the data set. Accordingly, for both UC1 and UC2, we model the execution time as:

$$T(x) = \alpha \times x + \beta \tag{1}$$

where x is the image size. We found the parameter values of Eq. (1) by using a non-linear least squares algorithm to fit experimental data (see Table 1).

Fig. 4(b) shows the execution time of T_2^{UC1} as a function of the image size. This task presents a different behavior than T_1^{UC1}, as the code executed is different. Note the slightly stronger positive skew of the data compared to that of Fig. 4(a) but still consistent with our conclusion that deviations are mostly due to fluctuations in the node performance, i.e., different code similar fluctuations.

Similar to T_1^{UC1}, Fig. 4(b) shows a weak correlation between the execution time of T_2^{UC1} and image size. In addition, the variance per bin is relatively similar across bins, as expected based on the analysis of T_1^{UC1}. The box-plot and mean execution time indicate that a linear function is a good candidate for a model of T_2^{UC1}. We fitted a linear function (as in Eq. (1)) to the execution time as a function of the image size for the same bins as T_1^{UC1}.

Fig. 5(b) shows the execution time of T_2^{UC2} as function of the size of the image pair. We notice a weak correlation between T_2^{UC2} execution time and the size of the image pair. Further, we see a similar variance among the bins as that measured for T_2^{UC1}. We notice that the execution time becomes positively skewed as the image pair size increases, as shown in Figs. 5(a) and 5(b). Node usage increases with image size, making the observed fluctuations consistent with the analysis from UC1.

Based on Table 1, the R^2 values for T_1^{UC1}, T_2^{UC1}, T_1^{UC2} and T_2^{UC2} show a good fit of the respective lines to the actual data. As a result, we can conclude that our estimated functions are validated.

6.2. Experiment 1, Design 2

Fig. 6(a) shows the execution time of T_1^{UC1} as a function of the image size for Design 2. In principle, design differences in middleware that execute tasks as independent programs should not directly affect task execution time. In this type of middleware, task code is independent from that of the middleware: once tasks execute, the middleware waits for each task to return. Nonetheless, in real scenarios with concurrency and heterogeneous tasks, the middleware may perform operations on multiple tasks while waiting for others to return. Accordingly, in Design 2 we observe an execution time variation comparable to that observed with Design 1 but Fig. 6(a) shows a stronger positive skew of the data for Design 2 than Fig. 4(a) for Design 1.

We investigated the positive skew of the data observed in Fig. 6(a) by comparing the system load of a compute node when executing the same number of tiling tasks for Design 1 and 2. The system load of Design 2 was higher than that of Design 1. As we used the same type of task, image and task concurrency, we conclude that the middleware implementing Design 2 uses more compute resources than that used for Design 1. Due to concurrency, the middleware of Design 2 competes for resources with the tasks, momentarily slowing down their execution. This is consistent with the architectural differences across the two designs: Design 2 requires resources to manage queues and data movement while Design 1 has only to schedule and launch tasks on each node.

Design 2 also produces a much stronger positive skew of T_2^{UC1} execution time compared to executing T_2^{UC1} with Design 1 (see Fig. 6(b)). T_2^{UC1} executes on GPU and T_1^{UC1} on CPU but their execution times have comparable skew in Design 2. This further supports our hypothesis that the long tail of the distribution of T_1^{UC1} and especially T_2^{UC1} execution times, depends on the competition for main memory and I/O between the middleware and the executing tasks.

Table 1 shows the model parameters for both tasks and their respective R^2 values. R^2 are worse compared to Design 1. This is expected based on the positive skew of the data observed in Design 2.

Fig. 7(a) shows the execution time of T_1^{UC2} as a function of image pair size for Design 2. We notice an execution time variation comparable to the one shown in Fig. 5(a) for Design 1. Nonetheless, while the number of outliers is lower than in Design 1, their spread is higher: between 10 s and 80 s for Design 2 compared to 5 s and 16 s for Design 1. As with UC1, this positive

Table 1
Fitted parameter values of Eq. (1) using a non-linear least squares algorithm to fit our experimental data.

Design	Fitted data	α value	β value	R^2 value	Figure
1	T_1^{UC1}	1.92×10^{-2}	60.49	0.97	Fig. 4(a), red line
1	T_2^{UC1}	5.21×10^{-2}	128.53	0.96	Fig. 4(b), green line
1	T_1^{UC2}	0.93	2.45	0.97	Fig. 5(a), red line
1	T_2^{UC2}	5.21×10^{-2}	128.53	0.96	Fig. 5(b), green line
2	T_1^{UC1}	3.17×10^{-2}	64.81	0.92	Fig. 6(a), red line
2	T_2^{UC1}	4.71×10^{-2}	95.83	0.95	Fig. 6(b), green line
2	T_1^{UC2}	0.62	1.52	0.61	Fig. 7(a), red line
2	T_2^{UC2}	3.16×10^{-2}	0.29	0.51	Fig. 7(b), green line
2.A	T_1^{UC1}	2.74×10^{-2}	49.03	0.94	N/A
2.A	T_2^{UC1}	4.80×10^{-2}	87.60	0.95	N/A
2.A	T_1^{UC2}	0.54	1.51	0.76	N/A
2.A	T_2^{UC2}	2.82×10^{-2}	0.26	0.89	N/A

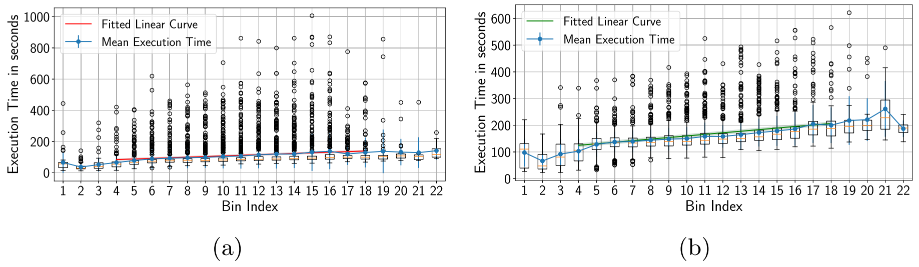


Fig. 6. Experiment 1, Design 2, UC1: Box-plots of (a) T_1^{UC1} and (b) T_2^{UC1} execution times, means and standard deviations (*STDs*) for 125 MB image pair size bins. Red line shows fitted linear function for T_1^{UC1}, green line for T_2^{UC1}. Red shadow shows confidence interval for T_1^{UC1}, green shadow for T_2^{UC1}.

skewness is due to the increased system load per compute node with Design 2. The fitted parameter values are shown in Table 1 indicating a good fit.

Fig. 7(b) shows the execution time of T_2^{UC2}. We notice a skew of the data similar to what observed for T_2^{UC2} in Design 1 and a larger spread of the outliers. The latter supports our hypothesis that the execution time of the task is sensitive to resource competition, especially related to the use of the file system. This wider spread supports a worse fit of our model, as shown in Table 1. Spread apart, there is not much difference between the execution times of T_2^{UC2} tasks in Design 1 and Design 2.

6.3. Experiment 1, Design 2.A

Similarly to the analysis for Design 1 and 2, we fitted UC1 and UC2 data from Design 2.A to Eq. (1). The fitted parameter are shown in Table 1. Based on R^2, we can conclude that all model are good fits for their respective data.

The results of experiment 1 indicate that with Design 2.A, on average, there is a decrease in the execution time of T_1 and an increase in that of $T2$ compared to Design 2, for both use cases. Design 2.A requires one queue more than Design 2 for T_1^{UC1} and therefore more resources for its implementation. This can explain the slowing of T_2 but not the speedup of T_1. This requires further investigation, measuring whether the performance fluctuations of compute nodes are larger than measured so far.

As discussed in Section 5.2, balancing of workflow execution differs between Design 2 and Design 2.A. Figs. 8(a) and 9(a) show that the task T_1 of the two use cases can work on a different number of images but all T_1 tasks concurrently execute for a similar duration. The histograms in Figs. 8(a) and 9(a) also show that this balancing can result in different input distributions for each compute node, affecting the total execution time of the T_2 tasks on each node. Thus, Design 2 can create imbalances in the time to completion of T_2, as shown by the red bars in both Figs. 8(a) and 9(a).

Design 2.A addresses these imbalances by early binding images to compute nodes. Comparing the lower part of Fig. 8(a) with Figs. 8(b) and 9(a) with Fig. 9(b), we notice the difference between the distributions of image size for each node between Design 2 and 2.A. In Design 2.A, due to the modeled correlation between time to completion and the size of the processed image, the similar distribution of the size of the images bound to each compute node balances the total processing time of the workflow across multiple nodes.

Note that Figs. 8 and 9 show just one of the runs we perform for this experiment. Due to the random pulling of images from a global queue performed by Design 2, each run shows different distributions of image sizes across nodes, leading to large variations in the total execution time of the workflow.

Fig. 8(b) shows also an abnormal behavior of one compute node: For images larger than 1.5 GBs, Node 3 CPU performance is markedly slower than other nodes when executing T_1^{UC1}. Different from Design 2, Design 2.A can balance these fluctuations in T_1^{UC1} as far as they do not starve T_2^{UC1} tasks.

Fig. 9(b) shows a more balanced and decreasing execution time of T_1^{UC2} among the 4 nodes, compared to Design 2. We investigated the decreasing of the execution time and we explained it with the different input distribution of the total size of the image pairs (smaller) in Node 3 and 4 compared to Node 1 and Node 2. This image size variation can create a significant fluctuations in the execution time.

6.4. Experiment 2: Resource utilization

Resource utilization varies across Design 1, 2 and 2.A. In Design 1, the runtime system (RTS), i.e, RADICAL-Pilot, is responsible for scheduling and executing tasks. For UC1, T_1^{UC1} is memory intensive and, as a consequence, we were able to concurrently

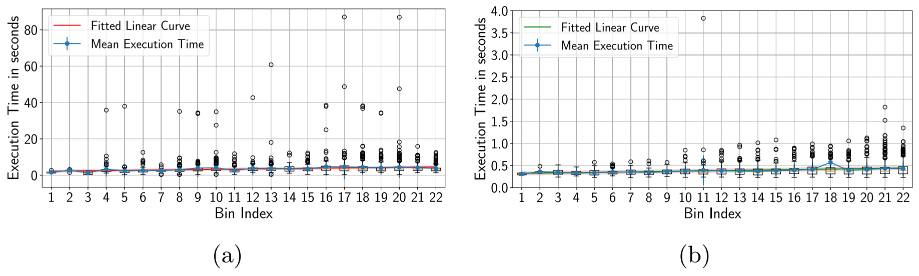


Fig. 7. Experiment 1, Design 2, UC2: Box-plots of (a) T_1^{UC2} and (b) T_2^{UC2} execution times, means and standard deviations (*STDs*) for 187 KB image size bins. Red line shows fitted linear function for T_1^{UC2}, green line for T_2^{UC2}. Red shadow shows confidence interval for T_1^{UC2}, green shadow for T_2^{UC2}.

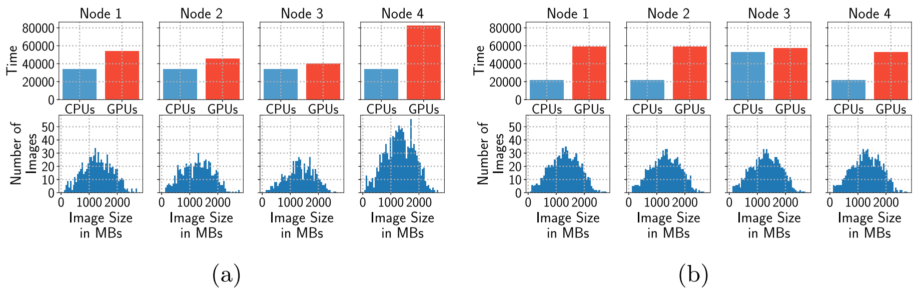


Fig. 8. Experiment 1, Design 2.A, UC1 execution time of T_1^{UC1} (blue) and T_2^{UC1} (red), and distributions of image size per node for (a) Design 2 and (b) Design 2.A.

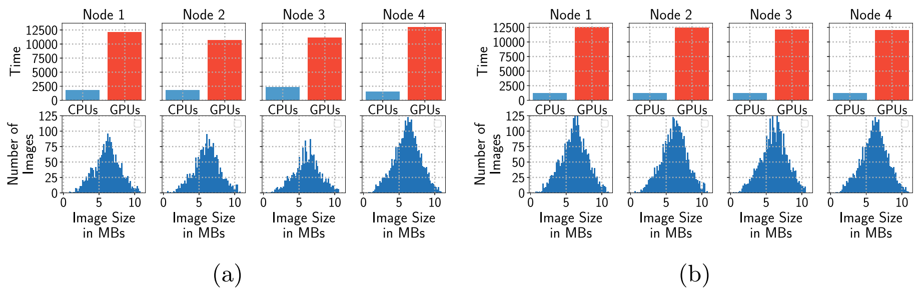


Fig. 9. Experiment 1, Design 2.A, UC2 execution time of T_1^{UC2} (blue) and T_2^{UC2} (red), and distributions of image size per node for (a) Design 2 and (b) Design 2.A.

execute 3 T_1^{UC1} on each compute node, using only 3 of the 32 available CPU cores. We were instead able to execute 2 T_2^{UC1} concurrently on each node, using all the available GPUs. Assuming ideal concurrency among the 4 compute nodes we utilized in our experiments, the theoretical maximum utilization per node would be 10.6% for CPUs and 100% for GPUs.

For UC2, T_1^{UC2} uses GPU-SIFT to process and match a pair of images. T_1^{UC2} is a GPU-memory intensive task that requires an amount of memory proportional to the size of the image pair on which to perform image matching. T_2^{UC2} runs on CPU and requires only one core to perform the RANSAC filtration. We were able to execute 2 T_1^{UC2} and up to 2 T_2^{UC2} concurrently on every compute node, utilizing all of the 8 GPUs but only up to 2 of the 32 available cores per node.

Figs. 10 and 11 show the resource utilization percentage, for all designs, for UC1 and UC2 respectively. CPU utilization for UC1 with Design 1 (Fig. 10(a)) closely approximates the 10.6% theoretical maximum utilization but GPU utilization is well below the theoretical 100%. GPUs are not utilized for almost an hour at the beginning of the execution and utilization decreases to 80% some time after half of the total execution was completed. Our analysis shows that RADICAL-Pilot's scheduler did not schedule GPU tasks at the start of the execution even if GPU resources were available [1].

Fig. 11(a) shows the resource utilization of UC2 with Design 1. Average GPU utilization is 97% and it is reached in about 13.75 s, showing that the issues with GPU execution observed for UC1 were addressed. Nonetheless, average CPU utilization is only 1% with large amount of time spent throttling T_2^{UC2} executions. This is explained by the distribution of T_2^{UC2} execution time and the capabilities of the RTS. The mean execution time of T_2^{UC2} is 1.1 s and the task scheduler of the RTS is not able to sustain the throughput required to use the available cores. While the output of T_1^{UC2} tasks accumulates, T_2^{UC2} tasks wait in the scheduler and executor queues of the RTS.

UC1 does not suffer from the same scheduling limitations of UC2 for Design 1. The mean execution time of T_2^{UC1} is 194 s, requiring much less throughput from the scheduler of the RTS. Further, the mean execution time of T_1^{UC1} is 85 s instead of the 6 s of T_1^{UC2}. This produces a much lower output rate for T_1^{UC1} than that of T_2^{UC1}. In turn, the RTS scheduler has fewer tasks per unit of time to schedule. Overall, we can conclude that for tasks with less than 1 min execution time, the overheads of scheduling and setting up the execution of a task become dominant in the RTS we utilized.

Fig. 10(b) shows resource utilization for UC1 with Design 2. GPUs are utilized almost immediately as images are becoming available in the queues between T_1^{UC1} and T_2^{UC1}. This quickly leads to fully utilized resources. CPU utilization is larger compared to Design 1, which is expected due to the longer execution times measured. In addition, two GPUs are used for more than 20,000 s compared to other GPUs. This shows that the additional execution

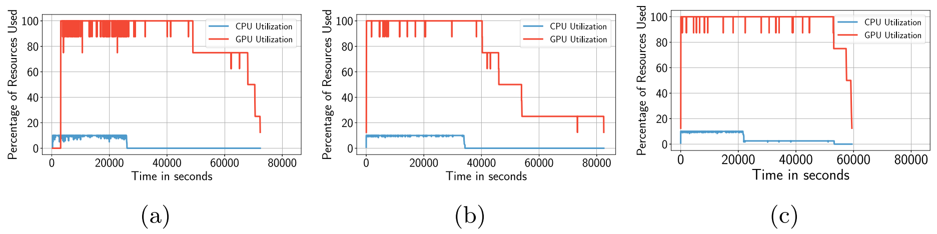


Fig. 10. Experiment 2, UC1 Percentage of CPU and GPU utilization for: (a) Design 1; (b) Design 2, and (3) Design 2.A.

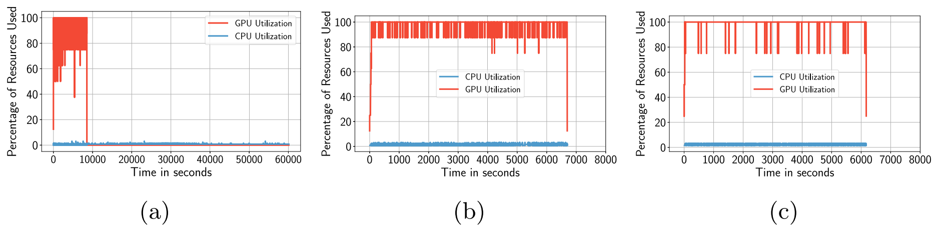


Fig. 11. Experiment 2, UC2 Percentage of CPU and GPU utilization for: (a) Design 1; (b) Design 2, and (3) Design 2.A.

time of that node was only due to the data size and not due to idle resource time.

Fig. 11(b) shows resource utilization for UC2 with Design 2. Compared to Design 1, average GPU utilization improves from 97% to 99% but, more relevantly, average CPUs utilization grows to 1.35% with almost no throttling. The use of queues, the early binding of data and the pinning of tasks to nodes, all contribute to reduce the need for throughput in the RTS scheduler. As a result, total execution time decreases from the 61,310 s of Design 1 to the 6,704 s of Design 2.

Figs. 10(c) and 11(c) show that, in Design 2.A, GPUs are released faster compared to Design 1 and Design 2. This leads to a GPU utilization above 90% for both use cases. As already explained in Experiment 1, this is due to differences in data balancing among designs. Two design choices are effective for the concurrent execution of data-driven, compute-intensive and heterogeneous workflows: (1) early binding of data to node with balanced distribution of image size; and (2) the use of local filesystems for data sharing among tasks.

Drops in resource utilization are observed in all three designs. In Design 1, although both CPUs and GPUs were used, in some cases CPU utilization dropped to 6 cores for UC1 and to 3 GPUs for UC2. Our analysis showed that this occurred when RADICAL-Pilot scheduled both CPU and GPU tasks, pointing to an inefficiency in the scheduler implementation. Design 2 and 2.A CPU utilization drops mostly by one CPU when multiple tasks try to pull from the queue at the same time. This confirms our conclusions in Experiment 1 about resource competition between middleware and executing tasks. In all designs, there is no significant fluctuations in GPU utilization, although there are more often in Design 1 when CPU and GPUs are used concurrently.

6.5. Experiment 3: Implementation overheads

Experiment 3 studies how the total execution time of our use cases workflow varies across Design 1, 2 and 2.A. Fig. 12(a) shows that Design 1 and 2 for UC1 have similar total time to execution within error bars, while Design 2.A is the fastest by a small margin. Fig. 13(a) shows that, for UC2, Design 1 total time to execution is around three times longer than the one of Design 2 and Design 2.A, while Design 2 and Design 2.A have similar durations. The discussion in Sections 6.1 and 6.2 explains how these differences relate to the execution time differences of tasks T_1 and T_2, and execution concurrency.

Figs. 12(b) and 13(b) show the overheads of each design implementation. For UC1, all three designs overheads are at least two orders of magnitude smaller than the total time to execution. A common overheads among the three designs is the "Dataset Discovery Overhead". This overhead is the time needed to list the dataset and it is proportional to the size of the dataset. RADICAL-Pilot has two main components: Agent and Client. RADICAL-Pilot Agent's overhead is less than a second in all designs while RADICAL-Pilot Client's overhead is in the order of seconds for all three designs. The latter overhead is proportional to the number of tasks submitted simultaneously to RADICAL-Pilot Agent.

EnTK's overhead in Design 1 includes the time to: (1) create the workflow consisting of independent pipelines; (2) start EnTK's components; and (3) submit the tasks that are ready to be executed to RADICAL-Pilot. This overhead is proportional to the number of tasks in the first stage of a pipeline, and the number of pipelines in the workflow. EnTK does not currently support partial workflow submission, which would allow us to submit the minimum number of tasks to fully utilize the resources before submitting the rest.

The dominant overhead of Design 2 is "Design 2 Setup Overhead" (Fig. 12(b)). This overhead includes setting up and starting queues, and starting and shutting down both tasks T_1^{UC1} and T_2^{UC1} on each compute node. Setting up and starting the queues accounts for most of the overhead as we use a conservative waiting time to assure that all the queues are up and ready. This can be optimized further, reducing the impact of this overhead. Design 2.A introduces an overhead called "Design 2.A Distributing Overhead" when partitioning and distributing the dataset over separate nodes. The average time of this overhead for UC1 is 7.5 s, with a standard deviation of 3.71 and is proportional to the dataset and the number of available compute nodes.

Compared to UC1, Fig. 13(b) shows a different composition of overheads for UC2. The overheads of Design 1 account for most of the execution time showed in Fig. 11(a). EnTK and RP Client/Agent overheads are all very large, indicating that RP spends most of the time scheduling, launching and unscheduling tasks, possibly with very large I/O overheads due to the high frequency of reading/writing to a shared file system. EnTK takes a long time waiting for the data required to describe the full workflow and more time waiting for the tasks to be handled by the RTS. We measured also high latency between the RTS and the external MongoDB instance used by EnTK and RTS to communicate task descriptions and state

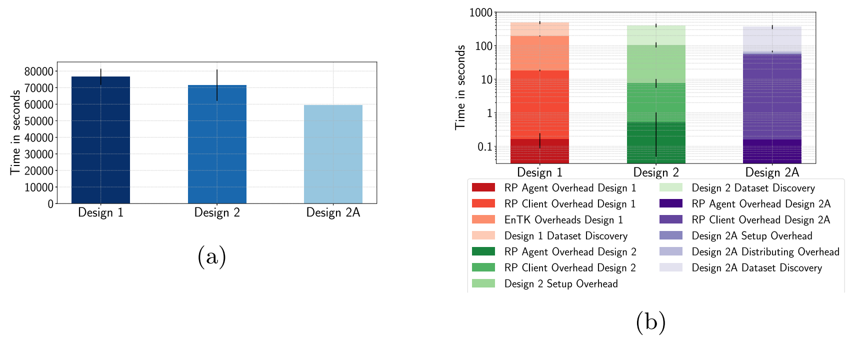


Fig. 12. Experiment 3, (a) UC1 total execution time of Design 1, 2 and 2.A. (b) Overheads of Design 1, 2 and 2.A are at least two orders of magnitude less than the total execution time.

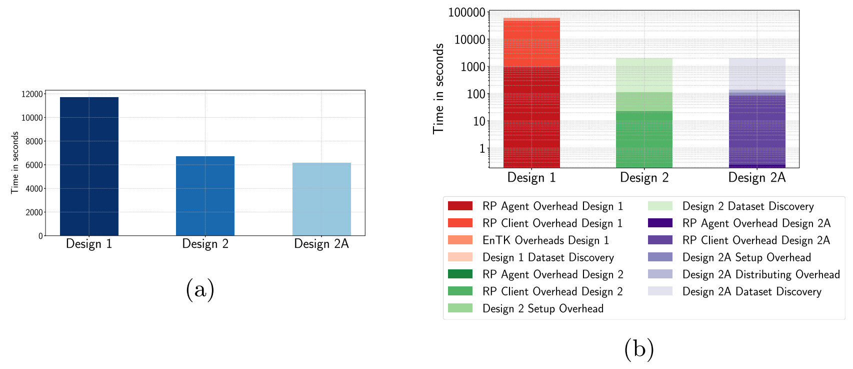


Fig. 13. Experiment 3, (a) UC2 total execution time of Design 1, 2 and 2.A. (b) Overheads of Design 1, 2 and 2.A.

updates. Overall, Design 1 is proven to be unfeasible for UC2 with the middleware used for our experiments.

Designs 2 and 2.A show much lower overheads than Design 1 for UC2. Note that the sum of RP and Setup overheads are comparable to those of UC1 but with a slightly different distribution across overheads components. For UC2, RP Agent overhead is smaller, possibly due to the improvements made to the RTS task scheduler. "Dataset Discovery" overhead is larger for UC2 compared to UC1 due to the larger dataset used and the need to form pairs.

In general, Design 2.A offers the best and more stable performance, in terms of overheads, resource utilization, load balancing and total time to execution. Although Design 2 has similar overheads, even assuming minimization of Setup Overhead, it does not guarantee load balancing as done by Design 2.A. Design 1 involves independent pipelines that are concurrently executed by the RTS on any available resource, leading to the described overheads. Based on the results of our analysis, these overheads could be reduced in both EnTK and RADICAL-Pilot by adopting early binding of images to each compute node as done in Design 2.A. Nonetheless, Design 1 would still require executing a task for each image, imposing bootstrap and tear down overheads for each task.

7. Conclusions

While Design 1, 2 and 2.A can successfully support the execution of the use cases described in Section 3, our experiments show that for the metrics considered, Design 2.A is the one that offers the better overall performance. Generalizing this result, use cases that are both data-driven and compute-intensive benefit from early binding of data to compute nodes so as to maximize data and compute affinity, and equally balance input data across nodes. Design 2.A minimizes the overall time to completion of this type of workflow while maximizing resource utilization.

Our analysis also shows the limits of an approach where pipelines, i.e., interdependent compute tasks, are late bound to compute nodes. In designs in which tasks are independent executables (i.e., programs), the overhead of bootstrapping a program needs to be minimized, ensuring that each pipeline processes as much input as possible (in our use case, single and pairs of images). In presence of large amount of data, late binding implies copying, replicating or accessing data over network and at runtime. We showed that, in contemporary HPC infrastructures, this is too costly both for resource utilization and total time to completion. Even when data are made available on the network filesystems of the HPC infrastructure, the time spent to access and/or write those amounts of data at runtime dominates the total time to completion of the application workflow, vastly reducing the amount of time computing resources can be used while available.

It should be noted that our insight does not depend on the middleware we used for our experiments, or on the type of data and computation that our use cases required. Our insight depends instead on the requirements of the given tasks and how the capabilities of the available resources satisfy those requirements. Given the ratio between CPUs and GPUs, the amount of memory per node and the filesystem performance in our experiments, Design 2.A will perform better than the other two designs for any use case that requires the analysis of multi-terabyte dataset with both CPUs and GPUs. Conversely, given a resource with a sufficiently fast filesystem and a 1:1 ratio between CPUs and GPUs, based on our analysis, all three designs will perform analogously, possibly with slightly different overhead distributions.

Infrastructure-wise, the experiments presented in Section 6 show the limits imposed by an imbalance between number of CPU cores and available memory. Given data-driven computation where multi GB images need concurrent processing, we were able to use just 10% of the available cores due to the amount of RAM required by each image processing. This applies also to the imbalance between CPUs and GPUs: use cases with heterogeneous tasks would benefit from a higher GPU/CPU ratio. Finally, filesystem performance limited the amount of concurrent I/O we could perform from concurrent processes. This is consistent with the current trend of building HPC infrastructures with higher GPU density per node, with different types of dedicated memories and multi-tiered data systems. ORNL Summit or TACC Frontera are contemporary examples of such a trend.

Sections 4 and 6 also show the limits of optimizing the executable of a task when multiple instances of that executable have to be executed concurrently. While GPU-SIFT largely improves on the computing efficiency of preexisting SIFT implementations, the amount of memory it required to match an image pair always depend on the size of the images of the pair. This imposes a limit on the number of GPU-SIFT tasks that can be concurrently executed. This make the use of dedicated accelerators preferable to the use of general purpose processors, but also shows the importance of optimizing the design of the middleware that has to execute those program instances concurrently.

Section 6 offers an example of a methodology for experimentally evaluating the performance of alternative but functionally equivalent middleware designs that support the execution of data and compute intensive workflow applications on HPC machines. This methodology is important to drive the development of middleware in a moment in which application workflows have become fundamental for many scientific domains [47] and academic efforts are multiplying to support such applications 5. While qualitative metrics like usability, security or portability are fundamental, to the best of our knowledge, we are lacking quantitative ways to compare alternative middleware designs for specific production infrastructures (i.e., experimental methodologies). Consistently, our methodology focuses on three quantitative performance metrics (total time to completion, resource utilization and middleware overheads) which measure the speed and efficiency with which users can obtain results and how well resources that have been “paid” for can be utilized.

The results presented open several future lines of research. We will extend both EnTK and RADICAL-Pilot to implement Design 2.A. We will use our characterization of overheads as a baseline to evaluate our implementations and further improve the efficiency of our middleware. Further, we will apply the presented experimental methodology to additional use cases and infrastructures, measuring the trade offs imposed by other types of task heterogeneity, including multi-core or multi-GPU tasks that extend beyond a single compute node. We will explore how the presented methodology applies to designs in which tasks are not independent programs but, instead, single functions or methods. In general, we will study how to evaluate the trade offs between in-memory and filesystem-based computations when use cases demand the maximization of concurrent execution. This will be particularly important to evaluate the execution of workflow applications on the upcoming exascale HPC infrastructures.

Beyond design, methodological and implementation insights, the work for this paper has already enabled the execution of use cases at unprecedented scale and speed. The 3097 images of the Seals use case can be analyzed in ≈20 hours, and the 11,030 image pairs of the Geolocation use case can be matched in ≈5.6 hours, compared to labor-intensive weeks previously required on non-HPC resources. These are by no means optimal results. The capabilities we will develop for our middleware based on the insight gained with the results presented in this paper, will allow for further improvement of execution performance. In this context, it will be important to integrate analytical models of optimal execution with the analysis of executions on actual computing infrastructures.

Data sources, the software used for their analysis and replication guidelines can be found at [1,48].

CRediT authorship contribution statement

Aymen Al-Saadi: Contributed equally to all section of the paper. **Ioannis Paraskevakos:** Contributed equally to all section of the paper. **Bento Collares Gonçalves:** Contributed to section 3. **Matteo Turilli:** Contributed equally to all section of the paper.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments and contributions

We thank Andre Merzky (Rutgers) and Brad Spitzbart (Stony Brook) for useful discussions. This work is funded by NSF EarthCube Award, United States of America Number 1740572. Computational resources were provided by NSF XRAC awards, United States of America TG-MCB090174. Geospatial support for this work provided by the Polar Geospatial Center, United States of America under NSF-OPP awards 1043681 and 1559691. We thank the PSC Bridges PI and Support Staff for supporting this work through resource reservations. Aymen Alsaadi, Ioannis Paraskevakos and Matteo Turilli contributed equally to all section of the paper. Bento Collares Gonçalves contributed to Section 3.

References

[1] I. Paraskevakos, M. Turilli, B.C. Gonçalves, H. Lynch, S. Jha, Workflow design analysis for high resolution satellite image analysis, in: 2019 15th International Conference on EScience (EScience), 2019, pp. 47–56.

[2] J. Dean, S. Ghemawat, MapReduce: a flexible data processing tool, Commun. ACM 53 (1) (2010) 72–77.

[3] H. Vo, J. Kong, D. Teng, Y. Liang, A. Aji, G. Teodoro, F. Wang, Mareia: a cloud mapreduce based high performance whole slide image analysis framework, Distrib. Parallel Databases (2018) http://dx.doi.org/10.1007/s10619-018-7237-1.

[4] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, in: HotCloud’10, USENIX Association, Berkeley, CA, USA, 2010, p. 10.

[5] Z. Zhang, K. Barbary, F.A. Nothaft, E.R. Sparks, O. Zahn, M.J. Franklin, D.A. Patterson, S. Perlmutter, Kira: Processing astronomy imagery using big data technology, IEEE Trans. Big Data (2016) 1, http://dx.doi.org/10.1109/TBDATA.2016.2599926, http://ieeexplore.ieee.org/document/7549106/.

[6] Y. Yan, L. Huang, Large-Scale Image Processing Research Cloud, 2014, http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=E70A37090C2094525DF8F818154EE7D3?doi=10.1.1.684.6787{&}rep=rep1{&}type=pdf.

[7] R. Ramos-Pollan, F.A. Gonzalez, J.C. Caicedo, A. Cruz-Roa, J.E. Camargo, J.A. Vanegas, S.A. Perez, J. David Bermeo, J.S. Otalora, P.K. Rozo, J.E. Arevalo, BIGS: A framework for large-scale image processing and analysis over distributed and heterogeneous computing resources, in: 2012 IEEE 8th International Conference on E-Science, IEEE, 2012, pp. 1–8, http://dx.doi.org/10.1109/eScience.2012.6404424, http://ieeexplore.ieee.org/document/6404424/.

[8] D.E.J. Hobley, J.M. Adams, S.S. Nudurupati, E.W.H. Hutton, N.M. Gasparini, G.E. Tucker, Creative computing with landlab: an open-source toolkit for building, coupling, and exploring two-dimensional numerical models of Earth-surface dynamics, Earth Surf. Dynam 5 (2017) 21–46, http://dx.doi.org/10.5194/esurf-5-21-2017, www.earth-surf-dynam.net/5/21/2017/.

[9] J. Borrill, C. Bischoff, T. Crawford, M. Hasselfield, R. Keskitalo, T. Kisner, A. Kusaka, N. Whitehorn, HPC / HTC Software Infrastructure for the Synthesis and Analysis of CMB Datasets, 2020, http://dx.doi.org/10.6084/m9.figshare.11821071.v2, https://figshare.com/articles/HPC_HTC_Software_Infrastructure_for_the_Synthesis_and_Analysis_of_CMB_Datasets/11821071.

[10] B.S. Manjunath, T. Pollock, R. Miller, N. Merchant, A. Roy-Chowdhury, LIMPID: Large-scale Image Processing Infrastructure, 2018, http://dx.doi.org/10.6084/m9.figshare.6169337.v1, https://figshare.com/articles/LIMPID_Large-scale_Image_Processing_Infrastructure/6169337.

[11] A. Galizia, D. D'Agostino, A. Clematis, An MPI–CUDA library for image processing on HPC architectures, J. Comput. Appl. Math. 273 (2015) 414–427, http://dx.doi.org/10.1016/J.CAM.2014.05.004, https://www.sciencedirect.com/science/article/pii/S0377042714002374.

[12] A. Gholami, A. Mang, K. Scheufele, C. Davatzikos, M. Mehl, G. Biros, A framework for scalable biophysics-based image analysis, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, in: SC '17, ACM, New York, NY, USA, 2017, pp. 19:1–19:13, http://dx.doi.org/10.1145/3126908.3126930.

[13] Y. Huo, J. Blaber, S.M. Damon, B.D. Boyd, S. Bao, P. Parvathaneni, C.B. Noguera, S. Chaganti, V. Nath, J.M. Greer, I. Lyu, W.R. French, A.T. Newton, B.P. Rogers, B.A. Landman, Towards portable large-scale image processing with high-performance computing, J. Digital Imaging 31 (3) (2018) 304–314, http://dx.doi.org/10.1007/s10278-018-0080-0.

[14] S.M. Damon, B.D. Boyd, A.J. Plassard, W. Taylor, B.A. Landman, DAX - The next generation: Towards one million processes on commodity hardware, Proceedings of SPIE–the International Society for Optical Engineering 2017 (2017) http://dx.doi.org/10.1117/12.2254371, http://www.ncbi.nlm.nih.gov/pubmed/28919661, http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5596878.

[15] R. Vescovi, M. Du, V. de Andrade, W. Scullin, G. Doğa, C. Jacobsen, Tomosaic: efficient acquisition and reconstruction of teravoxel tomography data using limited-size synchrotron X-ray beams, J. Synchrotron Radiat. 25 (2018) 1478–1489, http://dx.doi.org/10.1107/S1600577518010093.

[16] S. Petruzza, A. Venkat, A. Gyulassy, G. Scorzelli, F. Federer, A. Angelucci, V. Pascucci, P.-T. Bremer, ISAVS: Interactive scalable analysis and visualization system, in: SIGGRAPH Asia 2017 Symposium on Visualization, in: SA '17, ACM, New York, NY, USA, 2017, pp. 18:1–18:8, http://dx.doi.org/10.1145/3139295.3139299, http://doi.acm.org/10.1145/3139295.3139299.

[17] G. Teodoro, T. Pan, T.M. Kurc, J. Kong, L.A. Cooper, N. Podhorszki, S. Klasky, J.H. Saltz, High-throughput analysis of large microscopy image datasets on CPU-gpu cluster platforms, in: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IEEE, 2013, pp. 103–114, http://dx.doi.org/10.1109/IPDPS.2013.11, http://ieeexplore.ieee.org/document/6569804/.

[18] R. Grunzke, F. Jug, B. Schuller, R. Jäkel, G. Myers, W.E. Nagel, Seamless HPC integration of data-intensive KNIME workflows via UNICORE, Springer, Cham, 2017, pp. 480–491, http://dx.doi.org/10.1007/978-3-319-58943-5_39, http://link.springer.com/10.1007/978-3-319-58943-5{_}39.

[19] K. Benedyczak, B. Schuller, M. Petrova-El Sayed, J. Rybicki, R. Grunzke, UNICORE 7 — Middleware services for distributed and federated computing, in: 2016 International Conference on High Performance Computing & Simulation (HPCS), IEEE, 2016, pp. 613–620, http://dx.doi.org/10.1109/HPCSim.2016.7568392, http://ieeexplore.ieee.org/document/7568392/.

[20] N. Ghouaiel, S. Lefèvre, Coupling ground-level panoramas and aerial imagery for change detection, Geo-spatial Inf. Sci. 19 (2016) 222–232.

[21] N. Vo, N. Jacobs, J. Hays, Revisiting IM2GPS in the deep learning era, 2017, arXiv:1705.04838.

[22] K.U. Karanth, Estimating tiger panthera tigris populations from camera-trap data using capture—recapture models, Biol. Cons. 71 (3) (1995) 333–338.

[23] D. Western, R. Groom, J. Worden, The impact of subdivision and sedentarization of pastoral lands on wildlife in an African savanna ecosystem, Biol. Cons. 142 (11) (2009) 2538–2546.

[24] H.J. Lynch, R. White, A.D. Black, R. Naveen, Detection, differentiation, and abundance estimation of penguin species by high-resolution satellite imagery, Polar Biol. 35 (6) (2012) 963–968, http://dx.doi.org/10.1007/s00300-011-1138-3.

[25] B. Kellenberger, D. Marcos, D. Tuia, Detecting mammals in UAV images: Best practices to address a substantially imbalanced dataset with deep learning, Remote Sens. Environ. 216 (2018) 139–153, http://dx.doi.org/10.1016/j.rse.2018.06.028, http://www.sciencedirect.com/science/article/pii/S0034425718303067.

[26] A. Polzounov, I. Terpugova, D. Skiparis, A. Mihai, Right whale recognition using convolutional neural networks, 2016, CoRR, abs/1604.05605, arXiv:1604.05605.

[27] M.S. Norouzzadeh, A. Nguyen, M. Kosmala, A. Swanson, M.S. Palmer, C. Packer, J. Clune, Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning, Proc. Natl. Acad. Sci. 115 (25) (2018) E5716–E5725, http://dx.doi.org/10.1073/pnas.1719367115, arXiv:115/25/E5716, https://www.pnas.org/content/115/25/E5716.

[28] A. Fabra, V. Gascón, The convention on the conservation of antarctic marine living resources (CCAMLR) and the ecosystem approach, Int. J. Mar. Coast. Law 23 (3) (2008) 567–598.

[29] H. Hillebrand, T. Brey, J. Gutt, W. Hagen, K. Metfies, B. Meyer, A. Lewandowska, Climate change: Warming impacts on marine biodiversity, in: Handbook on Marine Environment Protection, Springer, 2018, pp. 353–373.

[30] K. Reid, Climate change impacts, vulnerabilities and adaptations: Southern ocean marine fisheries, Impacts Clim. Chang. Fish. Aquac. (2019) 363.

[31] O. Ronneberger, P. Fischer, T. Brox, U-net: Convolutional networks for biomedical image segmentation, in: International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2015, pp. 234–241.

[32] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2014, arXiv preprint arXiv:1412.6980.

[33] D.G. Lowe, Distinctive image features from scale-invariant keypoints, Int. J. Comput. Vis. 60 (2) (2004) 91–110, http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94.

[34] R.C. Bolles, M.A. Fischler, A RANSAC-based approach to model fitting and its application to finding cylinders in range data., in: IJCAI, 1981, 1981, pp. 637–643.

[35] M. Rodriguezs, et al., Fast Image Matching by Affine Simulation, CMLA, 2017–, https://github.com/rdguez-mariano/fast_imas_IPOL.

[36] M. Björkman, N. Bergström, D. Kragic, Detecting, segmenting and tracking unknown objects using multi-label MRF inference, Comput. Vis. Image Underst. 118 (2014) 111–127, http://dx.doi.org/10.1016/j.cviu.2013.10.007, http://www.sciencedirect.com/science/article/pii/S107731421300194X.

[37] A.-S. Aymen, Scalable algorithm and workload execution for geo locating satellite imagery, 2020, http://dx.doi.org/10.7282/t3-z0w1-sg59, https://rucore.libraries.rutgers.edu/rutgers-lib/62916/.

[38] N.A. Nystrom, M.J. Levine, R.Z. Roskies, J.R. Scott, Bridges: A uniquely flexible HPC resource for new communities and data analytics, in: Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled By Enhanced Cyberinfrastructure, in: XSEDE '15, ACM, New York, NY, USA, 2015, pp. 30:1–30:8, http://dx.doi.org/10.1145/2792745.2792775.

[39] A. Merzky, M. Turilli, M. Maldonado, M. Santcroos, S. Jha, Using pilot systems to execute many task workloads on supercomputers, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2018, pp. 61–82.

[40] L. Tu, C. Dong, Histogram equalization and image feature matching, in: 2013 6th International Congress on Image and Signal Processing (CISP), Vol. 01, 2013, pp. 443–447.

[41] K. Lin, J. Lu, C. Chen, J. Zhou, Learning compact binary descriptors with unsupervised deep neural networks, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 1183–1192, http://dx.doi.org/10.1109/CVPR.2016.133.

[42] L. Zheng, Y. Yang, Q. Tian, SIFT Meets CNN: A decade survey of instance retrieval, IEEE Trans. Pattern Anal. Mach. Intell. 40 (5) (2018) 1224–1244, http://dx.doi.org/10.1109/TPAMI.2017.2709749.

[43] G. Bradski, The opencv library, Dr. Dobb's J. Softw. Tool (2000).

[44] S. Gillies, et al., Rasterio: Geospatial Raster I/O for Python Programmers, Mapbox, 2013–, https://github.com/mapbox/rasterio.

[45] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in PyTorch, in: NIPS-W, 2017.

[46] V. Balasubramanian, M. Turilli, W. Hu, M. Lefebvre, W. Lei, R. Modrak, G. Cervone, J. Tromp, S. Jha, Harnessing the power of many: Extensible toolkit for scalable ensemble applications, in: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2018, pp. 536–545.

[47] T. McPhillips, S. Bowers, D. Zinn, B. Ludäscher, Scientific workflow design for mere mortals, Future Gener. Comput. Syst. 25 (5) (2009) 541–551.

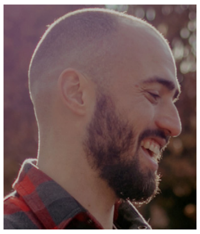
[48] ICEBERG Team, Imagery cyber-infrastructure and extensible building blocks to enhance geosciences research, 2018, [Online; accessed April 25th 2019], https://iceberg-project.github.io/.

Aymen Al-Saadi is a Ph.D. student and Junior Research Developer of the RADICAL group. Before that he earned his master degree in Computer Engineering from Rutgers University. His research focus currently lies on high throughput function execution in HPC contexts.

Ioannis Paraskevakos received his Ph.D. from Rutgers University.

His research focused on providing data-intensive capabilities on HPC for analyzing data from scientific simulations, through abstraction, architectures and middleware. His research interests include High Performance Computing, Task Parallel Applications and Architectures, Distributed System Communication Protocols, and Distributed Data Abstractions.

During his studies, he was involved in two projects, MIddleware for Data-Intensive Analytics and Science (MIDAS) and Imagery Cyberinfrastructure and Extensible Building-Blocks to Enhance Research in the Geosciences (ICEBERG). In addition, he regularly contributes to several RADICAL-Cybertools projects.



Bento is a biologist by training, with expertise in remote sensing and computer vision algorithms and a wide variety of machine learning and statistical methods in his toolbox. Fueled by intellectual challenge and wanting to go beyond observation and simple statistics, Bento picked up programming by himself and developed an interest in Machine Learning, which soon became his bread and butter through his quantitative ecology Ph.D. A pioneer in using deep learning in Ecology, Bento designed Convolution Neural Network architectures to automatically detect seals and penguin colonies in high-resolution satellite-imagery, enabling population estimates at unprecedented spatial and temporal scales.

Heather Lynch is the IACS Endowed Chair for Ecology & Evolution at Stony Brook University in Stony Brook, NY, USA. Her research focuses on the population dynamics of Antarctic wildlife and the development of novel wildlife survey methods using computer vision, UAVs, and satellite imagery. Lynch has a A.B. in Physics from Princeton University, an M.A. in Physics from Harvard University, and a Ph.D. in Organismic and Evolutionary Biology from Harvard University. Contact her at heather.lynch@stonybrook.edu.

Shantenu is an Associate Professor of Computer Engineering at Rutgers University and the Chair of the Department (Center) for Data Driven Discovery at Brookhaven National Laboratory. He was appointed a Rutgers Chancellor's Scholar in 2015. He has held visiting positions at the University of Edinburgh and UCL.

Shantenu's research interests are at the intersection of high-performance distributed computing and computational & data-driven science. He is the PI of RADICAL Lab and the lead investigator of RADICAL-Cybertools project which are a suite of middleware building blocks used to support large-scale science and engineering applications. He is proud to play a part in the upcoming revolution at the interface of computing and health-science—global health and "personalized" medicine. He collaborates extensively with scientists from multiple domains—including but not limited to Molecular Sciences, Earth Sciences and High-Energy Physics.

Shantenu was the recipient of the inaugural Chancellor's Excellence in Research (2016) for his cyberinfrastructure contributions to computational science. He was also awarded a Rutgers Board of Trustees Fellowship for Scholarly Excellence (2014). He is a recipient of the NSF CAREER Award (2013) and several best paper prizes at SC'xy and ISC'xy. His current research has been funded by multiple NSF awards and US Department of Energy (DoE); his work has also been funded by US National Institute for Health (NIH), and the UK EPSRC.

Matteo Turilli I am Assistant Research Professor at the department of Electrical and Computer Engineering, Rutgers University. Before moving to Rutgers, I was the Chair of the EGI Federated Cloud and Senior Research Associate at the Oxford e-Research Centre (OeRC), University of Oxford, UK. My research interests primarily concern bridging the gap between distributed and high performance computing, designing middleware for scientific cyberinfrastructures, and supporting domain-specific scientific workflows. I collaborate with several domain scientists to enable research in biological sciences, earth sciences, climate sciences and particle physics. For my research, I use production infrastructures like OLCF, NCSA, NCAR and XSEDE, performing large-scale experiments on among the biggest "supercomputers" currently available. Highly motivated students with similar research interests and looking for joining a Master or Ph.D. program are always welcome to get in contact.