|  |
| --- |
| **Design and Implementation of a Scalable, Adaptable Multi-stage Pipeline for Medical Applications** |

**Final Project Report**

**Final Requirements, Design, Implementation/Testing & Installation/Delivery**

**Dr. Frank Mueller & Mr. Amir Bahmani**

**CSC 492 Team 29:**

**Monis Khan**

**Hussein Koprly**

**Sarah Baucom**

**North Carolina State University**

**Department of Computer Science**

**May 5, 2016**

## **Executive Summary for Written Interim Project Report**

This project focused on creating a generic framework for representing a medical pipeline and executing it across a distributed cluster for a large number of patients. Two example pipelines were considered: knee cartilage MRI analysis and DNA sequencing.

We have designed a robust JSON specification for representing a pipeline. The data dependency information contained within this spec is used to represent the pipeline as a directed acyclic graph (DAG). The structure of this graph is used to prioritize different stages of the pipeline. To effectively distribute work across a cluster, we created a master & worker paradigm using Python and MPI. This system supports dynamic work scheduling using the aforementioned DAG. We implemented a simple filesystem based checkpointing system to limit the amount of wasted CPU time due to a system failure. We have completed unit testing with over 99% code coverage. We also have a thorough black box testing suite. We have completed the DNA pipeline’s JSON representation, and have successfully run multiples tests using medium size datasets. We have created the knee MRI pipeline’s JSON but were unable to test it due to various issues.

**Project Description**

**Sponsor Background**

Dr. Frank Mueller is a faculty member of NCSU’s Department of Computer Science. He leads the Systems Research Lab and conducts research that is related to Parallel / Distributed Systems. Mr. Amir Bahmani is one of his doctoral students. The motivation of this research is to speed up computations related to medical research, and thus to eventually have a positive impact on the lives of patients.

This project is the first step in a much grander design. Specifically, the researchers plan to develop generic, expandable technologies for creating pipelines suitable for a variety of medical big data applications running on a diverse set of hardware and software.

Dr. Frank Mueller

Professor of Department of Computer Science

North Carolina State University

3266 EB2

Raleigh, NC 27695

(919) 515-7889

[mueller@cs.ncsu.edu](mailto:mueller@cs.ncsu.edu)

Mr. Amir Bahmani

Computer Science Ph.D Candidate

North Carolina State University

3226 EB2

Raleigh, NC 27695

(919) 515-7889

[abahman@ncsu.edu](mailto:abahman@ncsu.edu)

**Problem Statement**

The quantitative analysis of MRI imaging and DNA sequence data has become important in both research and clinical studies. Domain experts have developed sequential "pipelines" for knee cartilage MRI analysis (KSRT) and DNA sequencing. A pipeline consists of a series of executables / stages that may have data dependencies (with other stages in the pipeline). These can be executed in a parallel and distributed fashion (assuming the necessary prerequisites have been completed) to speed up overall computation time.

Scientific researchers face a lot of problems when building these pipelines. For example, every pipeline is constructed from scratch and thus there is limited code reuse. Researchers commonly build many scripts (written in a variety of programming languages) that simply run each stage in the pipeline sequentially. Since the code to run the pipeline and the pipeline’s stages are deeply coupled, there is no way to reuse these scripts when it is time to build another pipeline. Thus, a lot of time is wasted on rebuilding the same infrastructure. Furthermore, these solutions do not scale, because they essentially run one stage at a time. This is no problem when you have a small number of patients, but grows without bound once a large number of patients need to be processed. Even in situations when data dependencies require that each stage be run serially, parallelism can be gained by running multiple patients at the same time. Traditional pipelines operate on one patient at a time, and thus have no way to achieve such parallelism. Due to a lack of recovery and verification mechanisms, these pipelines often have to be rerun because they produce incorrect results or the machine running the computation fails.

We built a system that decouples the processes of running a pipeline from its structure. This will allow the researches to focus on the structure of their pipelines, and not how to execute them. Furthermore, this upfront declaration of structure will help researchers realize and expose the parallel aspects of their pipelines.

**Project Goals & Benefits**

A core goal of this project was to create a generic and declarative framework for representing a pipeline. This is fundamental to the application of this project in a variety of scenarios. By constraining the definition of a pipeline to a declarative model, we will improve code reuse. That is, by limiting the capabilities present to express a pipeline, all pipelines will have a more constrained structure that is easier to understand, reuse and optimize.

There are a variety of technologies that allow scientists and engineers to create complex applications that require high compute capabilities. For example, Message Passing Interface (MPI) is a library for distributed memory parallel programming. Python is an easy to use high-level language. It is popular in the scientific community because of its emphasis on code readability and speed of development time. We used MPI and Python in conjunction to address the project’s goals.

MPI is used to scale and coordinate these operations across a cluster, and allows for patient level parallelism in all situations, and stage level parallelism when there are no outstanding prerequisites. Another important factor for scaling is to keep the entire cluster busy with processing each patient’s pipeline. Thus there must be some type of dynamic scheduling to ensure effective use of all MPI processes. Furthermore, based on the structure of the pipeline, different heuristics can be used to prioritize stages to allow for more parallelism. All of these can contribute to the overall amount of time saved. Domain experts will be able to complete their research quicker, and thus patients will receive the benefits of their efforts sooner.

To reduce the amount of time wasted due to system failure, we included the ability for the system to recover most of the work that has been completed so far. Each stage of the pipeline provides a configurable mechanism to validate the results, and thus avoid situations where data was incorrectly processed without the researcher’s awareness.

To demonstrate the effectiveness of this system, both the KSRT and DNA pipelines have been implemented using the declarative framework and run on ARC (a supercomputing cluster located in the Systems Research Lab). No modifications will be made to the stage executables as they will be treated as black boxes (only domain experts have the knowledge to modify these programs).

**Development Methodology**

This project is a hybrid between a research and software development project. Thus we did not employ any specific development methodology. We did not have a fixed number of iterations or iteration length. Our sponsor originally wanted us to directly implement the DNA and KSRT pipelines, but we convinced them it would be more benefit to the field if we build a generic system that could be used to run arbitrary pipelines.

**Challenges**

Defining a data format that represents the declarative pipeline framework required some work. It needed to be flexible enough to represent a variety of pipelines, but simple enough to reason about without too much coding knowledge. It represents the logical steps and structure of the pipeline.

* We created a robust JSON specification to represent the stages in a pipeline

Time is a major concern. For example, the preprocessing stage of the DNA pipeline takes twelve hours when running on ten nodes in ARC. Thus running tests takes a long time. We cannot simply saturate the ARC cluster as this would prevent others from using it.

* We ran a limited set of tests for the DNA pipeline using medium sized datasets
* The timing results for these were relatively consistent, but more tests should be run before making any strong conclusions

The size of ARC is also a concern. It is limited to a maximum of 1,728 processes (108 nodes with 16 processors each). We need to run the system at a large enough scale to determine if it has any bottlenecks or failure points (and to stress test it in general). ARC may not provide enough compute resources for us to do this thoroughly.

* We were never able to stress test the system as ARC does not have enough resources to run a large dataset

**Resources Needed**

* ARC Cluster
  + Supercomputing cluster housed in the Systems Research Lab at NCSU
  + Provides 108 CPU nodes with 16 processors each
* Linux (tested on CentOS 5.7 with kernel 2.6.32.27)
  + ARC is a Linux only environment
* MPICH2 (tested with version 1.4.1p1)
  + Required for mpi4py
  + Provides MPI standard
* Anaconda Python 2.7 (tested with CPython 2.7.11)
  + Required to run Hivemind
* mpi4py (tested with version 2.0.0)
  + Provides Python bindings for MPI
* NetworkX (tested with version 1.11)
  + Graph package used by Hivemind Pipelines

**Requirements**

**Overall View**

One of the main challenges researchers face with distributed systems is coming up with an effective way to parallelize their work, in order to reduce the time required to process their data. We are developing a generalized framework that will make it easier for researchers to accomplish efficient parallelization.

Domain experts at UNC, Duke, and NC State are currently running sequential pipelines for DNA sequencing and KSRT. Their existing implementations do not make any use of parallelization, and are purpose-built for the pipeline they want to run, meaning effort often has to be duplicated for other pipelines (for clarity, an example is included in the User Guide to illustrate how a sequential pipeline can be converted into a parallel one). Our framework will be able to run independent stages in parallel to more effectively make use of a distributed system, and dynamically schedule work using prioritization to keep all nodes busy. It will use a flexible specification design to handle many different pipelines, including KSRT and DNA. The framework will also use checkpointing to recover from system failure and post-stage verification to verify correctness.

**Detailed Requirements (names in brackets)**

Functional

1. [Parallel] Independent stages run in parallel. Stages that do not depend on the output of one another are considered independent stages and can be run in parallel if workers are available. For example, if there are 100 independent stages, but only 40 workers, 40 stages will run in parallel
2. [Flexible] Flexible specification design. The framework must be able to handle a variety of pipelines. A specification will be created so that domain experts can define their pipeline and patients.
   1. [DNA] Demonstrate DNA Sequencing pipeline. The existing DNA pipeline should be demonstrated as working on the new system, with identical output results.
   2. [KSRT] Demonstrate Knee MRI pipeline. The existing KSRT pipeline should be demonstrated as working on the new system, with identical output results.
3. [Scheduling] Dynamic scheduling of work. Rather than determine up-front which work will go to which workers, the system will dynamically assign work based on what stages are able to be run, and which workers are available.
4. [Recovery] Checkpointing for recovery. To recover from a system failure, the system shall utilize checkpointing as stages are completed. The pipeline and patients should be able to resume where the system failed. The system shall provide an easy mechanism for restarting execution.
5. [Prioritization] Prioritize stages via data dependencies. Pipeline stages may have other stages that depend on their output data to begin execution (successors). These stages can only begin once all of its predecessors complete. Each stage may have any number of successors, from zero to many. If a stage has many successors, it should be worked on before a stage with few or no successors, to open up more stages that can be worked on. This will lead to higher parallelization.
6. [Reporting] Output from each stage will be directed to a log file.

Non-functional

1. [Demo] for both KSRT and DNA pipelines should have improved performance (in terms of running time) over their sequential implementations.

Constraints

1. The system shall be developed using Python
2. The system shall use MPI via mpi4py to communicate across a distributed cluster
3. The system shall not require the stage executables to be modified

**External Dependencies**

1. The system will run on the NCSU ARC cluster
   1. Requires a shared file system across the entire cluster so that all nodes can see the output files from different stages
2. The KSRT pipeline binaries
3. The DNA sequencing pipeline binaries

**Design**

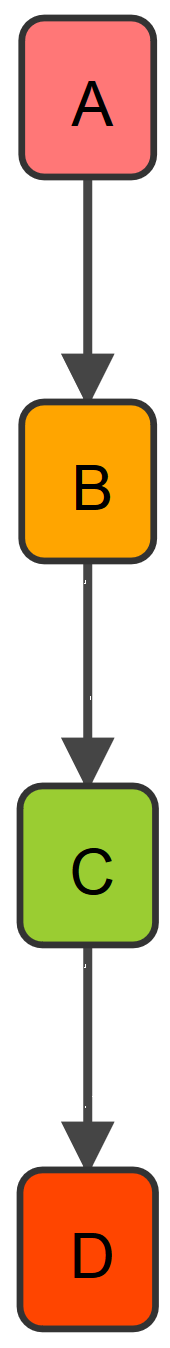
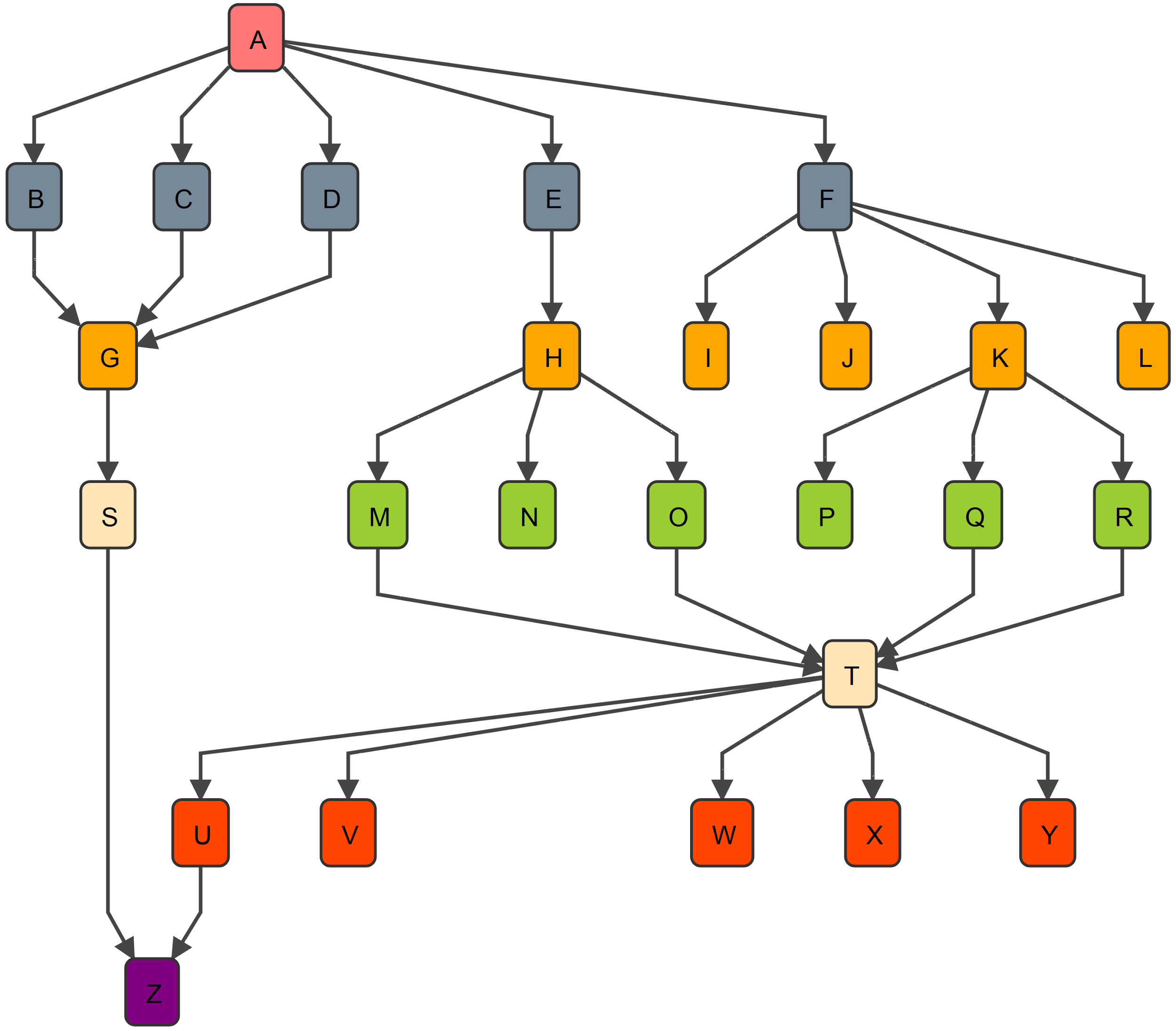
MPI significantly eases the task of writing distributed data-parallel code. The basic concept of MPI is enabling processes to communicate by sending messages to one another. MPI provides an opportunity for increasing the performance of parallel applications. We present the core design of a framework that makes it easy to develop and run pipelines in distributed fashion using MPI.

The core concepts of the framework are:

* Directed Acyclic Graph (DAG)
* Master-worker paradigm

**Directed Acyclic Graph**

A pipeline encapsulate a series of executables (also known as tasks) that may have data dependencies. The simplest pipeline represents a linear flow of tasks, as shown in *Figure 1* below:

*Figure 1 Figure 2*

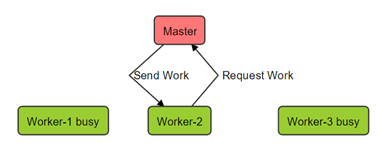
However, a pipeline can be more complex, as shown in *Figure 2*. The framework uses a DAG to represent the structure of a pipeline and prioritize different stages within a pipeline. Each node in the graph represents an executable and all of its arguments, and it can have data dependencies on a node or nodes that must complete before it can start executing. From Figure 2, we can see stage B, C, and D must complete before stage G can start. Furthermore, we can see that stages E and F are independent from each other, and from the previously mentioned stages, thus they can be run in parallel. We will use the pipeline structure to create a specific pipeline for each patient that can be executed on a distributed cluster.

**Master and worker paradigm**

The second core concept of the design is the master-worker paradigm. It consists of two entities: a master and multiple workers. The master is responsible for:

* Building the pipeline structure from given tasks and their requirements
* Ranking the tasks within the pipeline using different ranking methods (FIFO, total successors, and immediate successors)
* Creating a specific pipeline for each patient based on given patient data
* Maintaining a priority queue of ready tasks
* Maintaining a queue of available workers
* Coordinating ready tasks with available workers by sending ready tasks to idle workers
* Determining the maximum number of tasks that can still be executed concurrently

The workers are responsible for running tasks that are sent to them by the master. The master and workers communicate by sending and receiving messages to one another using MPI. The system will be based on a pull model. Thus, the master will only send work when a worker asks for it. As we can see in *Figure 3*, “worker-2” is asking the master for work, while both “worker-1” and “worker-3” are busy working. A worker can either be working or waiting for work until told to exit by the master.



*Figure 3*

**Design Assumptions**

* Stage Idempotence (allows for checkpointing). This essentially references a property that is common is REST APIs. Essentially the pipelines must be built in a way that it expresses the following properties (failing to due so will cause incorrect behavior during recovery from system failure):
  + There should be no negative side effects if a stage is executed more than once
  + Partial intermediate files should not cause any errors
  + The input and output files should be different per stage
  + Each stage should not modify the input file in any way
* The average running time for each stage should be greater than ten seconds otherwise the communication overhead of MPI and the funneling effect of having a single master node will cause most of system to be idle as the master distributes work (and it does not really make sense for a user to need a distributed cluster to run such simple stages)
* No cycles within a pipeline (they must be pre-flattened by the domain expert and we will raise an error if a pipeline contains a cycle)
* The domain expert must guarantee that all nodes can access any file that is used during the running of the system

**Implementation**

To implement the core concepts of the framework, we present the Hivemind package to create a pipeline and run it efficiently across a distributed computing system. The core components of the Hivemind package are (see the developer guide for the directory structure of Hivemind):

* Data input and reporting
* Pipeline
* Master-worker paradigm

**Data Input and reporting**

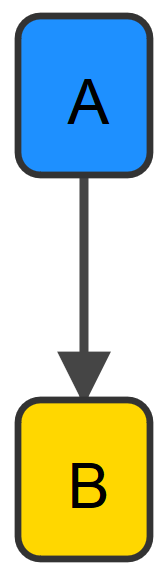
In Hivemind we use a flexible and high level intuitive primitive for describing the structure of a pipeline. This is very simple example of two stages pipeline. For a more detailed example see the User Guide.

To start building a pipeline, the user needs to provide the structure of a pipeline in a JSON file using the specification shown below, and a CSV file containing the patient’s data. Fields like “\_uid” (unique task id), “exe” (task executable), and “wd” (working directory) are required to be specified, while “args” (arguments passed), “requires” (dependency of a task), and “verify\_exe” (post-stage verification) are optional. See *Figure 4* below for an example.

[

{

"\_uid": "A",

"args": ["arg-1", "/$$arg$$"],

"exe": "bin/task-1",

"wd": "/workingDirectory"

},

{

"\_uid": "B",

"args": ["arg-1", "arg-2", "/$$arg$$"],

"exe": "/bin/task-2",

"wd": "/workingDirectory",

"\_requires": ["A"],

"verify\_exe": "/bin/verify"

}

]

*Figure 4*

Fields with $$ at each end are variable names that will be replaced with variables specified for that patient in the CSV file. The following example is a simple CSV file containing two patients’ data.

patientID,$$arg$$

136,flip

246,noFlip

Hivemind combines both the JSON file and CSV file to create a concrete pipeline for each patient specified in the CSV file. In the example above, the $$arg$$ variable will be replaced with "flip" for task A and task B in first patient’s DAG, and with "noflip" in the second patient’s DAG.

**Pipeline**

After parsing both JSON and CSV files from the user, Hivemind starts with validating the pipeline structure that is specified in the JSON file. As mentioned earlier, a DAG is used to represent the structure of a pipeline, and thus a pipeline with cycles is considered invalid. Hivemind uses NetworkX which is a Python library for creating and manipulating the structure of graphs and networks. The pipeline module is responsible for creating a specific pipeline for each patient, whereas the task module is used to create each Task within a pipeline. Hivemind uses different ranking algorithms for ranking and prioritizing tasks within a pipeline. The two algorithms implemented within Hivemind are total successors and immediate successors. This will help to schedule and run independent tasks within a pipeline in parallel as long as there are workers available.

The maximum concurrency of a pipeline is defined as the maximum independent set of the transitive closure of the pipeline's DAG. To calculate this we must reference Dilworth's theorem and Kőnig's theorem. Kőnig's theorem proves an equivalence between a maximum matching and a minimum vertex cover in bipartite graphs. A minimum vertex cover is the complement of a maximum independent set for any graph. Thus we create a bipartite graph based on the transitive closure of the DAG. Then we apply the Hopcroft–Karp algorithm to calculate the maximum matching of the bipartite graph, which is the complement of the maximum independent set. Thus we can use this to determine the maximum concurrency of the pipeline, which is the number of nodes in the maximum independent set.

**Master-worker paradigm**

Hivemind uses the two foundational concepts of MPI, send() and receive(), for communication between master and workers. To start running the actual pipeline, the user needs to run main.py and pass both the JSON and the CSV files as an arguments (see User Guide for further details). The runtime process is as follows:

* Creating a concrete pipeline for each patient using the data specified in the CSV file
* Maintain the workers queue for available workers, and the tasks queue for ready tasks
* Checking the number of maximum tasks that can be executed concurrently, and the number of workers available. If the number of workers exceed the number of maximum concurrent tasks, the master will close all unnecessary workers in the workers queue.
* Master will start sending tasks from the tasks queue to the workers in the workers queue.
* Workers start executing the given tasks from the master, and the master will wait for the completion message from worker
* When master receives a completion message from any worker, the master will create an empty file called "\_.done" in the task’s checkpoint folder indicating the completion of that task. This is a task-level checkpointing technique that uses the file system to save the current state of a pipeline. Hivemind uses checkpointing mechanism to recover from the system failure (the Master checks for the existence of these files at startup and skips the tasks that have already been completed)
* The master will also set the task in the pipeline to done, update the number of maximum concurrent tasks, and enqueuing the worker back to the workers queue
* The master will add new ready tasks from concrete pipelines to the tasks queue.

**Test Plan & Results**

For testing, we used a combination of unit tests, blackbox tests, and performance tests with the DNA pipeline on the ARC cluster.

For human readability, the text contents of the output files are shown. For the actual tests, the sha256sum hashes are compared.

|  |  |  |  |
| --- | --- | --- | --- |
| **Blackbox Test Plan** | | | |
| Test ID | Description | Expected Results | Actual Results |
| Linear | **Preconditions**:  files linear/linear.json and linear/linear.csv exist  **Steps**:  run the command  mpirun -n 4 python main.py -j linear/linear.json -c linear/linear.csv | file “linear/sarah5.txt” contains “This is Sarah’s test sentence.”  file “linear/mo5.txt” contains “This is Mo's test sentence.”  file “linear/hussein5.txt” contains “This is Hussein's test sentence.” | file “linear/sarah5.txt” contains “This is Sarah’s test sentence.”  file “linear/mo5.txt” contains “This is Mo's test sentence.”  file “linear/hussein5.txt” contains “This is Hussein's test sentence.” |
| DAG | **Preconditions**:  files dag/dag.json and dag/dag.csv exist  **Steps**:  run the command  mpirun -n 4 python main.py -j dag/dag.json -c dag/dag.csv | file “dag/sarah5\_1.txt” contains “This is Sarah's first sentence.”  file “dag/sarah5\_2.txt” contains “This is Sarah's second sentence.”  file “dag/sarah5\_3.txt” contains “This is Sarah's third sentence.” | file “dag/sarah5\_1.txt” contains “This is Sarah's first sentence.”  file “dag/sarah5\_2.txt” contains “This is Sarah's second sentence.”  file “dag/sarah5\_3.txt” contains “This is Sarah's third sentence.” |
| Disconnected | **Preconditions**:  files disconnected/disconnected.json and disconnected/disconnected.csv exist  **Steps**:  run the command  mpirun -n 4 python main.py -j disconnected/disconnected.json -c disconnected/disconnected.csv | file “disconnected/sarah\_A2.txt” contains “Sarah A”  file “disconnected/sarah\_B2.txt” contains “Sarah B”  file “disconnected/sarah\_B2.txt” contains “Sarah C” | file “disconnected/sarah\_A2.txt” contains “Sarah A”  file “disconnected/sarah\_B2.txt” contains “Sarah B”  file “disconnected/sarah\_B2.txt” contains “Sarah C” |
| ErrorCycle | **Preconditions**:  files cycle/cycle.json and cycle/cycle.csv exist  **Steps**:  run the command  mpirun -n 4 python main.py -j cycle/cycle.json -c cycle/cycle.csv | ValueError: Pipeline contains a cycle.  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 | ValueError: Pipeline contains a cycle.  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 |
| ErrorDuplicateNode | **Preconditions**:  files duplicate/duplicate.json and duplicate/duplicate.csv exist  **Steps**:  run the command  mpirun -n 4 python main.py -j duplicate/duplicate.json -c duplicate/duplicate.csv | ValueError: Pipeline contains duplicate Task A1  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 | ValueError: Pipeline contains duplicate Task A1  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 |
| ErrorMissingRequired | **Preconditions**:  files missing\_req/missing\_req.json and missing\_req/missing\_req.csv exist  **Steps**:  run the command  mpirun -n 4 python main.py -j missing\_req/missing\_req.json -c missing\_req/missing\_req.csv | KeyError: 'Unknown UID A4\_4 set as requirement for A5\_3'  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 | KeyError: 'Unknown UID A4\_4 set as requirement for A5\_3'  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 |
| ErrorBadJSON | **Preconditions**:  files duplicate/duplicate.json and duplicate/duplicate.csv exist  **Steps**:  run the command  mpirun -n 4 python main.py -j duplicate/duplicate.json -c duplicate/duplicate.csv | ValueError: Expecting , delimiter: line 10 column 5 (char 164)  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 | ValueError: Expecting , delimiter: line 10 column 5 (char 164)  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 |
| ErrorBadCSV | **Preconditions**:  files duplicate/duplicate.json and duplicate/duplicate.csv exist  **Steps**:  run the command  mpirun -n 4 python main.py -j duplicate/duplicate.json -c duplicate/duplicate.csv | ValueError: Not all $$ variables replaced in $$name$$'s  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 | ValueError: Not all $$ variables replaced in $$name$$'s  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 |
| ErrorNoArgs | **Preconditions**:  none  **Steps**:  run the command  mpirun -n 4 python main.py | usage: main.py [-h] -j JSON [JSON ...] -c CSV [CSV ...] [-p CHECKPOINT]  [-r {0,1}] [-d]  main.py: error: argument -j/--json is required  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 | usage: main.py [-h] -j JSON [JSON ...] -c CSV [CSV ...] [-p CHECKPOINT]  [-r {0,1}] [-d]  main.py: error: argument -j/--json is required  application called MPI\_Abort(MPI\_COMM\_WORLD, 1) - process 0 |
| SmallKSRT | **Preconditions**:  files ksrt.json and ksrt\_small.csv exist  **Steps**:  run the command  qsub -I -l nodes=20:ppn=4  run the command  mpirun -n 81 python main.py -j ksrt.json -c ksrt\_small.csv | The expected result must be determined by a domain expert. | Tests could not be completed due to errors in the pipeline. |
| SmallDNA | **Preconditions**:  files dna.json and dna\_small.csv exist  **Steps**:  run the command  qsub -I -l nodes=20:ppn=4  run the command  mpirun -n 81 python main.py -j dna.json -c dna\_small.csv | The expected result must be determined by a domain expert. | The pipeline ran without errors but the results could not be verified because we do not know what the expected output should be. |
| FullKSRT | **Preconditions**:  files ksrt.json and ksrt\_full.csv exist  **Steps**:  run the command  qsub -I -l nodes=64:ppn=4  run the command  mpirun -n 257 python main.py -j ksrt.json -c ksrt\_full.csv | The expected result must be determined by a domain expert. | Tests could not be completed due to errors in the pipeline. |
| FullDNA | **Preconditions**:  files dna.json and dna\_full.csv exist  **Steps**:  run the command  qsub -I -l nodes=64:ppn=4  run the command  mpirun -n 257 python main.py -j dna.json -c dna\_full.csv | The expected result must be determined by a domain expert. | The ARC cluster is simply not large enough to run this test. |

**Unit Test Suite**

For unit testing, we used the Python unittest framework, and Coverage.py for code coverage. As you can see below, Hivemind has over 99% coverage.

tests$ ./unittests.sh

**Master/Worker Tests**

test\_loop\_dry (test\_mpi.TestMPIDryRun) ... ok

test\_loop (test\_mpi.TestMPIMultipleWorker)

Test loop and end state using multiple threads / multiple Workers. ... ok

test\_start\_state (test\_mpi.TestMPIMultipleWorker)

Test Master / all Workers' initial state. ... ok

test\_loop (test\_mpi.TestMPISingleWorker)

Test loop and end state using multiple threads / single Worker. ... ok

test\_start\_state (test\_mpi.TestMPISingleWorker)

Test Master/Worker's initial state. ... ok

**Rank Tests**

test\_rank\_by\_fifo\_ranktree\_pipeline (test\_rank.TestRank) ... ok

test\_rank\_by\_fifo\_tree (test\_rank.TestRank) ... ok

test\_rank\_by\_fifo\_unbalanced\_pipeline (test\_rank.TestRank) ... ok

test\_rank\_by\_successors\_dag (test\_rank.TestRank) ... ok

test\_rank\_by\_successors\_dag\_disconected (test\_rank.TestRank) ... ok

test\_rank\_by\_successors\_linear (test\_rank.TestRank) ... ok

test\_rank\_by\_successors\_loose\_pipeline (test\_rank.TestRank) ... ok

test\_rank\_by\_successors\_ranktree\_pipeline (test\_rank.TestRank) ... ok

test\_rank\_by\_successors\_single\_node (test\_rank.TestRank) ... ok

test\_rank\_by\_successors\_tree (test\_rank.TestRank) ... ok

test\_rank\_by\_successors\_unbalanced\_pipeline (test\_rank.TestRank) ... ok

test\_rank\_by\_total\_successors\_dag (test\_rank.TestRank) ... ok

test\_rank\_by\_total\_successors\_dag\_disconected (test\_rank.TestRank) ... ok

test\_rank\_by\_total\_successors\_linear (test\_rank.TestRank) ... ok

test\_rank\_by\_total\_successors\_loose\_pipeline (test\_rank.TestRank) ... ok

test\_rank\_by\_total\_successors\_ranktree\_pipeline (test\_rank.TestRank) ... ok

test\_rank\_by\_total\_successors\_single\_node (test\_rank.TestRank) ... ok

test\_rank\_by\_total\_successors\_tree (test\_rank.TestRank) ... ok

test\_rank\_by\_total\_successors\_unbalanced\_pipeline (test\_rank.TestRank) ... ok

**Concrete Pipeline Tests**

test\_done (test\_pipeline.TestConcretePipeline) ... ok

test\_mc (test\_pipeline.TestConcretePipeline) ... ok

test\_replace\_back\_to\_back (test\_pipeline.TestConcretePipeline) ... ok

test\_replace\_invalid\_type (test\_pipeline.TestConcretePipeline) ... ok

test\_replace\_invalid\_var (test\_pipeline.TestConcretePipeline) ... ok

test\_replace\_multiple (test\_pipeline.TestConcretePipeline) ... ok

test\_replace\_none (test\_pipeline.TestConcretePipeline) ... ok

test\_replace\_partial (test\_pipeline.TestConcretePipeline) ... ok

test\_replace\_repeated (test\_pipeline.TestConcretePipeline) ... ok

test\_replace\_simple (test\_pipeline.TestConcretePipeline) ... ok

test\_replace\_substring (test\_pipeline.TestConcretePipeline) ... ok

test\_replace\_unicode (test\_pipeline.TestConcretePipeline) ... ok

**Pipeline Framework Tests**

test\_create\_empty\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_cyclic\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_dag\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_disconnected\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_duplicate\_node\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_linear\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_loose\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_ranktree\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_self\_ref\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_single\_node\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_tree\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_unbalanced\_framework (test\_pipeline.TestPipelineFramework) ... ok

test\_unknown\_uid\_framework (test\_pipeline.TestPipelineFramework) ... ok

**Task Tests**

test\_cmd\_error (test\_task.TestTask)

Test task with error in command. ... ok

test\_repr (test\_task.TestTask)

Test task's repr. ... ok

test\_verify\_error (test\_task.TestTask)

Test task with error in verification. ... ok

**Util Tests**

test\_json\_to\_tasks\_uid\_require (test\_util.TestUtil) ... ok

test\_json\_to\_tasks\_uid\_task (test\_util.TestUtil) ... ok

test\_json\_to\_tasks\_uid\_wd (test\_util.TestUtil) ... ok

test\_make\_path (test\_util.TestUtil) ... ok

test\_read\_csv (test\_util.TestUtil) ... ok

test\_read\_csv\_single\_col (test\_util.TestUtil) ... ok

test\_to\_bool\_error (test\_util.TestUtil) ... ok

----------------------------------------------------------------------

Ran 59 tests in 0.592s

Name Stmts Miss Cover Missing

-------------------------------------------------------------

hivemind/mpi/master.py 75 0 100%

hivemind/mpi/worker.py 32 0 100%

hivemind/pipeline/pipeline.py 118 1 99% 198

hivemind/pipeline/rank.py 20 0 100%

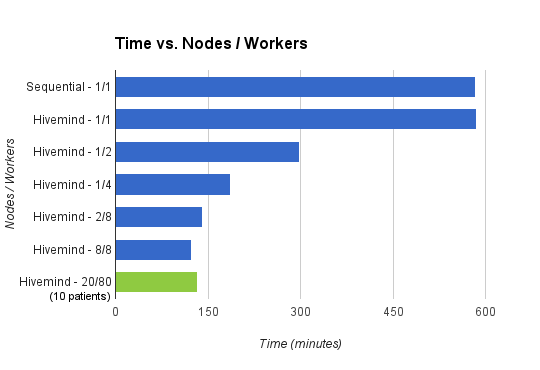
hivemind/pipeline/task.py 49 1 98% 113

hivemind/util.py 45 1 98% 136

-------------------------------------------------------------

TOTAL 339 3 99%

## Performance Testing



Above are the results of our tests with the DNA pipeline. First, we got a baseline time for running the pipeline sequentially with one patient on a single node. Then we tested using Hivemind with an increasing number of workers. As expected, Hivemind with a single worker on a single node has very similar performance to the sequential version of the pipeline. The DNA pipeline has a maximum concurrency of 8 for approximately half of the pipeline, and 2 for the other half. This made it so the running time was almost cut in half when running with two workers on one node, with smaller, but still significant performance gains when going to 4 and 8 workers. Since some of the tasks in the DNA pipeline take over 6GB of memory, we were unable to test with 8 workers on a single node, so we tested with 8 workers on 2 nodes (4 workers per node), and 8 worker on 8 nodes (1 worker per node). Finally, we tested the DNA pipeline with 10 patients using 80 workers on 20 nodes (4 workers per node), and had similar results to the single patient test with 8 workers on 2 nodes.

While we were able to run the DNA pipeline sequentially and using Hivemind without encountering errors, we had no way of validating results. We can only be sure that the output file sizes are comparable between the Hivemind and Sequential runs.

**Suggestions for Future Teams**

The JSON specification requires a lot of upfront work to build. In the future it would be nice to have a drag and drop UI that would visualize and build the pipeline. D3.js would be a good tool for building such a UI.

Apache Spark was a technology that was originally considered for this project. Hivemind’s pipeline module has no dependencies on MPI, and could be reused with Spark. A parallel topological sort would be an effective way splitting up a pipeline for use with a MapReduce architecture.

The NetworkX library can export DAGs in a format that is compatible with D3.js. Using this to build a visualization of Hivemind as it is running would be a cool way to demo the system.

We currently have debug logging that emits the various events that happen during the running of the system. These are quite useful, and human readable. However, if one wanted to know the entire state of the system, a more robust and non-human readable logging format should be used. A [JSON logging format](https://github.com/exoscale/python-logstash-formatter) would be ideal. This could be combined with the previous suggestion to create a robust way to monitor the status of a Hivemind pipeline.

The JSON specification currently only allows verification of results via another executable. This is sufficient in a large number of cases, but not in all cases. Adding the ability to have Python code run both before and after a Task would be a more robust solution. These “processors” would be defined in Python and referenced in the JSON spec.

There is some concern that having a single Master and many Workers does not scale when there is very large number of Workers (i.e. the Master becomes a bottleneck). Adding another layer of hierarchy (super master, sub master, worker) could be used to prevent this issue.

We currently implement checkpointing (with no node level fault tolerance) as way to recover from system failure. In the future it should be possible to at least make Workers fault tolerant (requires the MPI implementation to always return an error on a node failure).

Currently a Task that should be skipped is still added to the Master’s PriorityQueue. This is unnecessary. The Master could be modified so that instead of putting all Tasks in the queue, it instead recursively calls a function that handle these using the finish\_task method and by updating the necessary state variables. This should be done with great care and a large number of unit tests as it has the possibility of introducing subtle bugs into how the Master works. It would be a useful optimization as currently it is possible for a high priority Task to be considered “not ready” because it is waiting on a skipped Task to be processed.

There are three related features that could not be implemented due to time constraints. These include retrying a Task that throws an exception, failing a pipeline once a Task fails too many times, and then reporting on the status of all the pipelines when the system completes all Tasks. [This package](https://github.com/rholder/retrying) could be used to help with the implementation of the retry logic, but it may be overkill. A simple for-loop may be more appropriate. Failing a pipeline could be handled by having the Worker send a specific tag when a Task fails too many times. The Master would then mark that pipeline as failed and remove all nodes from that pipeline’s bipartite graph to remove it’s max concurrency from the system. The PriorityQueue would have to updated to remove any other Tasks from the failed pipeline, or the Master could “learn” to ignore them as they come into play (or perhaps they could be allowed to run). Other Workers that are also working on the same pipeline may need some type of change, or they can be allowed to finish whatever Task they are working on. Regardless, no new Tasks from the pipeline should be added to the queue as it is impossible to finish the pipeline. Since the Master would have to keep track of all of this state anyway, it should be straightforward to summarize this information into a final report at the end of the program.

Currently we close Workers when we no longer need them based on the max concurrency of the system. It should be possible to close the actual MPI node, and even the underlying hardware to reduce resource usage.

We implemented two different ranking methods. There are other possibilities such as variations based on topological sorting. Research should be conducted on how these affect the runtime of different pipelines.

One issue we ran into with the DNA pipeline was that we had to limit the number of workers per physical machine to 4, since one of the binaries took too much memory to run more on one worker, causing the binary to crash. We thought about the possibility of having an option for specifying an expected amount of memory for each task to use, which might be used to prevent overloading a worker.

We have a mechanism for validating results, but we had no parameters for validating the DNA and KSRT pipelines. These should be validated to ensure that the output of these pipelines is actually correct.