RAI: A Scalable Project Submission System for Parallel Programming Courses

Abdul Dakkak*, Carl Pearson†, Cheng Li*, and Wen-mei Hwu†
*Department of Computer Science
†Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, USA
{dakkak, pearson, cli99, w-hwu}@illinois.edu

***Abstract*—A major component of many advanced programming courses is an open-ended "end-of-term project" assignment. Delivering and evaluating open-ended parallel programming projects for hundreds or thousands of students brings a need for broad system reconfigurability coupled with challenges of testing and development uniformity, access to esoteric hardware and programming environments, scalability, and security. We present RAI, a secure and extensible system for delivering open-ended programming assignments configured with access to different hardware and software requirements. We describe how the system was used to deliver a programming-competition-style final project in an introductory GPU programming course at the University of Illinois Urbana-Champaign.**

***Keywords*-GPU; CUDA; OpenCL; OpenACC; massive open online courses; programming education; online education**

I. Introduction

Many programming classes have an open-ended project-style final assignment where students engage with a large-scale, realistic or otherwise more significant applications of course content. Typical goals of such class projects are to expose students to more practical processes and challenges of programming, including experience in manipulating ancillary tools like compilers, profilers, and debuggers. In contrast with weekly lab assignments, which have a focused design and narrow pedagogical goals, the end-term project typically presents a more open-ended task which requires 5 to 8 weeks for teams of 2 to 4 students to complete.

The "Applied Parallel Programming" (ECE408/CS598) course at the University of Illinois Urbana-Champaign ends with such a project. Applied Parallel Programming is an introductory junior/senior level course that starts with eight weekly labs, followed by a 5-week programming project. During the fall of 2016, the open-ended project task was for student teams to write a high-performance CUDA implementation of a convolutional neural network inference step.

In early offerings of the course, students were provided with remote access to a batch submission system on a small GPU cluster hosted at the University of Illinois.

This work was supported by the Starnet Center for Future Architecture Research (C-FAR), NVIDIA GPU Center of Excellence at UIUC, the IBM-Illinois Center for Cognitive Computing Systems Research (C3SR), and the NSF Blue Waters Award (0725070).

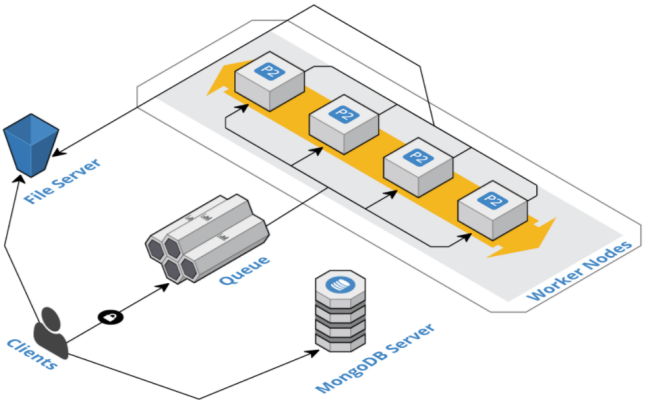

Figure 1. The RAI system architecture is composed of a client interacting with a message broker (queue). Many Amazon P2 instances (NVIDIA K80 GPUs) subscribe to the queue and accept jobs. The infrastructure relies upon on a file server (Amazon S3) as well as a MongoDB database.

As course enrollment grew and the Heterogeneous Parallel Programming course on Coursera was developed with the same material, challenges of scalability, security, and automatic grading led to the development and continued use of WebGPU [1] for the weekly labs. WebGPU is a scalable, secure system which allows students to write and execute code through a web interface. WebGPU hides some system configuration options and more advanced profiling and debugging tools to emphasize the educational objectives of the labs. WebGPU allows the weekly lab component of the course to scale to thousands of remote students on Coursera. The remaining key course infrastructure challenge became delivering a configurable, secure, flexible remote programming environment to scale the project component of the course.

This paper describes RAI [1], an open-source project-submission system designed as a configurable programming environment for parallel programming courses. RAI is an interactive command line tool used for project job submissions. Students specify steps to build and run their project; the project is then deployed to a worker node and run

[1]http://www.rai-project.com

within a sandboxed container. This paper also describes the experience of using RAI to administer a performance oriented competition offered during the fall of 2016 for the Applied Parallel Programming at the University of Illinois Urbana-Champaign.

The rest of this paper is organized as follows. Section II describes the motivation for developing the project submission system. Section III discusses related works. These systems include traditional batch-based submission system backed by a local cluster as well as continuous-integration infrastructures. Section IV describes the system architecture of RAI, detailing the components of the system, and shows how both security and scalability follow from the system's architectural choices. Section V describes how the students interact with RAI to submit their projects. Section VI describes the instructor and student utilities that can be used to interact and query the system. Section VII outlines the lessons learned during the development, deployment, and distribution of the tools. The section also details the challenges in administering the project as well as the current limitations of the system. Finally, section VIII concludes.

II. Motivation

Since students have more freedom to make development choices during a course project, the development environment must be correspondingly more flexible. This is often at odds with project evaluation, which must be done in a uniform way to ensure fair outcomes for students. The number and remote nature of students also require non-traditional approaches and extra attention to security and isolation.

Accessibility and Scalability

Even in the best circumstances, outfitting students with individualized access to different software stacks (e.g. CUDA, OpenCL, OpenACC, MPI, high-level synthesis toolchains, profilers, debuggers) and hardware (e.g. many-core CPUs, GPUs, DSPs, FPGAs, high-performance interconnects, distributed-memory systems) is inconvenient or impossible. For example, 70% of the 176 students in the fall 2016 Applied Parallel Programming course did not have access to a CUDA-programmable GPU. Consequently, appropriate development environments must be provided to students as part of the course. When coupled with the growing trend of delivering post-secondary education through web-based massive open online courses (MOOCs), the disparate student locations demand that access to programming environments with these capabilities to be offered over the internet. Furthermore, the sheer number of students only compounds any challenges. Any solution must scale to thousands of concurrent users. If even a small fraction of students face configuration problems, it will quickly overwhelm any course staff.

System Configurability and Evaluation Uniformity

The motivation for parallel programming is to achieve high performance. In pursuit of performance, students may be required to perform major code refactoring activities (e.g. replacing algorithms, performing program transformations, or picking constants such as unroll or tiling factors), modify build systems, and interact with profiling and debugging tools. The programming environment must therefore provide nearly the same capabilities and flexibility as full system access. However, for the teaching staff to efficiently evaluate projects, there must be uniformity in the development and testing process.

Security and Isolation

The flexibility of the project development system also introduces challenges from a system administration standpoint, since such flexibility poses a security concern. Even in a traditional educational environment where structures are in place to moderate student behavior, academic integrity, and system stability demand that student development and testing are isolated from other students. When remote students have a less formal relationship with an educational institution, the programming environment must be substantially hardened against bad behavior. Currently, there is a lack of scalable systems that allow instructors to assign projects and give students complete control of a system within a confined environment.

Maintainability and Cost

When a parallel programming course requires uncommon hardware (GPUs and FPGAs) and software (CUDA, high-level synthesis tools) it may be cost-prohibitive to outfit uniform labs and clusters. Some universities and classrooms address these challenges by mandating students to have the hardware/software resources as a course requirement. For anything other than a small class, the overhead of managing minor variations in student environments becomes intractable. Furthermore, the general unavailability of many high-performance parallel computing systems and the diversity of the computing architectures used makes this impractical for parallel computing courses. When access must be provided to students over the internet (who may not be directly affiliated with the host institution), the institution may be unwilling to incur the costs of managing the additional users and the security and privacy challenges they bring.

III. Related Work

Some programming and submission systems address concerns mentioned in Section II, Table I breaks down the relevant features of selected programming and/or submission systems and compares them to RAI.

Table I
EXISTING PROGRAMMING AND SUBMISSION SYSTEMS CURRENTLY USED DO NOT AFFORD THE RECONFIGURABILITY, ISOLATION, SCALABILITY, ACCESSIBILITY, AND UNIFORMITY NEEDED FOR LARGE OPEN-ENDED PROGRAMMING EXERCISES.

System	Configurability	Isolation	Scalability	Accessibility	Testing Uniformity
Sudent-Provided	✓	✓	✓	×	×
Torque/PBS	✓	✓	✓	✓	×
WebGPU	×	✓	✓	✓	✓
Jenkins	✓	✓	✓	×	✓
QwikLabs	×	✓	✓	✓	×
RAI	✓	✓	✓	✓	✓

Using Locally-Hosted Resources

Traditionally, a programming course would start with a lab submission system such as WebGPU [1] and transition to a managed local university cluster for the project. These clusters are typically configured with a Torque- or PBS-like [2] batch submission system for their users. In traditional high performance cluster settings, this can be combined with tools such as Maui [3], Moab [4] and Slurm [5] to abstract away the cluster resources and schedule workloads onto the cluster. Jobs are expected to be long (longer than a few minutes) to amortize the scheduling overhead.

Such systems can give the user the full flexibility as if the environment were their own, but also come with several problems. First, the fixed resources of the local cluster can become oversubscribed during the final weeks of the semester (especially if other instructors have the same project submission strategy). During the last few days, the cluster queue can become long, causing delays and a poor experience for the students. Second, the course staff is left to develop tools and techniques for ensuring that all students are developing code following whatever the evaluation criteria will be. Finally, equipping an entire cluster with esoteric hardware for a single programming course may not be achievable.

Using Cloud and Cluster Resources

Other approaches, like WebGPU, use remote cloud resources. These systems offer a scalable and secure online development environment where students write and execute code through a web interface. These online development environments hide the system configuration options and disallow more advanced profiling and debugging tools to keep the focus on the educational objectives of each lab.

Several cloud submission systems are similar to RAI. On cloud and clusters, Apache Mesos [6] with Chronos provide an abstraction of the hardware as well as the ability to specify required resources. Continuous Integration (CI) systems can also be thought of as a job submission system – where a build and test task is submitted after each commit to the repository. Both Hudson [7] and Jenkins [8] are widely used tools which hook into source code repositories and run whenever a developer commits a change. The build configuration for both Hudson and Jenkins is specified via a GUI or XML. Other CI tools such as TravisCI and Bitbucket Pipelines are more integrated with repositories hosted on Github or Bitbucket, respectively. TravisCI and Bitbucket Pipelines are easier to use, with the configuration being done primarily via a YAML [9] file (these tools inspire the configuration for RAI). Like RAI, TravisCI and Bitbucket Pipelines execute commands within a Docker container with the user specifying the base image; executing the build steps specified by the YAML configuration file committed to the repository. Unlike RAI, however, both tools are closed source and require code to be committed before running the build. Additionally, to our knowledge, no CI tool can run GPU or FPGA code.

IV. RAI SYSTEM ARCHITECTURE

The RAI job submission system is composed of clients, workers, queues, a file server, and a database. Figure 1 shows a high-level view of the system architecture. RAI is not hard-coded to Amazon's AWS and can be used on a local cluster if desired. RAI can also be configured to scale out to remote cloud instances as local resources if exhausted.

RAI Client

The RAI client is an executable downloaded by the students and runs on the students' machines. The executable requires no library dependencies and works on all the main operating systems and CPU architectures. Both features reduce the likelihood that students will have technical difficulties running the client. Students use the RAI client to interact with the system and to submit jobs. The client can submit jobs during project development, but students can also choose a mode to make a final submission for grading. The RAI client offers other features such as checking team ranking within a competition project.

To use the client, the student must create a file that contains authentication keys that uniquely identify students or teams of students. These keys are generated by the teaching staff through a companion utility and provided to the students as described in Section VI. Section VII describes how we deliver the client to the students.

RAI Worker

The RAI worker acts as an agent that starts a sandboxed environment to executes students' code. Multiple worker nodes can exist within the RAI environment, thus making the system elastic and able to scale out. Because of current limitations of Docker GPU integration, the worker must run on a Linux system.

Message Broker

The message broker arbitrates communication between clients and workers to maintain fairness and resiliency for submissions. Workers and clients connect and subscribe to different queues on the broker. Both the RAI clients and workers subscribe to the message broker and exchange messages using a publish/subscribe communication pattern.

MongoDB Database

We use MongoDB [10], a NoSQL database, to store meta-information about submissions, including execution times, run-times, and logs. The information in this database is useful for grading or any other coursework auditing process. The database is also used to store team ranking within a project competition.

File Storage Server

When a student submits a job to RAI, their project directory gets uploaded to a file server. Upon job completion, the worker node's output directory is also uploaded to the file server and is available for the student to download. In this way, students can have access to any output or logging files their job generates. The file server is also used by the instructor to download all files tagged as final project submissions.

Files uploaded to the file server can be configured to have a particular lifetime after which they get deleted. The current lifetime is set between 1 and 3 months. We currently use Amazon's S3 [11] services as the file server.

V. RAI Submission

A student submission requires coordination between the client (running on the student's machine), the message broker, and the worker nodes. Since RAI is a distributed architecture, these operations need to happen in order and be robust to failures. This section describes how a student specifies the project build and run procedure as well as the sequence of operations that happen when a job is submitted to RAI.

Execution Specification

The student build steps are specified in YAML format in a file called `rai-build.yml` that exists in the project base directory. The build file is architected to be minimal, allowing it to be extended for future changes. We may want to specify the machine requirements (such as the number of GPUs) in the future, for example. Listing 1 shows the default `rai-build.yml` file.

The build file is split into a configuration section (lines 1-3) and a command section (lines 4-11). Within the configuration part, line 2 specifies the client version. Line 3 specifies the Docker base image used to run the commands. Students can choose from a whitelist of base images. The build commands (lines $6-11$) are the commands to be executed on the worker node. Line 6 echoes `Building project` to the terminal. Line 7 uses CMake [12] to configure the build system. CMake is used to provide a portable build system that can generate Makefile, Apple's XCode, or Microsoft's Visual Studio project build files. For the Applied Parallel Programming class, we use the Hunter [13] to manage and install the project dependencies[2]. To speed the build process, we disable Hunter and provide the dependencies as part of the base Docker image. By default, CMake generates a Makefile project, line 8 uses the generated Makefile to build the project. The generated executable is called `ece408`. Line 9 runs the `ece408` binary with the test dataset `/data/test10.hdf5` along with the pre-trained model `/data/model.hdf5`. Lines $10-11$ is a command split into multiple lines. The command runs the `ece408` executable within `nvprof` and saves the results to `timeline.nvprof`. The `/build` directory is uploaded to the file server and student can access the `timeline.nvprof` file and view it using the `nvvp` viewer.

```
1  rai:
2    version: 0.1
3    image: webgpu/rai:root
4  commands:
5    build:
6      - echo "Building project"
7      - cmake /src
8      - make
9      - ./ece408 /data/test10.hdf5
         ↪  /data/model.hdf5
10     - nvprof --export-profile timeline.nvprof --
11       ./ece408 data/test10.hdf5 /data/model.hdf5
```

Listing 1: The default YAML file used in Applied Parallel Programming when a student-written `rai-build.yaml` is not found. The above file configures the project using CMake and then uses `make` to build the project. The code is executed using the `test10` dataset – a small test dataset. It then runs `nvprof` to profile and program and stores the result into `timeline.nvprof`.

Message Broker Operations

The message broker is composed of multiple topics, each of which has multiple channels. Consumers can subscribe and enqueue messages to a given topic and a channel.We will use the notation `topic_name/channel_name` to denote the channel within a topic (also called the queue route).

By default, clients publish the job message onto `rai/tasks`. Each job is given a unique identifier. All workers listen to this channel and accept the message if they have enough resources; we place constraints on the number of jobs that can be executed concurrently. If the constraints are met, then the worker accepts the message and creates a new ephemeral topic and channel called `log_${job_id}/#ch` which the client subscribes to — both the topic and channel are deleted if there are no

[2]The project uses the HDF5 [14] format to store the neural network's model and test data files and thus depends on the `libhdf5` library.

producers and consumers. The worker sends the output of `stdout` and `stderr` along with logs, as messages to the `log_${job_id}/#ch` and the client prints them to the user's screen.

Client Execution

On the client side, i.e. on the student's machine, we perform the following steps: ❶ The client checks if the project directory specified exists on the file system and contains the `rai-build.yml` file. If the `rai-build.yml` file does not exist, then a default (shown in Listing 1) is used. ❷ The user credentials are verified. This means that both the `RAI_SERCRET_KEY` and the `RAI_ACCESS_KEY` are valid – these keys are sent by the instructor using the mechanism detailed in section VI. ❸ The project directory is compressed into a `.tar.bz2` file and uploaded to the file server. We currently use Amazon's S3, but other object file servers can be used. We configure the uploaded file so that it is deleted one month after the last use. ❹ A new job request is created and pushed onto the queue server. ❺ The client subscribes to the `log-${job_id}` topic which is where the worker will publish the `stdout` and `stderr` outputs. ❻ The client prints the messages on the `log-${job_id}` topic until the `End` message is received. ❼ If the execution is a submission, then the execution time along with the team name are recorded by the database server. ❽ The client exits once the `End` message is received.

Worker Operations

On the worker side, i.e. on the machine outfitted with a GPU, we perform the following steps: ❶ a worker subscribes to the `rai` channel on the queue. ❷ When a message is received, the worker parses the message, checks the credentials, and then extracts the `rai-build.yaml` file embedded in the job message. ❸ A Docker container starts with the base image specified by the job configuration, if the machine does not have the Docker image, then it's pulled from the Docker repository. For security, the container is configured with limited RAM and no network access. The worker mounts the `nvidia-docker` [15] CUDA volume onto the container. The worker also creates a pipe where all the output from the container's `stderr` and `stdout` are forwarded as messages to the `log-${job_id}` topic on the queue. ❹ The location of the client directory (posted on the file worker) is parsed from the job message. The worker then downloads the client directory and mounts it onto the Docker container's `/src` directory, the worker also creates a `/build` directory and sets it to be the user's working directory. ❺ The worker then starts executing the commands within the `rai-build.yaml` file in the container. ❻ After the execution is complete, the worker creates a `.tar.bz2` of the container's `/build` directory and uploads it to the file server. The URL of the uploaded `/build` directory is sent as a message to the `log-${job_id}` topic on the queue. ❼ Once the `/build` directory has been uploaded, the container is destroyed and an `End` message is sent to the queue.

```
1  rai:
2    version: 0.1
3    image: webgpu/rai:root
4  commands:
5    build:
6      - echo "Submitting project"
7      - cp -r /src /build/submission_code
8      - cmake /src
9      - make
10     - /usr/bin/time ./ece408 /data/testfull.hdf5
         ↪  /data/model.hdf5 10000
```

Listing 2: RAI enforces student final submissions to automatically use this build file. This is done to maintain uniformity across submissions and guarantee that instructors can replicate the execution while grading. The build file copies the student source code to `/build/submission_code` and uses the full dataset for evaluation.

The worker can be configured to have multiple jobs in flight. Towards the beginning of the project, when students are not utilizing the GPU, and CPU time dominates the job runtime, the worker accepts multiple jobs at the same time. In the last two weeks of the project, when students are micro-profiling and optimizing their codes, the worker accepts only one task at a time – this makes the performance timing more accurate and repeatable.

Student Final Submission

Students use the same RAI client to perform their final submission. The final submission execution has four additional requirements on top of the regular submission. ❶ Students used the `rai submit` subcommand. ❷ The submission required the presence of the `USAGE` (which contains instructions on how to run the code to get the profile results in the report) and `report.pdf` (which is the final report). ❸ The code is run using the `rai-build.yaml` file shown in Listing 2 and the student's local `rai-build.yaml` file is ignored – this is used to maintain consistency between all team submissions.

In Listing 2, line 7 performs a copy of the `/src` directory to the `/build/submission_code` directory. The file server contains the submitted code along with the build results in one archive. We record the location of the build results on the file server and use it to download all student submissions. This is described in section VI. ❹ The timing results are recorded onto the ranking database, and overwrites existing timing records. Both the results from the internal timer and the output from `/usr/bin/time` are recorded with only the internal timer visible to students. The results from the `time` command are shown to the instructors

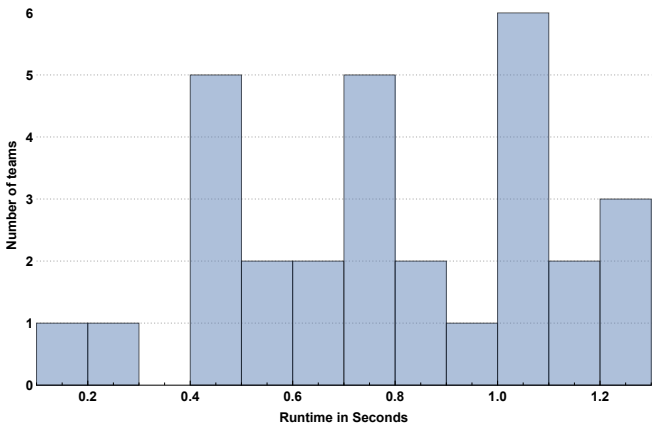

Figure 2. The histogram plots the distribution of the top 30 team runtimes. Each bin in the histogram is 0.1 second interval. For example, 5 teams had a runtime between 0.4 and 0.5 seconds.

during grading.

Container Execution

All student execution commands, specified within the `rai-build.yaml`, are performed within a Docker container. The default RAI container is configured with the latest CUDA toolkit along with CUDNN [16] and other neural network frameworks such as Tensorflow [17] and Torch7 [18]. This was done to allow students to compare their implementation against standard tools.

A new container is started for each job and is terminated after completion. To limit denial of service attacks and to maintain fairness, each student can only submit a job every 30 seconds, and the Docker container is configured without network access, only $8GB$ of memory, and a maximum lifetime of 1 hour. These limits can be changed using the RAI worker configuration file.

VI. Instructor and Student Utilities

Outside of job submission, RAI offers instructors and students a set of utilities that can be used to interact with and query the system. These commands perform tasks such as checking the student ranking within the competition, creating and sending the keys to students, downloading the student's final submissions, etc...

Competition Ranking

The Applied Parallel Programming project was a competition where students formed teams and worked to create the fastest implementation of a fixed neural network inference step. Teams were provided with a baseline serial CPU implementation and a set of neural network weights. The serial implementation took around 30 minutes to complete using the full dataset. Teams were required to use the provided weights and maintain a target accuracy. To encourage competition, teams were able to see their ranking using RAI. The students could also see other teams' anonymized runtimes.

Figure 2 shows how the final submission of the top 30 teams fell into each runtime quantum (each quantum is 0.1 seconds). Most teams fell within the 1 second runtime. The slowest submission took 2 minutes to complete.

Sending Authorization Keys

```
Hello FirstName LastName,

For the Applied Parallel Programming project,
we will not be using WebGPU. The RAI submission
requires authentication tokens to be present
in your $HOME/.rai.profile (Linux/OSX) or
%HOME%/.rai.profile (Windows) file.
The following are your tokens:

    RAI_USER_NAME='myusername'
    RAI_ACCESS_KEY='BsqJuFUI2ZtK4g1aLXf-OjmML6'
    RAI_SECRET_KEY='tUO8PuKhtR9qozBNn33RcH7p5A'
```

Listing 3: The above is an abbreviated version of the authentication email sent to each student in the class. It includes unique authorization keys required to use the RAI client for the project. The complete email also details how to download and configure the RAI client.

To prevent RAI resources from being consumed by people who are not registered for the course, each student is required to have an authorization key to use the RAI system. We developed a tool to automate the generation and delivery of the keys. The tool takes as input the class roster, a comma separated file of the form `{firstname,lastname,userid}`, and creates an email message based on a predefined template (the core of the email is shown in Listing 3). The tool then emails the message to the students.

Downloading and Running Students' Submissions

Section IV describes how all submissions are stored on a centralized file server, and how the final submissions are recorded. We developed a tool that queries the database for the final submissions and downloads the corresponding files from the file server. The tool would then un-archive the downloaded file.

The tool can be run with the option to delete unneeded files, such as intermediate files generated by `make` and the given dataset. The tool can also be instructed to rerun the students' submissions multiple times and display the minimum time. This was done to get a more accurate measurement of the student execution times during project evaluation.

Operating System	Architecture	Stable Version Link	Development Version Link
Linux	i386	URL	URL
Linux	amd64	URL	URL
Linux	armv5	URL	URL
Linux	armv6	URL	URL
Linux	armv7	URL	URL
Linux	arm64	URL	URL
OSX/Darwin	i386	URL	URL
OSX/Darwin	amd64	URL	URL
Windows	i386	URL	URL
Windows	amd64	URL	URL

Figure 3. The RAI client download links are available on the project website. Students can download either the stable or a development versions of the client. The links are continuously updated and reflect the latest builds from both the $master$ and $devel$ branches of the RAI source code.

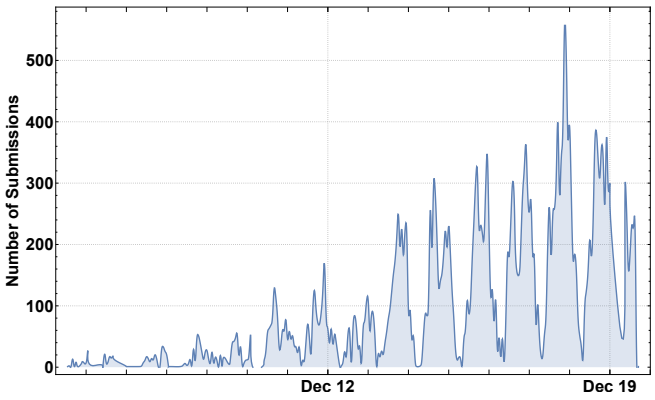

Figure 4. During the last 2 weeks of the course, a total of $30,782$ submissions were made to RAI. This figure shows a timeline of the number of compilation per hour for the last 2 weeks of the course. Students made a significant number of submission during the last week of the course which followed their circadian rhythm.

VII. Lessons and Experience

In the Applied Parallel Programming course, 176 students formed 58 teams and used the RAI system to submit their projects. This section describes the successful experience of using RAI in real coursework, discusses some interesting insights, and lessons learned.

Project Grading

Delivering, administering, and maintaining fairness for a competition with many participants is tricky. Since students have wide latitude when modifying code in open-ended project, RAI does not attempt to specifically support automated grading. The project grading rubric is based on a combination of performance (30%), functionality and correctness (20%), code quality (10%), and an 8-10-page written report (40%). Both the code quality and the report evaluation are performed with human intervention. A project which is graded purely on performance and correctness could be automatically graded using RAI and a simple script.

RAI checked the students' submissions for the required files, source code, and written report. Grading the final submission included ❶ re-running project multiple times and recording the best observed performance, ❷ recomputing the ranking, ❸ and grading the project report. While ❸ needs manual review, ❶ and ❷ were completed by RAI automatically using utility commands.

A grade report for each team was then generated by combining the automated and manual feedback. The grade report was then posted onto the University's grade management system.

RAI Client Delivery

Information on how to use RAI and the location to download the client were given during the course lectures, posted on the forums, and linked within the project description. Figure 3 shows how the RAI client downloads was presented to the students.

We configured the RAI source code repository with two branches: a $master$ branch which contains a stable version of the code base and a $devel$ branch with added features and fixes for non-critical bugs. The $devel$ branch was merged into $master$ as the changes were deemed to be stable.

A continuous build system was configured to build both branches and cross-compile them to other operating systems and architectures. The built binaries are then uploaded to Amazon S3 and linked to the project's home page. Students were able to download either the development or stable version of the client.

The commit version information and build date are embeeded within the RAI binary. Students would provide this information when they reported bugs, which allowed us to narrow which commit introduced the regression. Since we automated the build and delivery process, code changes to fix bugs or address features were automatically made available to students without further action from us. This reduced the administrative load on the teaching staff.

Resource Usage

All students were observed to be using Linux, Mac OSX, and Windows, which were available on their personal machines or campus labs. We kept all the submissions on the file server during the entire duration of the project period. In total, the file server held $100GB$ of data for 176 students.

At the beginning of the project, student activity revolved around experimenting with the provided baseline serial CPU code. This code is slow and requires more CPU compute resources than GPU resources. Therefore, we initially provisioned RAI workers which had less powerful GPUs (AWS $G2$ instances with NVIDIA Tesla $K40$ GPUs). These

instances are cheaper than instances with more powerful GPU resources. Since this baseline code took dozens of minutes to execute, student submissions were infrequent, and we were able to improve performance consistency by restricting a RAI worker to run a single job at a time.

Over the following weeks as students developed optimized GPU implementations, we transitioned to AWS $P2$ instances with NVIDIA Tesla $K80$ GPUs. By the end of the course, 10 instances each allowing multiple pending submissions were needed to provide an interactive response time for students. The last week of the course is when students start performing benchmarks and sensitive profiling. To accommodate this, we provisioned 20 to 30 AWS $P2$ instances which only accept one job submission at a time.

We found that students worked in bursts, which required RAI to be elastic to remain reliable and cost-efficient. Figure 4 shows the number of submissions per hour for the last 2 weeks of the course.

VIII. Conclusion

This paper describes the design and architecture of RAI, a scalable project submission system used for parallel programming courses. RAI addresses challenges of scalability, configurability, security, and cost in delivering a flexible parallel programming environment to students in large-scale programming classes.

By using RAI, students in the Applied Parallel Programming course and the University of Illinois Urbana-Champaign were able to develop and profile their project without having local GPU resources or being aware of the implementation details of RAI. In all, 176 students used RAI to make over $40,000$ project submissions which amounted to over $100GB$ of uploaded data and $25GB$ of logs and meta-data. Students had full flexibility to engage with the development environment in a realistic way.

RAI can cope with submission bursts that occur before project deadlines while allowing administrators to optimize utilization and costs of making esoteric hardware architectures and programming environments available to students. Future work of RAI includes allowing instructors to configure interactive sessions to enable more debugging and profiling tools.

References

[1] A. Dakkak, C. Pearson, and W.-M. Hwu, "Webgpu: A scalable online development platform for gpu programming courses," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 942–949.

[2] A. Computing and G. Computing. Torque resource manager. [Online]. Available: http://www.adaptivecomputing.com

[3] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson, "The portable batch scheduler and the maui scheduler on linux clusters." in *Annual Linux Showcase & Conference*, 2000.

[4] A. Computing. Moab hpc suite. 2016. [Online]. Available: http://www.adaptivecomputing.com/products/

[5] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[6] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[7] O. Hudson. Extendable cntinuous integration server. [Online]. Available: http://hudson-ci.org

[8] C. Jenkins. An extendable open source continuous integration server. [Online]. Available: http://jenkins-ci.org

[9] O. Ben-Kiki, C. Evans, and B. Ingerson, "Yaml ain't markup language (yaml™) version 1.1," *yaml. org, Tech. Rep*, 2005.

[10] Mongodb database. 2017. [Online]. Available: https://www.mongodb.com/

[11] Amazon. Amazon simple storage service (amazon s3). 2017. [Online]. Available: https://aws.amazon.com/s3/

[12] Kitware. Cmake build system. 2017. [Online]. Available: https://cmake.org/

[13] R. Baratov. The hunter cross-platform package manager for c++. 2017. [Online]. Available: https://github.com/ruslo/hunter

[14] M. Folk, A. Cheng, and K. Yates, "Hdf5: A file format and i/o library for high performance computing applications," in *Proceedings of Supercomputing*, vol. 99, 1999, pp. 5–33.

[15] NVIDIA. Nvidia docker. 2017. [Online]. Available: https://github.com/NVIDIA/nvidia-docker

[16] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[18] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.