Emutl: Multi-stage Distributed Computing with BOINC

Vance Zuo
Senior Project, Fall 2015
Advisor: Holly Rushmeier

Abstract

We built Emutl, a system enabling users to define and automatically execute workloads involving multiple applications and arbitrary relationships between the inputs and outputs of these applications on BOINC, an open-source distributed computing middleware system. Currently in BOINC, even something like using outputs from one application as inputs in another application is nontrivial. Emutl aims to fill this gap by providing a simple, flexible PHP API and web interface for users to define and execute pipelines, which are conceptually represented as graphs of dependencies between jobs and resources. We also tested the system with a demo image histogram comparison project involving two applications, including a test run of over 60,000 BOINC jobs.

Background

In recent years, “big data” has spurred the development of distributed computing platforms. One popular framework has been BOINC (Berkeley Open Infrastructure for Network Computing), an open-source middleware system. It was originally created in 2002 to manage volunteer computing projects like SETI@home, which processes radio data for potential evidence of extraterrestrial intelligence. Since then, its use has expanded—as of December 2015, there were 58 active public projects and over 3.4 million contributors. While it is known more for volunteer computing, BOINC can easily be adapted to closed cluster computing applications.

BOINC follows a master-worker model. A project server sends tasks to clients, which execute the tasks and then upload the results back to the server. Because clients are potentially untrustworthy, BOINC emphasizes security and accountability. For example, it can send multiple tasks representing the same job, or workunit in BOINC parlance, to different machines and accept the majority result as correct.

At the Yale Graphics Group, we adopted BOINC in 2014. BOINC has several advantages compared to other solutions that we considered, like Hadoop MapReduce:

1. It is designed to handle heterogeneous computing resources. When a client requests work, the server determines what to send based on that client’s speed and BOINC usage. This is helpful because the Yale Graphics Group’s computers vary widely in power—in particular, some have GPUs and others don’t. By contrast, alternatives usually presume homogenous computing resources, i.e., similar if not identical computing power and availability per node.
2. BOINC’s application framework is flexible – existing applications with multiple files, e.g., executable and DLLs, and arbitrary input and output files can be deployed with little to no modification using the BOINC wrapper. The alternatives require applications to conform to a MapReduce programming model to chop up a task into pieces to distribute, and typically need huge input files sizes (in the gigabytes) to be efficient, so adopting them would require significant changes to our applications and input files.

3. BOINC’s weaknesses are not as significant in our use cases. Its efficiency is poorer with lower compute/data ratios due to the network communication costs, but many image and graphics processing applications involve long computations on relatively small (megabytes) files. It is also not well-suited for real-time applications, but most of our projects involve batch jobs.

Overall, BOINC is a little simpler and more flexible than Hadoop and similar frameworks. However, there is room for improvement. BOINC primarily supports computing sets of independent jobs, and has few tools for managing projects that contain multiple related computations. The first project we used with BOINC was a pipeline that included several stages, including extracting texture information from images and of computing similarities between the texture results. To execute these stages in BOINC would require manually obtaining outputs of one application and submitting them to another application – time-consuming, especially with our multi-terabyte datasets.

Thus, we constructed Emutl, a system for defining and automatically executing these kinds of pipelines. It is intended as an add-on to existing BOINC projects, providing a PHP API and web interface for defining pipelines as a graph of dependencies between resources and jobs, and then automatically execute the pipelines in BOINC. The goal is to abstract away the internals of BOINC, allowing users to focus on program logic.

Emutl is based on our previous work with BOINC, which consisted of scripts for submitting large batch jobs and tracking them in a database. However, this work did not possess automatic pipeline execution capabilities and had very project-specific logic. Emutl, which was largely written from scratch, can automatically execute multi-stage pipelines and has a more general API. It includes:

- A database that tracks pipelines, resources, jobs, and job-resource dependencies
- An administrative web submission interface that takes a pipeline creation script and input resource files, and executes the pipeline accordingly
- A rudimentary web interface for keeping track of job progress

Emutl’s immediate applications are with distributing computations in existing Yale Graphics Group projects, but it should be usable with any BOINC-based project.

Design

This section describes the abstract concepts of Emutl, as well as their concrete implementations.
Technologies

This system makes use of MySQL and PHP scripts. These were chosen so that it would integrate seamlessly with BOINC, which is also based on MySQL and PHP. The MySQL component stores state information about pipelines, including resources and jobs. The PHP scripts component manipulates the database and interacts with BOINC to realize a pipeline.

Pipeline Concept

A pipeline in Emulti is represented as a directed acyclic graph of nodes, where an edge A → B represents a dependency on A by B. There are three types of nodes:

- **Root** – a node that represents the beginning of the pipeline; it should have no dependencies, i.e., zero incoming edges
- **Resource** – a node representing an input or output file associated with one or more jobs
  - It also has state: unavailable, or available
  - If available, it contains resource information needed for creating a BOINC workunit
  - Initially, all resource nodes are unavailable, except for the children of the root node, whose resource information should be obtained during pipeline creation
- **Job** – a node representing a computation, i.e., a BOINC workunit; incoming edges represent inputs and outgoing edges represent outputs of the job
  - It also has state: either initializing, pending, or done
  - If pending or done, it has BOINC workunit information, including output file names
  - All job nodes begin initializing

The graph also follows the following constraints: (1) there is only one root node, (2) resource nodes have outgoing edges only to job nodes, and (3) root and job nodes have outgoing edges only to resource nodes. See Figure 1 for an example.

Figure 1. Illustration of a hypothetical pipeline, consisting of three “stages” corresponding to map and reduce operations, which are common in distributed computing. Notice the alternating layers of resource and job nodes.

---

1 In practice, jobs can also fail, which we take into account in our actual implementation – see the Database Schema and PHP Scripts sections for details.
During the execution of a pipeline, when a job node’s parent resource nodes are all available, it enters the pending state, and a BOINC job representing it and its parent resources is created. When that BOINC job finishes, the job node is set to done, and its children updated to available (and output resource information added). Job nodes are processed asynchronously, so different parts of a pipeline can be executing simultaneously.

This representation provides great flexibility in the definition of complicated sets of interrelated jobs. We, however, anticipate typical pipelines to essentially consist of “stages” of map and reduce operations on lists of resources, and accordingly created an API that allows a user to define a these kinds of graphs without manually creating nodes and setting edges.

**Database Schema**

Emutl uses a relational database to persist a pipeline graph and its state. It schema contains the following tables, which are directly installed onto an existing BOINC database (see Figure 2 for an illustration):

- **Progress** – contains string constants representing pipeline and job state: init, pending, done, and failed
- **Pipeline** – contains entries representing independent pipelines; also tracks each pipeline’s progress and approximate start/end time
- **Resource** – contains metadata for job resources in a pipeline (for local files, the name of the file; for remote files, the URL, MD5 hash, and size of file)
- **Job** – contains information about a job in a pipeline, including its BOINC app name and workunit name (to allow us to submit it to BOINC and later retrieve the results), and also its progress and start/finish times
- **Input and Output** – contains many-to-many relationships between BOINC jobs and their input and output resources, respectively; also stores their BOINC file number as defined in the BOINC job’s input and output templates (allowing us to properly identify each input or output); inputs may also have a token

Pipeline, resources, and job entries also have a create time attribute to mark when they are added to the database. This field is automatically set to the time they are created via a SQL trigger.

These tables directly corresponds with aspects of the pipeline graph representation described previously. Pipeline rows specify independent graphs, job and resource rows describe nodes, and input and output rows represent edges. Job node state is represented by the job progress field, while resource node availability is represented by whether the resource URL field is non-null.

A full SQL schema for the Emutl database can be found in install.sql. Note also uninstall.sql, which drops the tables created by install.sql.

---

2 A token is designed to limit non-Emutl access to remote server files. Emutl includes the token in its GET request for remote files; the remote server then queries the Emutl server to verify the token’s validity before allowing the download (see validate.php in the Web Layer section). Note the Emutl token implementation is currently unfinished.
Figure 2. Illustration of the Emul database schema. Note the app and workunit tables originate from the BOINC database, but are referenced by the job table.

PHP Scripts
The PHP component of the system contains most of the pipeline execution logic, including the necessary queries for manipulating the Emul database. It consists of scripts located in the BOINC ops directory, which is protected by htaccess restrictions to prevent unauthorized access to sensitive administrative operations. They are organized into modules covering different areas: database, utility, graph, and web.

Database Layer
The database layer contains static functions for executing SQL queries to read and modify pipeline representations in the database, using PHP’s PDO driver. It is further organized into several classes (in a similar way to how BOINC’s database code is organized) for general database and table-specific operations:

- **EmutlDb** – contains functions for:
  - Establishing a database connection
  - Executing arbitrary queries
  - Installing/uninstalling the database tables
- **EmulPipeline** – contains functions for:
  - Creating a pipeline entry
  - Updating pipeline state either to pending or done
Retrieving pipelines that meet conditions to transition to pending or done states

- **EmutlResource** – contains functions for:
  - Creating a remote or local resource entry with initial URL, MD5, and size attributes
  - Updating a resource's attributes

- **EmutlJob** – contains functions for:
  - Creating the job, input, output, and resource entries for an Emutl job with a given application name, input resource IDs, and number of output resources.
  - Updating job state to either pending or done
  - Retrieving jobs that meet conditions to become pending or done

This layer's source code is located in `emutl_db.php`, with a test script in `emutl_db_test.php`.

**Utility Layer**

The utility layer provides functions for creating and executing pipelines. It provides a layer of abstraction over the database layer and BOINC – handling both the manipulation of the database to keep it in an up-to-date, consistent state and also BOINC shell function calls to prepare and execute jobs. Its public API includes functions for:

- Installing and uninstalling Emutl in the current BOINC project
- Validating tokens for input resource URLs
- Creating and returning IDs for a named pipeline with a list of initial resources (these correspond to the children of the root node in a pipeline graph)
- Creating and returning an ID for a job for a certain pipeline ID with a certain application name, input resource IDs, and number of outputs
- Updating all pipelines; this involves several distinct parts:
  - Staging files added during pipeline creation or job completion – that is, adding resources to BOINC's download path (where it is accessible by clients)
  - Processing BOINC's and workunit error file and result files to find failed or finished workunits, and updating their job and resource entries accordingly
  - Updating pipeline state from init to pending, and pending to done as appropriate
  - Updating job state from init to pending, and pending to failed or done as appropriate

Note the update function currently makes a few assumptions about the BOINC project configuration to work properly. It requires the BOINC project to be using the default sample assimilator for all of its applications to correctly locate workunit errors and results. It also

---

3 For pending, all init pipelines; for done, all pending pipelines whose associated jobs are done. Currently failed jobs are also automatically restarted, i.e., changed to pending.
4 For pending, all init jobs whose associated input resources are available; for done, all pending jobs whose associated output resources are available. Recall that available in the database context means the URL field is non-null.
5 Assuming BOINC is using the default sample assimilator, results are stored in a project root directory called sample_results. An error file containing one line for each failed workunit, including the workunit's name, is also written there.
needs resources used for multiple jobs to specify <no_delete/> in their BOINC input file template; otherwise, the file is automatically deleted after the first job.

The source code for this layer is contained in `emutl_util.php`, with a demonstration of some of its functionality in `emutl_util_test.php`.

### Graph Layer

The graph layer provides a further layer of abstraction on top of the utility layer, providing an object-oriented, map-reduce-inspired API for generating pipelines and then actualizing them in the system. It contains the following classes:

- **EmulPipelineBuilder** – represents a pipeline graph; contains the following methods:
  - `build()` – “actualizes” the pipeline by converting it to database form via a breadth-first traversal of the graph
  - `input_one()` – adds a pipeline input resource, i.e., a root child resource node
  - `input()` – adds multiple input resources to the graph
  - `map_one()` – adds a job applying an application on a list of input resource nodes, creating job and output resource nodes accordingly
  - `map()` – adds jobs applying an application on a set of input resource node lists
  - `map_combination()` – a map() on fixed-size mathematical combinations of a set of resource node lists
  - `map_product()` – a map() on the Cartesian product of sets of resource node lists
  - `reduce()` – adds jobs that aggregate a set of input resource node lists by recursively applying an application on pairs of the set and the resulting output resource node lists

- **EmulNode** – represents an node in a pipeline graph; has methods for:
  - Adding input and output edges
  - Setting a processed attribute (to aid `EmulPipelineBuilder build()`’s traversal)
  - Indicating if the node is “ready,” i.e., whether all of its parents have been processed

- **EmulRootNode** – subclass of `EmulNode` for root nodes

- **EmulResourceNode** – subclass of `EmulNode` for resource nodes; has an additional `resource` attribute for storing resource information or IDs

- **EmulJobNode** – subclass of `EmulNode` for job nodes; has an additional `app_name` attribute for storing a BOINC application name

---

6 This is similar to a MapReduce map operation.

7 For example, a map of combinations of size two on `[[1, 2], [3, 4], [5, 6]]` would be the same as a map on `[[1, 2, 3, 4], [1, 2, 5, 6], [3, 4, 5, 6]]`. Note the generated combinations are flattened.

8 For example, a map of the (Cartesian) product of `[[1], [2]]` and `[[3], [4]]` would be equivalent to a map on `[[1, 3], [1, 4], [2, 3], [2, 4]]`.

9 This is similar to a MapReduce reduce operation. Starting with the input resource lists, as long as there is more than one list remaining, it produces a round of jobs for pairs of lists. For example, a reduce on `[[1, 2], [3, 4], [5, 6], [7, 8]]` would create jobs for input lists `[1, 2, 3, 4]` and `[5, 6, 7, 8]`, producing output lists (say) `[9, 10]` and `[11, 12]`; it would then create a job for `[9, 10, 11, 12]`.

---
The intended typical pattern for creating a pipeline with this API is to create an EmutlPipelineBuilder object, and then add initial resources using input(), and then add computations with map(), reduce(), etc., and then finally call build() to actually create and execute the pipeline. This doesn’t do anything we couldn’t do with the utility layer, but it is arguably easier to use and understand (see Figure 3).

This layer’s source code is in emutl_graph.php, with a test in emutl_graph_test.php.

```php
$name = "Example Pipeline";
(resources = [ ["resource/apple1.jpg", null, null, false], // URL, MD5, nbytes, remote
 ["resource/apple2.jpg", null, null, false]]);

// Using utility layer
function create_pipeline_using_utility() {
    list($pipeline_id, $image_ids) = EmutlUtil::create_pipeline($name, $resources);
    list($job_id1, $hist_ids1) = EmutlUtil::create_job($pipeline_id, "hist",
        $image_ids[0], 2);
    list($job_id2, $hist_ids2) = EmutlUtil::create_job($pipeline_id, "hist",
        $image_ids[1], 2);
    EmutlUtil::create_job($pipeline_id, "cmp", [$hist_ids1[0], $hist_ids2[0]], 1);
}

// Using graph layer
function create_pipeline_using_graph() {
    $graph = new EmutlPipelineBuilder($name);
    $images = $graph->input($resources);
    $hists = $graph->map("hist", $images, 2);
    $graph->map("cmp", [$hists[0][0], $hists[1][0]], 1);
    $graph->build();
}
```

Figure 3. A comparison of two ways of creating a simple pipeline with either the utility layer functions or graph layer functions – see create_pipeline_using_utility() and create_pipeline_using_graph(), respectively. The sample pipeline consists of applying an application ‘hist’ to two images, and then applying ‘cmp’ to the two result’s first outputs. Notice that the graph API is simpler and abstracts away details like the IDs of specific pipelines, resources, and jobs.

Web Layer
The web layer provides a simple administrative interface for managing Emutl. It contains several basic PHP pages for executing Emutl operations:

- **emutl_install.php** – a form for installing Emutl
- **emutl_uninstall.php** – a form for uninstalling Emutl
- **emutl_validate.php** – a form for checking if a token for a resource URL is valid
- **emutl_create.php** – a form for submitting an Emutl pipeline with for execution; based on three user-provided parameters:
  - The name of the pipeline
  - A file containing resource entries with URL, MD5, size, and remote flag
- A file containing a PHP script that takes two parameters, an EmutlPipelineBuilder object and a list of resource entries, and returns a function when executed will create an pipeline graph (for a simple example see Figure 4)
  - `emutl_update.php` – a script that manually updates all Emutl pipelines
  - `emutl_check.php` – a script that displays some information about pipeline progress, including how many jobs are init, pending, done or failed; and how many resources are available or unavailable.
  - `emutl_index.php` – a page containing links to the aforementioned pages

Users can, of course, extend this layer with their own scripts.

```php
var $create = function ($graph, $resources) {
    $graph->map("example_app", $graph->input($resources), 1);
    $graph->build();
}
return $create;
```

Figure 4. A simple example of the kind of PHP script that would be submitted in an `emutl_create.php` form. It simply returns a function that applies a single map of the application "example_app" on the pipeline input resources.

**Other**
Besides the database, utility, graph, and web scripts, Emutl also has a logging module in `emutl_logger.php`. It is tested and usable, but hasn’t yet been integrated with the rest of the project. Emutl also has a module containing exceptions thrown by its scripts in `emutl_exception.php`.

**Usage**
Emutl provides infrastructure for defining and executing pipelines. This section details how users can use it with their own applications.

**Installation**
Users need to have a BOINC project set up with all the applications they intend to use in Emutl. As discussed in the utility layer section, these applications should be using the sample assimilator and their input templates should have `<no_delete/>` for files used in multiple jobs. Additionally, users need to set certain file and directory permissions for BOINC and Emutl to work (these operations were compiled into the `set_permissions.sh` script).

Once BOINC is set up, users can extract an Emutl installation\(^\text{10}\) into the project directory. The installation contains all the PHP scripts and other miscellaneous files, organized in directories so that when merged with BOINC they will be in the correct locations. Users should then create the Emutl database via the `emutl_install.php` web script. Finally, they should add a periodic task in BOINC’s config.xml for `emutl_update.php` to enable automatic pipeline updating (see Figure 5).

\(^{10}\) This is the contents of the src directory in the emutl.zip archive that accompanies this report.
Figure 5. Entry to add to BOINC’s config.xml (inside the <tasks> section) to enable periodic Emutl pipeline updating.

**Pipeline Creation**

To create a pipeline, users can use the `emutl_create.php` web form, or write and execute their own pipeline creation script in the server. In either case, users need to write a small amount of pipeline creation code using the graph layer API and possibly the utility layer API (recall Figure 3 and 4), and provide a set of input resources for the pipeline. These input resources can be local or remote (though outputs are all currently stored locally in a project root resource directory).

**Pipeline Inspection**

Currently there aren’t many tools for inspecting pipelines in detail, but users can broadly track pipeline progress via the `emutl_check.php` web script, and check individual BOINC jobs via BOINC’s administrative ops interface.\(^x\)

**Tests**

In this section we present practical demonstrations of Emutl.

**Image Comparison Pipeline**

Emutl was tested on a demo pipeline consisting of two simple applications:

- **hist** – a image HSV histogram calculator
  - Input: an image file
  - Outputs: a text file containing the histograms (counts for each HSV value), and an image file illustrating the histograms

- **cmp** – image histogram comparator
  - Input: two histogram text files, i.e., outputs of hist
  - Output: text file with one number indicating histogram similarity

Both programs were written in C++ with the OpenCV (Open Computer Vision) library. Their source file names are `hist.cpp` and `cmp.cpp`.

The pipeline itself consists of two stages:

1. A hist map on input resources
2. A cmp map on combinations of the text file outputs from the previous stage

That is, this pipeline calculates a color histogram for all images, and then compares these histograms with each other to compute image similarity. It isn’t a particularly sophisticated algorithm, but is sufficient for demonstrating Emutl.
We tested Emutl on the image comparison pipeline on a BOINC cluster of three commodity PCs. BOINC was active full-time on one PC, but sharing the other two with other users.

Initially we processed a small dataset of 3 fruit images that were on average 5KB (see Figure 6). We then tested running a pipeline on a more realistic dataset, taken from various Yale Graphics Group sources, of 384 images averaging 2.4 MB each – first a short run with 10 images, and then a run with the full dataset. These resulted in the creation of 55 and 67896 total BOINC jobs, respectively.

Emutl was able to execute these pipelines practically to completion, albeit with some delays stemming from novel Emutl bugs and BOINC server/client issues. Because of these problems, Emutl's recorded performance for these pipelines and their jobs vary widely – not to mention they have poor resolution, as the update script appears to run only once every five minutes even when setting it at a faster interval – so they aren't very useful or informative.

![Image Comparison Pipeline Diagram](image.png)

Figure 6. Illustration of the image comparison pipeline on a three image data set of fruits. The hist app is applied to each, yielding HSV image histograms in image and text. The cmp app is then applied to each unique pair of text outputs, producing a similarity score. Evidently the similarity measure isn’t the best (which wasn’t the goal), but this pipeline demonstrates how an actual application may be incorporated into Emutl.

---

11 Technically, the full pipeline didn’t complete because one of the input files was (unintentionally) corrupted. But discounting that, it successfully executed all jobs in both pipelines.

12 For example, we neglected to add `<no_delete/>` to cmp’s input template. It took several days to realize this was the problem. We also faced issues with running out of storage space on our VM server, which led to periodic delays to free up space.

13 Nonetheless, for completeness these were the recorded timings: the short pipeline ran for 6 days; the full pipeline has no recorded length due to a corrupt input resource preventing it from finishing. The mean recorded hist job time was 40 minutes for the short pipeline, and 5 minutes for the full pipeline. For cmp it was 6 minutes for the short pipeline, and 26 hours for the full pipeline. The `timings.txt` file details the queries used to calculate these averages.
Improvements

There are many areas that could be improved in future work.

Testing
Due to time constraints, testing on Emutl is incomplete. We only managed to set up and test one BOINC project, and weren't able to test it on two Graphics Group projects as intended – coding the system itself took longer than expected, and integrating the projects with BOINC proved to be more complicated than anticipated. A high priority in future work would be to run the system on more projects.

We also have performance results for just one large data set, not to mention it had a small cluster and many runtime issues. It is important we run more trials on larger clusters. We should also address the issue where the update script runs only every five minutes – perhaps redesigning the update script to run continuously as a daemon.

Further, some features developed more recently, such as some of the web scripts and the graph layer reduce operation, haven’t been fully tested. In the future we intend to test these features and ensure they work correctly.

Features
Some of Emutl’s features are still rough around the edges or incomplete:

- Its handling of failed jobs is rudimentary – it simply retries the job. It may be better to cancel it and successor jobs, or at least to give users the options to do so.
- The token feature is only partially implemented – it’s in the database schema, and there are functions for validating tokens, but Emutl doesn’t currently create or manage tokens.
- All job outputs currently can only be stored locally on the BOINC server. An option to upload them to a remote location should be available.
- The graph layer has map and reduce-like operations, but nothing resembling a filter operation – this should be added.
- The pipeline inspection interface could be expanded. Besides the tools provided by BOINC, currently the only feature in this area is the rudimentary emutl_check.php script. It would be useful in the future to create more detailed tools that allow users to view and filter specific job and resource entries, and to walk through a pipeline graph. A graph visualization tool would also be helpful.
- The Emutl web scripts are not currently well-integrated with the rest of the BOINC web interface. It would make sense to add links in the BOINC ops page to the Emutl web scripts, and vice versa.

These would all be fruitful areas for additional work.

Code Quality
We tried to make the code as organized and self-documenting as possible; however, it certainly has its fair share of redundant and messy code. And there is little formal documentation. For long-term maintainability, it will be important to thoroughly document the codebase.
The organization of the modules could also be improved. In particular, the BOINC-related code of the utility layer probably should be split into its own module – much like the database operations are organized into their own layer.

Conclusion

Creating Emutl has been a great learning experience in a variety of areas: distributed computing, databases, and API design. In particular, we found integrating the system with an external framework (BOINC) and making it as user-friendly as possible to be particularly challenging.

Although there is much room for improvement, we have created a basic system that can run pipelines containing thousands of jobs, with a pipeline creation and execution API that is much simpler than manually scripting in vanilla BOINC. We are hopeful that Emutl can be effectively applied to complex BOINC projects in the Yale Graphics Group and beyond.

---