A Profiling Mechanism for dispel4py

Anqi Lu

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2015

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Anqi Lu*)

To everybody I met here in Edinburgh.

Abstract

This project built an infrastructure to profile performance of dispel4py workflows. The profiling mechanism consists of profiling framework (PF) and performance database (PDB). PF captures workflow characteristics (i.e. timings, memory, data stream properties, topology of workflow), and organizes raw data before permanently store it. Then PF flushes processed data to PDB, which manages memory, topology of workflow and other characteristics respectively. Future analysis on the historical data can be made on PDB to improve the efficiency of dispel4py workflows. This profiling infrastructure now works in the Multi-Processing distributed computational infrastructure. To illustrate the usage and value of this mechanism, Pipeline Test workflow and Internal Extinction of Galaxies workflow were profiled and analyzed.

Contents

Chapter 1

Introduction

Data-intensive computing, assumed as the fourth Paradigm[8], is getting its popularity in many research fields these days. Such computational applications involve a large volume of data when enacting and spend most of processing time on I/O and manipulation of data. As a result of growing applications of data-intensive computing, heterogeneity, scale and complexity of data involved in scientific research increase fast. Consequently, to automatically handle the low-level data processing, scientific workflow systems emerge. Scientific workflow systems allow scientists to focus on high-level infrastructure (i.e. logical level of computations) rather than being distracted by implementation details of a computation on distributed systems[9].

The performance data of scientific workflows and scientific workflow systems helps researchers to better understand the workflow enactments in practice. It can be examined for errors and exceptional behaviors, analyzed for optimization, mined and processed for a profile of a certain workflow.

dispel4py[1] is a novel scientific workflow system, which translate abstract workflow graphs into a couple of mappings, including Apache Storm, MPI, shared-memory Multi-Processing and Sequential Mode. It enables users to use a fine-grained workflow to describe computational tasks. Additionally, dispel4py streams data in the graph, which avoids writing intermediate values to disk except when buffers overflow RAM.

This dissertation presents a profiling mechanism for dispel4py workflow system in multi-processing distributed environment. The goal is to profile each computational block of a workflow in following aspects: time cost, memory occupation and data stream properties (i.e. data stream rate, data size, data type). Performance data is captured and processed by profiling framework (PF) , an infrastructure to track runtime workflows. Then, raw data collected from intercepting certain parameters is gathered and used to build a performance database (PDB) . Further study is made in the characterization statistics for learning about the workflows. To illustrate an implementation of the profiling mechanism, benchmark profiles were built for two workflows, *Pipeline Test* workflow and *Internal Extinction of Galaxies* workflow, working with Multi-Processing mapping, on local server and clusters, respectively.

Chapter 2

Background

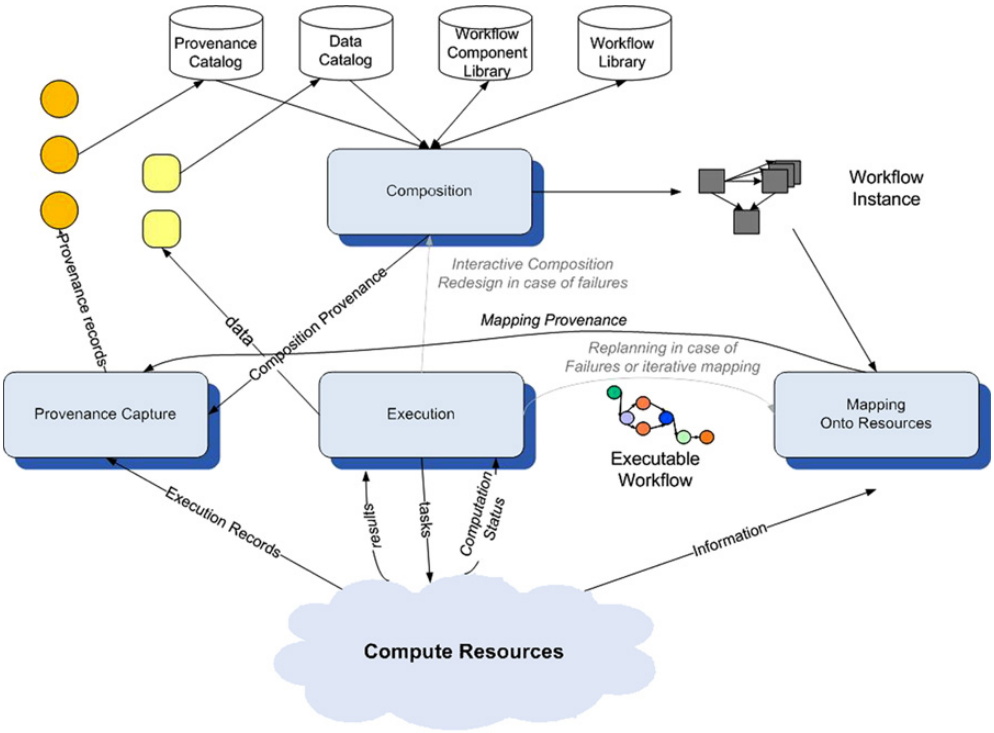

Figure 2.1: The workflow lifecycle: composition, mapping, execution, and provenance capture (Deelman, E. 2009)

2.1 Scientific Workflows

As complex and vast computations are developing rapidly, there emerge requirements for computation management. Specifically, imagine there is a bunch of repetitive operations involving inputting data to a supercomputer, a sequence of processing on data, and outputting data to somewhere. Apparently, it requires automatic management for the movement of the data and the control of the repetitive cycle. This is exactly the purpose of scientific workflow systems, which aims at automating this cycle to enable scientists focus on algorithms or services themselves instead of computation infrastructure[2].

There are a couple of concepts to clarify about *workflow*:

Workflow Orchestration refers to the activity of defining the sequence of tasks needed to manage a business or computational science or engineering process.

A *Workflow* is a template for such an orchestration.

A *Workflow Instance* is a specific instantiation of a workflow for a particular problem and includes the definition of input data.

A workflow is made up of a set of tasks and dependencies between each task. Furthermore, the dependencies between tasks must be satisfied to ensure the workflow work as expected. As is said in [4], the goal of e-Science workflow systems is to provide a specialized programming effort required by scientists to orchestrate a computational science experiment. That is also the aim of dispel4py, which is the e-Science workflow system being studied in this paper and will be introduced in the next subsection.

As is illustrated in Figure 2.1, the lifecycle of a workflow can be classified as four phases:

1. **Composition:** Certain workflow is defined in this phase by users. The composition phase allows the specification of steps and dependencies of a workflow.

   Workflows can be described either directly or with a template. In the former case, users specify a workflow with computational steps and the specific data. In contrast, in the latter case, users first create a template and then instantiate it with actual data.

   Depending on the level of composition tasks, workflow systems can be classified as task-based or service-based. Task-based systems focus on the resource-level features, leaving higher-level tasks to other tools, while service-based ones provide interfaces to certain classes of service for management and composition.

2. **Mapping:** In this phase, the description, created in the previous phase, is used to build an executable workflow.

   Both task-based systems and service-based systems involve finding necessary resources/ services to execute computation, and optimization. Moreover, task-based systems may modify the original workflow in this phase.

3. **Execution:** The workflow is executed in this phase.

4. **Provenance:** Data provenance refers to a record of the history of the creation of a data object.

2.2 dispel4py

dispel4py[1] is a Python library used to describe abstract workflows for distributed data-intensive applications. Data-intensive applications refer to applications that are I/O bound or with a need to process large volumes of data[3]. Due to the size of input data, the processing time of these specific applications is mostly consumed at the following phases: data input, data movement, data manipulation, and data output.

The purpose of dispel4py is to enable users to implement manipulations on big data without being distracted by details of the computing framework. To achieve this, dispel4py works in a dataflow-oriented way. With data streaming, it therefore can process requests with large-scale data by an efficient implementation of buffering in the main memory where the processing speeds of memory access outperform the disk by a factor of more than 10^5 [10]. Dispel4py workflow can be viewed as a light-weight composition of data operations. Data is streamed from one computational operation to another. That means any intermediate I/O cost is avoided.

Dispel4py instantiate workflow with a *graph* customized by users. A graph defines the ways in which computational blocks are connected and hence the paths taken by data, i.e. the topology of the workflow. A *processing element* (PE) can be assumed as a basic computational block, which encapsulates an algorithm, a service or another data transformation process. It allows users to describe their tasks in each PE in Python script and then define the connections of each PE, which is called graph. In a workflow, each PE is connected with data stream. A *processing element instance* (PEI) is the executable copy of a PE instantiated in runtime workflow.

According to different infrastructure of workflows, there are four types of PEs, which are composed of *generic PE*, *iterative PE*, *simple function PE* and *iterative chain*. Generic PE allows multiple input data flows and multiple output data flows while the others just allow single input and single output. The compositions of a variety of PEs in a graph enables the

merging and splitting of data streams as desire. Additionally, different PEs can be composed together to form a *composite PE*, which allow for synthesis of increasingly complex workflows. In parallel environment, a dispel4py *partition* refers to several PEs running in a single process.

dispel4py currently supports following mappings: Apache Storm, MPI, Multi-Processing and Sequential Mode. Apache Storm is a distributed realtime computation system. The dispel4py system maps to Storm by translating its graph description to a Storm topology. MPI, Message Passing Interface, is a standardized and portable message-passing system. dispel4py uses mpi4py, a Python binding for MPI based on the MPI-2 standard to map PEs to a collection of MPI processes. Compared with message passing model, Multi-Processing, a Python package that supports spawning processes, works in a memory sharing way to commit parallel tasks. Sequential mode (simple) is a standalone mode that is ideal for small applications running on local server. The first three mappings are designed for distributed resources while the last one acts as baseline for workflow development. The performance profiling and evaluation in this project is mainly focused on Multi-Processing.

2.3 Related Work on Profiling Scientific Workflows

Many efforts have been made in the study of scientific workflow profiling. An workflows profiling method was introduced in [5]. Juve and Chervenak developed *wfprof*, a workflow characterization tool-kit, to record and analyse performance metrics for workflows. The workflow parameters studied in this research were composed of process I/O, runtime, memory and CPU (Central Processing Unit) utilization. In [2], a taxonomy was proposed to characterize and classify methods for enacting workflows in Grid computing. Callaghan et al. proposed a novel method to accurately capture the scale of scientific workflows and quantify their efficiency in [4]. Detailed task-level characterization was studied in their research. Profiles built from historical enactments of workflows could be used in optimization of future workflows along with scheduling algorithms [6].

Performance database was introduced by Liew et al. in [7]. A performance database (PDB) refers to a database storing performance-related data gathered during workflow enactment. To describe how performance data is collected and transformed into information, a four-stage lifecycle (collect, organize, use and discard) was introduced in this paper. The concept of PDB in our study is different with what they proposed. PDB is permanently stored on our disk instead of being discarded after analysis. This enables further usage on the history performance data for workflows working in the similar conditions, such as topology of the workflow network.

Chapter 3

dispel4py Performance Profiling Mechanism

3.1 Overview on Performance Profiling Mechanism

dispel4py performance profiling mechanism is designed for better understanding the efficiency of enactments of workflows, and optimizing it. In terms of architecture, it consists of two parts: profiling framework (PF) and performance database (PDB).

During runtime of each workflow, PF collects, analyzes and stores data to PDB. Future decision makings (e.g. modifying PE to save memory space and execution time), will be based on analysis of the historical data in PDB. Performance profiling mechanism works in three phases:

1. PF captures data from runtime workflow
2. PF analyzes and stores data to PDB
3. Users and developers manually analyze historical data in PDB to optimize PE design and processes allocations

PF is used for observing and analyzing data during workflow runtime, while PDB is used for performance data storing, management, and future analysis.

3.2 Profiling Framework

3.2.1 Overview of Profiling Framework

In this section, the way how the profiling framework (PF) works will be presented.

PF profiles following parameters of a workflow: timings, memory, data type, data size, rate of data streams, and deployments and connections of processing elements. The framework is activated with the initialization of the workflow, and terminated with the end of the workflow.

In terms of lifecycle, PF has four main stages: capturing performance data in *Processing Elements Instances*, parsing data back to Monitor instance, analyzing data in background, and flushing data to PDB.

There are two exceptions of the common 4-stage life cycle of PF. The first one is when PF profiles memory usage, it skips the second and the third stages. In particular, the memory usage is accessed and logged for a certain frequency, which can be customized by developers, to RAM (random access memory). After the memory usage data reaches a specific size in memory, which can also be set by developers, the data is flushed to PDB on disk directly. The advantage of this strategy is balancing memory cost and I/O cost of profiler, as the size of memory usage information for a whole workflow can be enormous, depending on the customized rate on which the profiler inquires info. The second one is when PF profiles graph of a workflow, it skips the first and the second stages. The graph is accessed and flushed to disk directly.

As mentioned above, PF consists of two stages. The first two stages of PF are completed with hard-coding in dispel4py system, `dispel4py.new.processor` and `dispel4py.new.multi_process`. The last two stages are implemented in `dispel4py.new.monitor_workflow`, an independent Python module. These three files are tagged out in following file tree of dispel4py project.

dispel4py project files structure

```
[dispel4py]
├── __init__.py
├── __main__.py
├── base.py
├── core.py
├── new
│   ├── __init__.py
│   ├── aggregate.py
│   ├── aggregate_test.py
│   ├── mappings.py
│   ├── monitor_workflow.py
│   ├── mpi_process.py
│   ├── mpi_process_test.py
│   ├── mpi_queue_process.py
│   ├── multi_process.py
│   ├── processor.py
│   ├── simple_process.py
│   └── spark_process.py
└── storm
    ├── __init__.py
    ├── client.py
    ├── storm_submission.py
    ├── storm_submission_client.py
    ├── topology.py
    └── utils.py
```

Figure 3.1: dispel4py project files structure

As is shown in the file tree (Figure 3.1), the main program of dispel4py is `dispel4py.new.processor`. The whole workflow system starts up from here, with a command line in following format:

```
dispel4py [-h] [-a attribute] [-f inputfile] [-d inputdata] [-i iterations] [-p] target
 ↪  module
```

Listing 1: Command line to enact a dispel4py workflow

The argument `target` indicates an execution platform (also called as *mappings*) for the workflow. Available *mappings* include: MPI, Apache Storm, Multiprocessing, and Sequential mapping. Depending on the distributed computational platform that the user specifies for `target`, the program invokes one of the four modules:

- `dispel4py.new.mpi_process`
- `dispel4py.new.multi_process`
- `dispel4py.new.simple_process`
- `dispel4py.storm.storm_submission`

The profiling module is toggled by command argument, `--profileOn` or `-p`. Workflow will only be monitored with this flag set when running dispel4py, e.g.

```
python -m dispel4py.new.processor multi your_workflow.py -n 6 -i 1 -p
```

Listing 2: Command line to run a dispel4py workflow with profiling infrastructure

As is mentioned, the profiling mode can be switched by the `--profileOn` in command line. When profiling mode is on, following steps are taken to monitor the enactment of the workflow.

A `Monitor` is a Python class defined in `dispel4py.new.monitor_workflow`. It is responsible for storing performance data (excluding memory usage and graph) temporarily in memory, analyzing data in background and writing data to PDB.

`Monitor` is instantiated at the beginning of the workflow enactment. The `Monitor` is initialized with following parameters:

1. `profiles`, a dictionary proxy which is initiated in the main process.
   This dictionary proxy acts as the container for all relational performance data (timing, data stream property and workflow config). Key in `profiles` is the index for the performance data, with PEI id and name of operation measured. Value in `profiles` is the corresponding benchmark captured in the PEI. It shares data between multiple processes in a monotonically increasing manner. The key-value pairs in `profiles` will never be modified or deleted. When a new performance data parsed to `profiles`, it simply adds a new key-value pair to the dict proxy. In this way, lock is avoided, which significantly improves speed of sharing variables.

2. `args`, arguments running the workflow from command line. The `Monitor` gets configs about the workflow from `args`. These configs include: workflow name, workflow mapping, number of processes assigned to workflow, and workflow iteration number.

3. `workflow`, dispel4py workflow instance. Parameter `workflow` contains the graph as a property. Thus the `Monitor` needs it for profiling the graph.

After initialization, the `Monitor` gathers performance data from each child process, where PEI runs.

Then, the `Monitor` analyzes data in background, after the termination of the workflow. It derives performance information for PE from data captured from PEI.

After all, `Monitor` records performance data of the workflow, the PEs and the PEIs respectively to PDB.

In the following sections, the way of profiling specific characteristics will be introduced.

3.2.2 Timing

PF tracks a variety of timing characteristics of workflows, PEs and PEIs. Particularly, it tracks runtime of a workflow, average operation (including execute, read, write, process, and terminate) time of a PE, and operation time of a PEI. All the time cost is captured as seconds.

Time cost for each activity carried by a PEI is observed by computing the time difference between the activity begin timestamp and the activity end timestamp. Time cost, t_{cost}, refers to the time taken for a PEI to complete a specific operation, e.g. read, write, process. PF derives t_{cost} from the operation begin timestamp, t_{begin}, and the operation end timestamp, t_{end}, for PEIs.

For PEI_i, its time cost t_{cost_i} for each action, assuming this action has been executed for n times, are computed as follows:

- reading time cost of PEI_i:

$$t_{read_i} = \sum_{j=1}^{n} (t_{endread_{i\_j}} - t_{beginread_{i\_j}})$$

- writing time cost of PEI_i:

$$t_{write_i} = \sum_{j=1}^{n}(t_{endwrite_{i\_j}} - t_{beginwrite_{i\_j}})$$

- processing time cost of PEI_i:

$$t_{process_i} = \sum_{j=1}^{n}(t_{endprocess_{i\_j}} - t_{beginprocess_{i\_j}})$$

- terminate time cost of PEI_i, given `terminate()` just runs once for each PEI:

$$t_{terminate_i} = t_{endterminate_i} - t_{beginterminate_i}$$

- total time cost of PEI_i:

$$t_{total_i} = t_{endpei_i} - t_{beginpei_i}$$

Compared with the time cost of a PEI, time cost of a PE, T_{cost}, does not refer to the actual time cost of a PE. Because of the parallel processing, the time cost of a PE in reality, derived from time difference of the end timestamp of the last ending PEI and the begin timestamp of the first beginning PEI, $Max(t_{endpei_i}) - Min(t_{beginpei_i})$, is meaningless.

Therefore, to better figure out how much time has been spent in each computation for a specific PE, the `Monitor` defines time cost of each PE as the average of the time costs of corresponding activities in all its instances.

Assuming PE_I has N runtime instances, the cost can be derived by following equation:

$$T_{cost_I} = \frac{\sum_{i=1}^{N} t_{cost_i}}{N}$$

Following equations illustrate how time performance in different angles of a PE is computed.

- reading time cost of PE_I:

$$T_{read_I} = \frac{\sum_{i=1}^{N} t_{read_i}}{N}$$

- writing time cost of PE_I:

$$T_{write_I} = \frac{\sum_{i=1}^{N} t_{write_i}}{N}$$

- processing time cost of PE_I:

$$T_{process_I} = \frac{\sum_{i=1}^{N} t_{process_i}}{N}$$

- terminate time cost of PE_I:

$$T_{terminate_I} = \frac{\sum_{i=1}^{N} t_{terminate_i}}{N}$$

- total execution time cost of PE_i:

$$T_{total_I} = \frac{\sum_{i=1}^{N} t_{total_i}}{N}$$

Additionally, runtime of the whole workflow is tracked. The begin timestamp, τ_begin, of a workflow is logged at the beginning of the `process()` function in multi-processing mapping, right after the initialization of the `Monitor`. The end timestamp, τ_end, of the workflow is logged after the termination of the whole computation, which is viewed as putting `TERMINATED STATUS` to the final results queue, or the joining of the last PEI process to the main process.

Based on the begin timestamp and the end timestamp, total execution time of $Workflow_\alpha$ is:

$$\tau_\alpha = \tau_{end_\alpha} - \tau_{begin_\alpha}$$

3.2.3 Memory

In computing, the memory taken by a process consists of Resident Set Size (RSS) and Virtual Memory Size (VMS) . RSS refers to the part of memory that is held in Random Access Memory (RAM). As occupied memory could page out, or some parts of the executable might never be loaded, the other portions of memory taken is in the swap space or file system. VMS stands for pages of data temporarily transferred to disk from RAM by operating system, to compensate for shortages of physical memory.

In this case, the memory size to monitor is RSS, instead of VMS or the sum of VMS, since RSS is the non-swapped physical memory used by each process, minus a small amount of overhead, which gives the user a better idea of the actual space efficiency of current workflow.

To measure the memory usage of dispel4py, PF tracks three metrics for each PEI: peak memory of process, peak memory of read, peak memory of write. In terms of tracking implementation, to reduce the memory burden on memory profiling process itself, memory usage data is flushed to files, separate from CSV files which store other performance data if data stored to files instead of database system. The specific data collecting strategy will be explained in *dispel4py Workflow Performance Data Collecting* chapter of this dissertation.

Memory usage information is computed by the function `memory_usage()` in `dispel4py . new . monitor_workflow`. `memory_usage()` takes an input signature consists of following parameters:

- `proc`, the process to monitor
- `interval`, interval at which measurements are collected
- `timeout`, maximum amount of time (in seconds) to wait before returning
- `timestamps`, flag to indicate whether to collect timestamps of memory usage measurement
- `include_children`, flag to indicate whether to additionally profile children processes of the process monitored
- `max_usage`, flag to indicate whether to collect all memory usages during interval or compute the maximum
- `retval`, flag to indicate whether to save the return value of the function being profiled
- `stream`, output file for collection of data
- `description`, description of process being monitored

`memory_usage` supports both memory measurements for processes specified by function name and by process id. In multi-process environment, the process to profile is specified by process id.

The memory profiler will get memory at the specific frequency, which can be computed by the given `interval` and `timeout` parameters. For instance, if the parameters are input as $interval = 1^{-6} second$, and $timeout = 1^{-3} second$, then the profiler will inquire current memory information for 1,000 times in 0.001 second, with the rate at 1,000,000 times per second.

Apparently, if all the collected data is stored to memory, the `memory_usage` process will be under heavy memory burden. To balance between memory cost and I/O cost, the profiler temporarily stores data to memory, and then flushes it to disk in every 50 tuples.

`memory_usage` invokes `psutil` to inquire memory information, if `psutil` can be imported correctly. Otherwise, it tries to retrieve memory usage with `ps`, run by `subprocess` in a new process.

`psutil` (python system and process utilities) is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network) in Python. It is useful mainly for system monitoring, profiling and limiting process resources and management of running processes. With `psutil`, memory usage of current process can be accessed as follows:

```
import psutil

# Instantiate psutil.Process
process = psutil.Process(os.getpid())

# Retrieve RSS, which stored as the first element in tuple returned
memory = process.memory_info()[0] / float(2 ** 20)
```

Listing 3: Access memory info with psutil

The variable `memory` contains the memory usage as value. The returned memory usage is in MiB. With a deeper study in the resource code of `psutil`, it uses different ways to get information with consideration to different platforms. In this case, two platforms are involved, *darwin* and *linux2*. *Darwin* is an open source Unix computer operating system released by Apple Inc. It is composed of code developed by Apple, as well as code derived from NeXTSTEP, BSD, and other free software projects. *Linux2* refers to Linux in version 2. PF runs in *darwin* (OSX) on local server, while in *linux2* (Linux) on cluster. Memory usage is retrieved as follows:
A. For *darwin* (OSX) platform, the method is illustrated in Listing 4.

```
@wrap_exceptions
def memory_info(self):
    rss, vms = cext.proc_memory_info(self.pid)[:2]
    return _common.pmem(rss, vms)
```

Listing 4: Retrieve memory info on darwin

B. For *linux2* (Linux) platform, the method is illustrated in Listing 5.

```
@wrap_exceptions
def memory_info(self):
    with open("/proc/%s/statm" % self.pid, 'rb') as f:
        vms, rss = f.readline().split()[:2]
        return _common.pmem(int(rss) * PAGESIZE, int(vms) * PAGESIZE)
```

Listing 5: Retrieve memory info on linux2

`ps` is an UNIX command line tool which gives a snapshot of the current processes. As it is possible that the `psutil` library cannot be correctly imported, `ps` will be used in such situation, to ensure the `Monitor` keeps working. In command line, memory info can be accessed with command below: For the example showing in Listing 6, the returned result will be in template as Listing 7:

In `memory_usage`, `subprocess` module is used for executing the command (Listing 8) from another process inside the program, and then returned data in template as illustrated in Listing 7 is analyzed.

3.2.4 Data Type and Data Size

Data type and data size are key variables to take into consideration, when developers study into the performance of a specific workflow. Since properties of data streamed directly affect

```
# Assuming getting memory info for processor with pid 7464
ps v -p 7464
```

Listing 6: Command of getting memory info for processor with pid 7464

```
  PID STAT    TIME  SL  RE PAGEIN      VSZ    RSS   LIM  TSIZ  %CPU %MEM COMMAND
 7464 T    0:00.02   0   0      0  2464480   4940     -     0   0.0  0.0 python
```

Listing 7: Result of getting memory info for processor with pid 7464

characteristics such as runtime, memory and data streaming efficiency.

These two data properties are captured during runtime, for both input data streams and output data streams. Each time when a PEI gets an item, $item_{in_i}$, from input queue, the data type and the data size are measured (, and accumulated for the latter one) before any other operations. In terms of outputs, the measurements are taken once an output, $item_{out_i}$, is generated, before being put to the output stream. Python built-in function `type()` and method `sys.getsizeof()` are used respectively. The class `dispel4py.processor.GenericWrapper` provides method `process()` to observe data type.

For each PE, data type is monitored for both input data and output data. If a PE has several input/output (I/O) streams, data types on all entries/exits will be monitored. The recorded value will be a string joining all types together with comma as separators.

For each PEI, the data it reads from inputs (or writes to outputs) will be analyzed for data type, with Python built-in function `type(object)`. Then the value, which retrieved by the property `__name__` will be added to input or output data type sets. The sets are stored in memory, until it is transferred to the `Monitor`. In monitor, a string will be computed from data type sets, before storing to disk. The pseudo code of gathering data type is shown as displayed in Listing 9. To give a clearer view of data type collecting method, irrelevant operations have been hidden.

In the analysis and record phase, data type of each PEI need to be gathered together to compute the final result for a PE, since it could happen that, for a PE with multiple input streams, one PEI did not get any data from some of those streams, thus the data types it collected were not integrity.

Besides data type, data size has a heavy influence on the benchmarks of a workflow, as well. Theoretically, runtime of future workflow can be predicted given the data size and historical data size - runtime information.

Similar to the profiling on data type, data size is monitored in the `process` method of class `dispel4py.processor.GenericWrapper`. The sizes of the items which a PEI gets from input streams (or write to output streams) are accumulated together and returned as the total input/output data size of this PEI.

The pseudo code in Listing 10, irrelevant parts hidden, explains the way how data size is captured for PEI and then computed for PE.

In the analysis phase, input/output data size of PEIs of a PE is summed up to get the input/output size for the PE.

3.2.5 Rate of Data Stream

To figure out the speed of data streams between two PEIs, `Monitor` also profiles the efficiency of `Queue`, which is a class defined in `multiprocessing` module. Assuming PEI_i reads q_{input_i} items in time t_{read_i}, and writes q_{output_i} in time t_{write_i}, the rate of accessing I/O data, r_{IO_i}, in the `Queue` are as follows:

- Average Speed of PEI_i reading data:

$$r_{input_i} = \frac{q_{input_i}}{t_{read_i}}$$

```
import subprocess

# run command "ps v -p pid" and get returned value
out = subprocess.Popen(['ps', 'v', '-p', str(pid)],
 ↪  stdout=subprocess.PIPE).communicate()[0].split(b'\n')

# retrieve column index of RSS
vsz_index = out[0].split().index(b'RSS')

# retrieve RSS value in MiB
mem = float(out[1].split()[vsz_index]) / 1024
```

Listing 8: Use subprocess module to access memory

```
# Inside process method of dispel4py.processor.GenericWrapper
# Use Set to store data types to avoid duplicate values
in_data_types = set()
out_data_types = set()
# Continuously reading and writing data
while status != STATUS_TERMINATED:
    # Check inputs validity
    if inputs is not None and inputs:
            # Get data type of input value
            in_type = type(inputs.itervalues().next()).__name__
            # Collect input data type to set
            in_data_types.add(in_type)
            # Generate output through customized PE description
            outputs = self.pe.process(inputs)
            # Check whether output value is empty
            if outputs:
                # Get data type of output value
                out_type = type(outputs.itervalues().next()).__name__
                # Collect output data type to set
                out_data_types.add(out_type)
# Return a string format value which contains all data types gets/ delivers
self.in_data_type = ",".join(in_data_types)
self.out_data_type = ",".join(out_data_types)
```

Listing 9: Get data type

- Average Speed of PEI_i writing data:

$$r_{output_i} = \frac{q_{output_i}}{t_{write_i}}$$

Given average rate of I/O data, r_{IO_i}, of its N PEIs, the average rate, R_{IO_I}, of accessing data of PE_I can be derived as follows:

- Average Speed of PE_I reading data:

$$R_{input_I} = \frac{\sum_{i=1}^{N} r_{input_i}}{N}$$

- Average Speed of PE_I writing data:

$$R_{output_I} = \frac{\sum_{i=1}^{N} r_{output_i}}{N}$$

As dispel4py PEI reads/writes one item at once, the quantity of items it accesses in total, is equal to the number of times that the read/write operation has been executed. To be more

```
# Inside process method of
# dispel4py.processor.GenericWrapper
# Initialize total input data size
total_in_size = 0
# Initialize total output data size
total_out_size = 0
# Continuously read and write data
while status != STATUS_TERMINATED:
    # Check inputs validity
    if inputs is not None and inputs:
            # Accumulate input data size
            total_in_size += sys.getsizeof(inputs)
        # Generate output through customized PE description
        outputs = self.pe.process(inputs)
        # Check whether output value is epmty
        if outputs:
            # Accumulate output data size
            total_out_size += sys.getsizeof(outputs)
# Transfer the references of I/O data size back to monitor
self.in_data_size = total_in_size
self.out_data_size = total_out_size
```

Listing 10: Get data size

precisely, in consideration of the last element a PEI reads/writes should be a termination signal, the actual equation should be:

$$\text{Quantity of I/O Items of a PEI} = \text{Number of Times Read/Write Method of the PEI Invoked} - 1$$

But in this case, this difference is not considered, as in data intensive environment, the inaccuracy can be ignored.

The methods `_read()` and `_write()` of class `dispel4py.processor.GenericWrapper` are wrapped with a decorator `calls_count`. This decorator monitors the times of `_read()` and `_write()` have been called, and keeps track of these counts.

To be specific, the decorator, shown in Listing 11, is activated each time when `_read()` or `_write()` is invoked. Then, it checks the function/method name and decides which counter, `_read_calls` or `_write_calls`, to increment in the instance of `GenericWrapper`. After increments the corresponding counter, the decorator executes the function/method and then returns the value, if there exists.

```
def calls_count(f):
    def wrapped(*args, **kwargs):

        if isinstance(args[0], object):
            if hasattr(args[0], "_read_calls") and f.__name__ == "_read":
                setattr(args[0], "_read_calls", getattr(args[0], "_read_calls") + 1)
            if hasattr(args[0], "_write_calls") and f.__name__ == "_write":
                setattr(args[0], "_write_calls", getattr(args[0], "_write_calls") + 1)

        return f(*args, **kwargs)
    return wrapped
```

Listing 11: Decorator calls_count which counts times an operation has been performed

The evaluations of `calls_count` decorator are shown as Listing 12

The average speed can be computed as the quotient of result of items accessed divided by total time of reading/writing. (Listing 13)

The `read_rate` and `write_rate` returned above refer to the properties of a PEI. With multiple input streams, `read_rate` represents the average speed of getting items out of all these input streams which this PEI has access to. The same applies to the `write_rate`.

```
@calls_count
def _read():
    ...

@calls_count
def _write():
    ...
```

Listing 12: Implementation of calls_count decorator

```
# Inside process method of dispel4py.processor.GenericWrapper

if self.read_time != 0:
    self.read_rate = self._read_calls / self.read_time
if self.write_time != 0:
    self.write_rate = self._write_calls / self.write_time
```

Listing 13: Average speed computation

With the average rate over the all entries, it clearly shows the I/O performance of each PEI (and PE), after the later data analysis.

The average I/O speed will then be transferred back to the `Monitor`. In the `Monitor`, the average I/O speed of PE are computed, based on the corresponding PEI speed values.

3.2.6 Deployments and Connections of PEs

The deployments and connections of PEs can be described by the graph of the workflow. In this section, the way how the `Monitor` profiles and stores the graph will be discussed. Also, alternative solutions, such as graph databases and relational databases, will be compared with the method used.

The description of the graph of a workflow can be generated with function `draw` from the `dispel4py.workflow_graph` module. The returned value is in *DOT* format, which is a plain text graph description language. The DOT graph description is written to a special directory which is specially used for storing graph files for each workflow. Then, the path leading to the graph file is stored to the workflow profile table or workflow profile CSV file, depending on a database system is used or file system is used.

There are alternative solutions for storing graphs. Apart from converting the `NetworkX` graph object to plain text, graph databases can be a good option. In computing, a graph database is a database that uses graph structures for semantic queries with nodes, edges and properties to represent and store data. There are a couple of well-know graph database project, such as Neo4J, agamemnon, GraphDB, and AllegroGraph.

Compared with DOT files, graph databases provide graph-like queries into graphs, which is apparently not supported in files. For instance, through query, the shortest path from node X and node Y can be easily computed.

Nevertheless, a graph stored in a graph database takes much more space than what the same graph converted into DOT file takes.

In the case of dispel4py monitor, DOT files overtake graph databases. As the graph of a workflow can be basically classified as Pipeline, Tree, Split and Merge, and unconnected, it is potentially not to be large scaled and complicated. The feature of performing queries over a graph database, in price of space complexity, is not necessary.

In contrast to the graph databases mentioned above, relational databases are relatively unreasonable for storing a graph. Nodes/Vertices need to be stored in one table, while edges, referencing a FromNode and a ToNode to convert a graph data structure to a relational data structure, need to be stored another table. It ends up in a large number of lookups. Also, it involves complex query statements for inserting the graph in Python program, which surely increases the difficulty to maintain it.

3.2.7 Configuration for Profiling Framework

The PF, through a configuration file, gives developers and users control and flexibility over the way it runs. The configuration file is a JSON file that can be changed as needed. This file includes configuration settings that Python runtime reads, and settings that dispel4py system can read.

Developers and users can modify preferences in configuration file, eliminating the need to recompile the whole application every time a setting changes. Also it separates the settings from source code to allow users with low-level privilege to customize the PF.

So far, there are five values can be customized by users, of which the keys are shown as follows:

- `mysql_db_config`: configuration for connection between dispel4py and MySQL database. It takes effect when monitor uses database to store performance data.
- `workflow_profile_store`: path to store performance data on disk. It takes effect when monitor stores performance data in CSV file.
- `graph_profile_store`: path to store the graph of a workflow.
- `store_png`: a flag which indicates whether to convert graph into PNG file and store it.
- `memory_profile_store`: path to store the memory usage data of a workflow.

The configuration file is located at `%HOME%/workspace/dispel4py/config.json` , by default.

3.3 Performance Database

3.3.1 Overview of Performance Database

Performance database (PDB) is a virtual concept, as performance data comes from different data source. It refers to all performance data, which is collected from an enactment of a workflow. The data sources of PDB are composed by three parts:

1. Relational Performance Data
   This part consists of timings, data stream properties and workflow configurations. In order to make it convenient to refer to these data, we call it `relational performance data`. These data could be stored to either database system or file system, depending on different platforms. If MySQL database is installed, and required database connection module can be successfully imported to PF, relational performance data will be stored to MySQL databse system. Otherwise, these data is arranged to stored to CSV files on disk.
2. Memory
   Memory data is stored in plain-text file on disk.
3. Graph
   Graph data is stored as DOT file on disk. If the flag of storing PNG file is set to TRUE in PF configurations, PNG file will be stored, as well.

3.3.2 Timings, Data Stream Properties and Workflow Configs

As is illustrated in Figure 3.2, the relational performance data is classified to 3 tables: *WorkflowProfiles*, *PEProfiles* and *PEInstanceProfiles*.

Table *WorkflowProfiles* stores performance and configs of a workflow, including workflow submission ID, workflow name, workflow mapping, workflow processor number, workflow iteration number, workflow submitted timestamp, workflow runtime, workflow graph path, workflow memory file path.

Table *PEProfiles* stores estimated performance data, which is computed based on PEI characteristics. Its columns/properties are made up of: PE ID, workflow submission ID, PE average

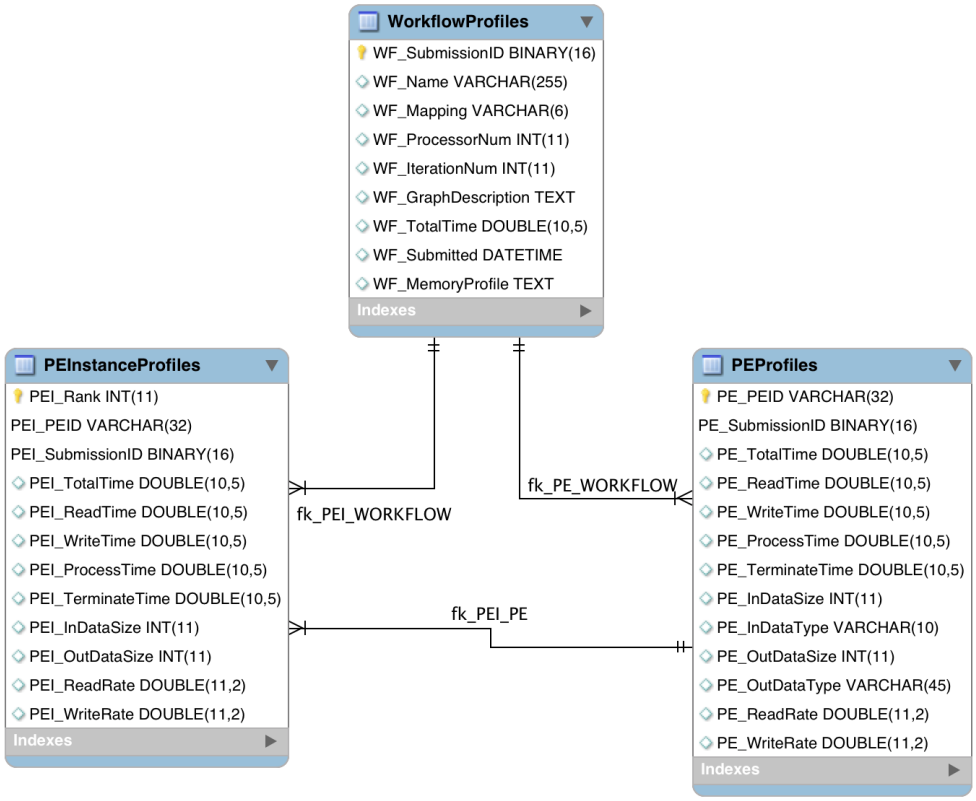

Figure 3.2: The enhanced entityrelationship (EER) model of relational performance data in MySQL database

execution time, PE average read time, PE average write time, PE average process time, PE average terminate time, PE total input data size, PE input data type, PE total output data size, PE output data type, PE average read rate, PE average write rate.

Table *PEInstanceProfiles* stores benchmarks collected from runtime workflow from each PEI. The columns are PEI ID, PE ID, workflow submission ID, PEI execution time, PEI read time, PEI write time, PEI process time, PEI terminate time, PEI input data size, PEI output data size, PEI read rate, PEI write rate.

In terms of CSV files storing, relational performance data is stored to 3 files, named as "SubmissionID-wf.csv", "SubmissionID-pe.csv" and "SubmissionID-pei.csv" respectively, for a workflow with $SubmissionID$.

File *SubmissionID-wf* contains the characteristics of the workflow, while file *SubmissionID-pe* and file *SubmissionID-pe* accommodate those of PEs and PEIs.

The data content is the same as what is stored to MySQL database system.

3.3.3 Memory

Memory usage files are stored to file in name of "SubmissionID.dat" to the path specified in PF configs. The default path should be in a directory called "memory".

Every tuple in memory usage file is made up of following columns, separated by tab stop:

1. keyword "MEM"
2. RSS memory usage in MiB
3. timestamp of the log
4. operation measured

5. ID of PE measured

6. ID of PEI measured

An example tuple extracted from a real-case memory usage file is shown in Listing 14.

```
MEM 3.628906    2015-08-18 05:37:15    read    SimpleProcessingPE5    1
```

Listing 14: A tuple from a real-case memory usage file

3.3.4 Graph

Graph file path can be set in configs for PF. Also, it can be specified in the configs whether PNG file will be stored along with DOT file.

In a DOT file, the graph of a workflow is stored in a way both human and machine can understand.

Since dispel4py workflow graph is directed, digraph keyword is used to begin the graph. PEIs are described within curly braces and an arrow (-¿) is used to show relationships between PEIs.

Also, labels of PEIs, which specify names of PEIs, are placed in square brackets ([]) after a statement and before the semicolon (which is optional). Besides the description of the PEIs and the connections, a couple of attributes which control the display of the graph on screen are stored in file as well.

Listing 15 is an example of the DOT file generated for a pipeline-like workflow, with 1 producer PEI and 5 consumer PEI.

```
digraph request
{
node [shape=Mrecord,            style=filled, fillcolor=white];
TestOneInOneOut30[label="{ {<in_input>input} | TestOneInOneOut3 | {<out_output>output} }"];
TestOneInOneOut11[label="{ {<in_input>input} | TestOneInOneOut1 | {<out_output>output} }"];
TestOneInOneOut52[label="{ {<in_input>input} | TestOneInOneOut5 }"];
TestOneInOneOut23[label="{ {<in_input>input} | TestOneInOneOut2 | {<out_output>output} }"];
TestOneInOneOut44[label="{ {<in_input>input} | TestOneInOneOut4 | {<out_output>output} }"];
TestProducer05[label="{ TestProducer0 | {<out_output>output} }"];
TestOneInOneOut30:out_output -> TestOneInOneOut44:in_input;
TestOneInOneOut11:out_output -> TestOneInOneOut23:in_input;
TestOneInOneOut23:out_output -> TestOneInOneOut30:in_input;
TestOneInOneOut44:out_output -> TestOneInOneOut52:in_input;
TestProducer05:out_output -> TestOneInOneOut11:in_input;
}
```

Listing 15: DOT file generated for a pipeline-like workflow

PNG format graph description file is generated based on the attributes placed in the corresponding DOT file. An example of the PNG file generated from the DOT example above is shown in Figure 3.3.

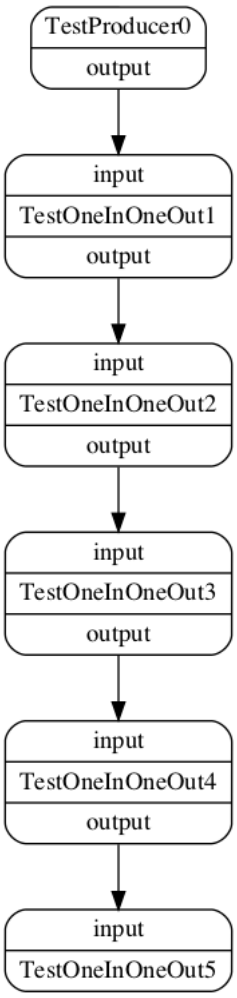

Figure 3.3: The graph of PipelineTest, a pipeline workflow with 1 producer and 5 consumers

Chapter 4

dispel4py Workflow Performance Data Collecting

In data collecting phase, two workflows were enacted with profiling mechanism. Performance data were captured by profiling framework (PF) and then stored to performance database (PDB). One of the workflow was *Pipeline Test* workflow, and the other was *Internal Extinction of Galaxies* workflow. Both workflows were shaped as pipeline. *Pipeline Test* was enacted on local server, while the *Internal Extinction of Galaxies* was run on the Eddie cluster with partitioned manner and non-partitioned manner respectively.

4.1 dispel4py Workflows

A. Pipeline Test
Pipeline Test is a pipeline workflow with 1 producer PE, *TestProducer*, and 5 consumer PEs, *TestOneInOneOut*. The graph of *Pipeline Test* is illustrated as Figure 3.3. It acts as a good testing pipeline workflow, due to its light computation workload in each PE.

The producer in the workflow, *TestProducer*, with no input and 1 output, produces an `int` object and writes it to its output `Queue`, in each iteration. The consumer, *TestOneInOneOut*, with 1 input and 1 output, transfers any item from input `Queue` to output `Queue`.

B. Internal Extinction of Galaxies (Astro)
As shown in Figure 4.1, *Internal Extinction of Galaxies*, also called *Astro*, is a pipeline workflow with 1 producer and 3 consumers. The first PE, *ReadRaDec* (or *read*), reads data from an external file, which contains right ascension (Ra) and declination (Dec) for 1051 galaxies. The second PE, *GetVoTable*, executes a lookup for galaxy, based on given Ra and Dec, with a specific astronomical network service, *Virtual Observatory*. The following PE, *FilterColumns*, filters the results applying to certain condition. The last PE, *InternalExtinction*, calculates internal extinction out out of the data parsed from *FilterColumns*.

4.2 Data Collecting on Local Server

4.2.1 Hardware Environment and Software Environment

Local server machine was a MacBook Pro, with 2.8 GHz intel Core i7 processor, and 16GB 1600 MHz DDR3 memory. *Pipeline test* workflow was enacted on local server to test the profiling mechanism.

MySQL Database Server was installed on local server. The version of the database server that we used was 5.6.25-enterprise-commercial-advanced MySQL Enterprise Server - Advanced Edition (Commercial). MySQLdb, a Python DB API-2.0-compliant interface, was imported in `monitor_workflow` to support the connection between the profiling framework (PF) and

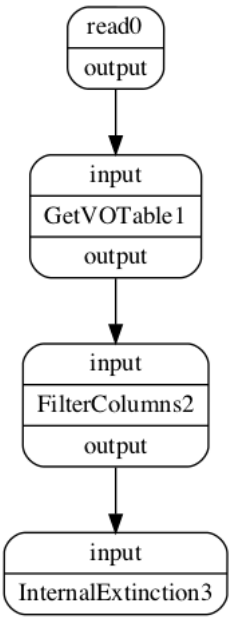

Figure 4.1: The graph of Internal Extinction of Galaxies (Astro), a pipeline workflow with 1 producer and 3 consumers

MySQL database. Therefore, relational performance data will be stored to MySQL database, with connection specified in PF config file (*config.json*).

4.2.2 Implementation

Pipeline Test workflow was initialized with 6 processes, and 5000 iterations. The command to enact the workflow was illustrated with Listing 16.

```
dispel4py multi dispel4py/examples/graph_testing/pipeline_test.py -n 6 -i 5000 -p
```

Listing 16: Command line to enact Pipeline Test

Here was a detailed explanation on how PF worked for a real-case workflow in four stages. PF observed performance data from runtime workflow in each PEI of *Pipeline Test*, and inserted them into `profiles`, which was a dictionary proxy shared between processes. The `profiles` was transferred back to the `Monitor` after the last PEI terminated. Then a method, `get_data_from_profiles`, was invoked to tidy up the raw data in `profiles`, since the raw data was indexed with process rank ID, which was hard to be understood by human. After data tidy-up, the decent-looking performance data container, `cleaned_profiles` would be logged on to screen before dumped to PDB. For this workflow case, the `cleaned_profiles` was recorded as Listing 17.

The `Monitor` then committed the third stage, deriving and estimating PE performance based on analysis on `cleaned_profiles`.

At the last stage, the `Monitor` flushed characteristics data (relational performance data and graph) of workflow, PE and PEI to PDB. The log of the last stage printed to standard output is shown as below. These two lines of information (Listing 18) indicate that MySQL connection was built between PF and PDB. Besides, the inserting queries for adding performance data were successful.

```
{'pipeline_test': {'f0e44f87460c11e5a155a0999b075e3f': {'submitted': '2015-08-19 01:55:09',
 ↪  'exec': 72.02648282051086}},
'TestProducer0': {'4': {'indatasize': 0, 'process': 0.023691415786743164, 'outdatasize':
 ↪  1400000, 'terminate': 2.002716064453125e-05, 'write': 10.652345418930054, 'read':
 ↪  61.17087650299072, 'outdatatype': 'int', 'readrate': 81.75459116979458, 'total':
 ↪  71.89328193664551, 'writerate': 469.3801978214683, 'indatatype': ''}},
'TestOneInOneOut5': {'3': {'indatasize': 1400000, 'process': 0.010790586471557617,
 ↪  'outdatasize': 1400000, 'terminate': 2.1457672119140625e-06, 'write':
 ↪  10.121912240982056, 'read': 61.827826499938965, 'outdatatype': 'int', 'readrate':
 ↪  80.88590984198574, 'total': 72.01523900032043, 'writerate': 493.9778058691098,
 ↪  'indatatype': 'int'}},
'TestOneInOneOut4': {'5': {'indatasize': 1400000, 'process': 0.0101318359375, 'outdatasize':
 ↪  1400000, 'terminate': 3.4809112548828125e-05, 'write': 10.565333366394043, 'read':
 ↪  61.38535523414612, 'outdatatype': 'int', 'readrate': 81.4689428923945, 'total':
 ↪  72.01250195503235, 'writerate': 473.24583395578213, 'indatatype': 'int'}},
'TestOneInOneOut3': {'1': {'indatasize': 1400000, 'process': 0.00999593734741211,
 ↪  'outdatasize': 1400000, 'terminate': 3.504753112792969e-05, 'write':
 ↪  10.548195123672485, 'read': 61.387120485305786, 'outdatatype': 'int', 'readrate':
 ↪  81.46660016732805, 'total': 71.99697089195251, 'writerate': 474.01474293729103,
 ↪  'indatatype': 'int'}},
'TestOneInOneOut2': {'2': {'indatasize': 1400000, 'process': 0.010299205780029297,
 ↪  'outdatasize': 1400000, 'terminate': 3.504753112792969e-05, 'write':
 ↪  10.492842674255371, 'read': 61.40846276283264, 'outdatatype': 'int', 'readrate':
 ↪  81.4382867604829, 'total': 71.96380686759949, 'writerate': 476.51529287365656,
 ↪  'indatatype': 'int'}},
'TestOneInOneOut1': {'0': {'indatasize': 1400000, 'process': 0.010103464126586914,
 ↪  'outdatasize': 1400000, 'terminate': 2.09808349609375e-05, 'write': 10.57215428352356,
 ↪  'read': 61.262603521347046, 'outdatatype': 'int', 'readrate': 81.63218199267999,
 ↪  'total': 71.89675498008728, 'writerate': 472.9405063443291, 'indatatype': 'int'}}}
```

Listing 17: cleaned_profiles variable of Pipeline Test workflow

```
Trying to store performance data to MySQL Database
Performance data stored to MySQL Database
```

Listing 18: Logs showing MySQL database storing

4.2.3 Results

Workflow submission ID, can be read from the log of PF, which was $f0e44f87460c11e5a155a0999b075e3f$ from the `cleaned_profiles` example above. This 32bit string was used as an index to query performance data of a certain workflow.

We looked up relational performance data records by SQL query script shown in Listing 19

```
1 SELECT * FROM WorkflowProfiles WHERE WorkflowProfiles.WF_SubmissionID =
   ↪  x'f0e44f87460c11e5a155a0999b075e3f';
2 SELECT * FROM PEProfiles WHERE PEProfiles.PE_SubmissionID =
   ↪  x'f0e44f87460c11e5a155a0999b075e3f';
3 SELECT * FROM PEInstanceProfiles WHERE PEInstanceProfiles.PEI_SubmissionID =
   ↪  x'f0e44f87460c11e5a155a0999b075e3f';
```

Listing 19: SQL query script to look up relational performance data records

Relevant tuples were returned from table *WorkflowProfiles* (Figure 4.2), table *PEProfiles* (Figure 4.3) and table *PEInstancePRofiles* (Figure 4.4).

Memory usage log file of this workflow was located at the path, stored in the field of `WF_MemoryProfile` in table *WorkflowProfiles*. The memory usage log file of *Pipeline Test* worklow had 60012 lines of data. As the `interval` of memory logging was 1^{-3} second, while the `timeout` was 1^{-2} second, each PEI was supposed to inquire RSS value for 1000 times in every second. That was the reason why the memory usage data was large. Also, the size of the file showed the necessaries to flush memory data to disk in bunches.

WF_SubmissionID	WF_Name	WF_Mapping	WF_Processor Num	WF_Iteration Num	WF_GraphDescription	WF_Total Time	WF_Submitted	WF_MemoryProfile
f0e44f87460c11e5a155a0999b075e3f	pipeline_test	multi	6	5000	/Users/anqilu/dispel4py/graphs/f0e44f87460c11e5a155a0999b075e3f	72.02648	2015-08-19 01:55:09	/Users/anqilu/dispel4py/memory/f0e44f87460c11e5a155a0999b075e3f.dat

Figure 4.2: Characteristics of workflow captured from Pipeline Test, stored as WorkflowProfiles in MySQL database

PE_PEID	PE_SubmissionID	PE_Total Time	PE_Read Time	PE_Write Time	PE_Process Time	PE_Terminate Time	PE_InDataSize	PE_InDataType	PE_OutDataSize	PE_OutDataType	PE_Read Rate	PE_Write Rate
TestOneInOneOut1	f0e44f87460c11e5a155a0999b075e3f	71.89675	61.26260	10.57215	0.01010	0.00002	1400000	int	1400000	int	81.63	472.94
TestOneInOneOut2	f0e44f87460c11e5a155a0999b075e3f	71.96381	61.40846	10.49284	0.01030	0.00004	1400000	int	1400000	int	81.44	476.52
TestOneInOneOut3	f0e44f87460c11e5a155a0999b075e3f	71.99697	61.38712	10.54820	0.01000	0.00004	1400000	int	1400000	int	81.47	474.01
TestOneInOneOut4	f0e44f87460c11e5a155a0999b075e3f	72.01250	61.38536	10.56533	0.01013	0.00003	1400000	int	1400000	int	81.47	473.25
TestOneInOneOut5	f0e44f87460c11e5a155a0999b075e3f	72.01524	61.82783	10.12191	0.01079	0.00000	1400000	int	1400000	int	80.89	493.98
TestProducer0	f0e44f87460c11e5a155a0999b075e3f	71.89328	61.17088	10.65235	0.02369	0.00002	0		1400000	int	81.75	469.38

Figure 4.3: Characteristics of PE captured from Pipeline Test, stored as PEProfiles in MySQL database

PEI_Rank	PEI_PEID	PEI_SubmissionID	PEI_Total Time	PEI_Read Time	PEI_Write Time	PEI_Process Time	PEI_Terminate Time	PEI_InDataSize	PEI_OutDataSize	PEI_Read Rate	PEI_Write Rate
0	TestOneInOneOut1	f0e44f87460c11e5a155a0999b075e3f	71.89675	61.2626	10.57215	0.0101	0.00002	1400000	1400000	81.63	472.94
1	TestOneInOneOut3	f0e44f87460c11e5a155a0999b075e3f	71.99697	61.38712	10.5482	0.01	0.00004	1400000	1400000	81.47	474.01
2	TestOneInOneOut2	f0e44f87460c11e5a155a0999b075e3f	71.96381	61.40846	10.49284	0.0103	0.00004	1400000	1400000	81.44	476.52
3	TestOneInOneOut5	f0e44f87460c11e5a155a0999b075e3f	72.01524	61.82783	10.12191	0.01079	0	1400000	1400000	80.89	493.98
4	TestProducer0	f0e44f87460c11e5a155a0999b075e3f	71.89328	61.17088	10.65235	0.02369	0.00002	0	1400000	81.75	469.38
5	TestOneInOneOut4	f0e44f87460c11e5a155a0999b075e3f	72.0125	61.38536	10.56533	0.01013	0.00003	1400000	1400000	81.47	473.25

Figure 4.4: Characteristics of PEI captured from Pipeline Test, stored as PEInstanceProfiles in MySQL database

First 15 rows of the memory log of *Pipeline Test* are presented in Listing 20.

From the brief of the memory log, the increase trend in the memory taken by each PEIs is shown. Future data analysis will be taken on the file to extract more information through the space efficiency of *Pipeline Test* workflow.

The graph description, and PNG file, Figure 3.3, generated from the *Pipeline Test* workflow have been displayed earlier.

4.3 Data Collecting on Eddie Cluster

4.3.1 Hardware Environment and Software Environment

The Edinburgh Compute and Data Facility (ECDF) provides large scale storage and high performance computing services to the University of Edinburgh and associated institutions. Eddie is a high-performance cluster of computers provided by ECDF. Currently, Eddie comprises two main parts. One is Mark2Phase1 - 130 IBM dx360M3 iDataPlex servers, each with two Intel Xeon E5620 quad-core processors. All these nodes are connected by a GigabitEthernet network with a 10 Gigabit network core. The other is Mark2Phase2 - 156 IBM dx360M3 iDataPlex

```
MEM  3.937500  2015-08-19 01:55:09  process TestOneInOneOut3  1
MEM  3.960938  2015-08-19 01:55:09  process TestOneInOneOut1  0
MEM  4.035156  2015-08-19 01:55:10  process TestOneInOneOut2  2
MEM  3.972656  2015-08-19 01:55:10  process TestOneInOneOut5  3
MEM  3.871094  2015-08-19 01:55:10  process TestProducer0     4
MEM  4.000000  2015-08-19 01:55:10  process TestOneInOneOut4  5
MEM  4.164062  2015-08-19 01:55:10  read    TestOneInOneOut1  0
MEM  4.160156  2015-08-19 01:55:10  read    TestOneInOneOut3  1
MEM  4.234375  2015-08-19 01:55:10  read    TestOneInOneOut2  2
MEM  4.203125  2015-08-19 01:55:10  read    TestOneInOneOut5  3
MEM  4.191406  2015-08-19 01:55:10  read    TestOneInOneOut4  5
MEM  4.070312  2015-08-19 01:55:10  read    TestProducer0     4
MEM  5.140625  2015-08-19 01:55:10  write   TestProducer0     4
MEM  5.152344  2015-08-19 01:55:10  write   TestOneInOneOut1  0
MEM  5.230469  2015-08-19 01:55:10  write   TestOneInOneOut2  2
```

Listing 20: Memory log of Pipeline Test (first 15 lines)

servers, each with two Intel Xeon E5645 six-core processors. All these nodes are connected by Gigabit Ethernet network. In terms of file system, Eddie is connected to a large amount of high performance disk storage via a number of dedicated machines.

Access to Eddie is via a scheduler and batch system. The batch system is responsible for accepting large numbers of jobs and queueing them. The scheduler decides in which order the jobs will be run. Furthermore, we use file systems to store all three parts (relational performance data, graph and memory log) of PDB, instead of relational database systems.

4.3.2 Implementation

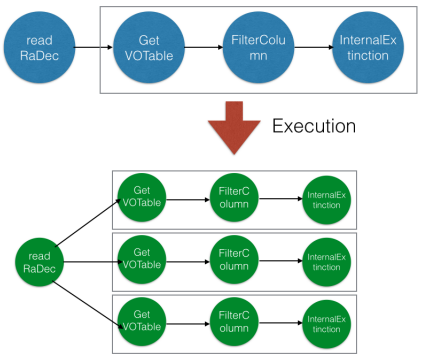

Figure 4.5: Enact Astro workflow in a partitioned manner (AstroPartition)

We implemented our profiling framework with *Astro* on Eddie cluster. To give a better reference for future enactment decision, we executed it both with common iterative manner (*AstroIteration*) and partitioned manner (*AstroPartition*) . In the former mode, each PE was instantiated to a process. In the latter mode, as illustrated at Figure 4.5, *GetVoTable*, *FilterColumns* and *InternalExtinction* were wrapped into a sub-workflow, which was treated as a *composite PE*, with internal data stream connection relations remained. This means, each instance of this *composite PE* ran on the same process. In contrast, as shown in Figure 4.6, each original PE works on an independent process.

To make sure the rate of memory inquiry was reasonable, we configured the frequency of memory capturing in *config.json* based on experience. Specifically, we ran the *Astro* workflow for several times with different memory capturing rate, and decided a reasonable `interval` and `timeout` value. A good setting of memory tracing frequency should not put the process into too much stress, which could happen if the access frequency is too high. Meantime, the

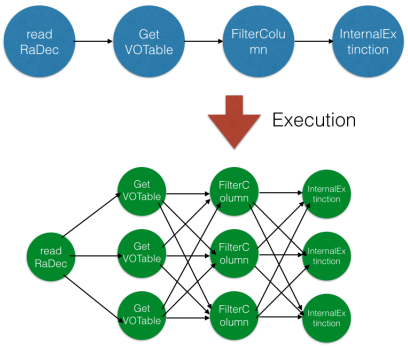

Figure 4.6: Enact Astro workflow in a common iterative manner (AstroIteration)

frequency should not to be so low as changes in RSS might be missed. By now, this setting is made manually. But it will be completed by the profiling framework itself in future.

The whole configuration files is attached in Listing 21.

```
{
    "graph_profile_store": "graphs/",
    "workflow_profile_store": "performance_data/",
    "memory_profile_store": "memory/",
    "store_png": false,
    "interval": {"read":0.0005, "write":0.0004, "process":0.004},
    "timeout": {"read":0.005, "write":0.004, "process":0.04}
}
```

Listing 21: Configuration file for profiling Astro

To run *Astro* on Eddie, we described the execution commands in bash script, and then submitted the script as a task to Eddie batch system.

Listing 22 presents the bash script to execute the workflow in a partitioned manner, through argument `-s` specified.

```
#!/bin/sh
#£ -N astro
#£ -cwd
#£ -pe OpenMP 4
#£ -l h_rt=00:30:00

. /etc/profile.d/modules.sh
module load anaconda/2.1.0

python -m dispel4py.new.processor multi Astro/int_ext_graph.py -d '{"read"
↪  : [ {"input" : "Astro/coordinates.txt"} ]}' -n 4 -s -p
```

Listing 22: Bash script to execute AstroPartition

Listing 23 shows the bash script to enact the workflow in the common iterative mode, without adding argument `-s`.

Then we submitted both script to Eddie cluster task scheduler.

```
#!/bin/sh
#£ -N astro
#£ -cwd
#£ -pe OpenMP 4
#£ -l h_rt=00:30:00

. /etc/profile.d/modules.sh
module load anaconda/2.1.0

python -m dispel4py.new.processor multi Astro/int_ext_graph.py -d '{"read"
 ↪  : [ {"input" : "Astro/coordinates.txt"} ]}' -n 4 -p
```

Listing 23: Bash script to execute AstroIteration

4.3.3 Results

The task of *AstroPartition* was given 4235977 as its task ID, while the one of *AstroItertation* was given 4249268.

dispel4py system runtime events of both workflow instances were logged to specific output files. The The head of the log file shows the process allocation of this workflow, while the tail of the log file gives information on the performance data container, `cleaned_profiles`. The latter variable, `cleaned_profiles`, is a Python dictionary used by profiling framework to manage data before transferring it to performance database. It shows detailed information (e.g. workflow submission ID, workflow execution parameters, etc.) which can both be read by machine and human.

The heads of runtime logs of both workflow instances are attached below. Listing 24 shows first 10 lines of events log of *AstroPartition*, while Listing 25 shows that of *AstroIteration*.

The `Submission ID` of each workflow instances, $15278efe46ae11e5af97e41f13efbb4a$ for *AstroPartition* and $90df2138478f11e58583e41f13f00272$ for *AstroIteration*, are displayed at the first line.

For *AstroPartition* (Listing 24), the partition of the workflow graph can be seen from the second line. Apparently, *GetVOTable* PE, *FilterColumn* PE, and *InternalExtinction* were wrapped as a composite PE, named as *SimpleProcessingPE5*. Meanwhile, *read* PE was wrapped by *SimpleProcessingPE4*. The fifth line demonstrates the map from wrapper PEs to the processes which they ran on. *read* PE ran on rank 3 while the other three ran as a composite PE on rank 0, rank 1 and rank 2. In another word, *SimpleProcessingPE5*, the wrapper of the three, had three instances at runtime.

For *AstroIteration* (Listing 25), each PE was allocated with one process. *read* PE ran on rank 1, GetVOTable PE ran on rank 2, *FilterColumn* PE ran on rank 3, and *InternalExtinction* PE ran on rank 0.

```
15278efe46ae11e5af97e41f13efbb4a
Partitions: [read0], [GetVOTable1, FilterColumns2, InternalExtinction3]
SimpleProcessingPE5 contains ['FilterColumns2', 'GetVOTable1', 'InternalExtinction3']
SimpleProcessingPE4 contains ['read0']
Processes: {'SimpleProcessingPE5': [0, 1, 2], 'SimpleProcessingPE4': [3]}
read0 (rank 3): Reading file Astro/coordinates.txt
SimpleProcessingPE4 (rank 3): Processed 1 iteration.
GetVOTable1 (rank 0): reading VOTable RA=0.77345, DEC=-1.9143
FilterColumns2 (rank 0): extracted column: MType = SBbc
FilterColumns2 (rank 0): extracted column: logR25 = 0.35
```

Listing 24: Log of AstroPartition (first 10 lines)

The tails of the logs, from which PEIs characteristics can be observed, are illustrated at Listing 26 for *AstroPartition* and Listing 27 for *AstroIteration*.

```
90df2138478f11e58583e41f13f00272
Processes: {'read0': [1], 'FilterColumns2': [3], 'InternalExtinction3': [0], 'GetVOTable1':
 ↪  [2]}
GetVOTable1 (rank 2): reading VOTable RA=0.77345, DEC=-1.9143
GetVOTable1 (rank 2): reading VOTable RA=0.83487, DEC=29.7972
GetVOTable1 (rank 2): reading VOTable RA=0.84108, DEC=30.7821
FilterColumns2 (rank 3): extracted column: MType = SBbc
FilterColumns2 (rank 3): extracted column: logR25 = 0.35
InternalExtinction3 (rank 0): internal extinction: [1, '0.77345', '-1.9143', 'SBbc',
 ↪  0.34999999, 4.0, 0.5533987734977625]
InternalExtinction3 (rank 0): internal extinction: [1, '0.77345', '-1.9143', 'SBbc',
 ↪  0.34999999, 4.0, 0.5533987734977625]
GetVOTable1 (rank 2): reading VOTable RA=0.99487, DEC=20.7524
```

Listing 25: Log of AstroIteration (first 10 lines)

```
InternalExtinction3 (rank 2): internal extinction: [1047, '359.19825', '1.3550', 'Sbc',
 ↪  0.68000001, 4.0, 1.0176387280184689]
GetVOTable1 (rank 2): reading VOTable RA=359.63387, DEC=26.2148
FilterColumns2 (rank 2): extracted column: MType = SBm
FilterColumns2 (rank 2): extracted column: logR25 = 0.05
InternalExtinction3 (rank 2): internal extinction: [1050, '359.63387', '26.2148', 'SBm',
 ↪  0.050000001, 9.0, 0.09255426954753808]
SimpleProcessingPE5 (rank 2): Processed 350 iterations.
{'int_ext_graph': {'15278efe46ae11e5af97e41f13efbb4a': {'submitted': '2015-08-19 21:08:39',
 ↪  'exec': 96.23444318771362}}, 'SimpleProcessingPE5': {'1': {'indatasize': 98000,
 ↪  'process': 88.1598846912384, 'outdatasize': 0, 'terminate': 9.5367431640625e-07,
 ↪  'write': 0, 'read': 2.6891915798187256, 'outdatatype': '', 'readrate':
 ↪  130.5224970337221, 'total': 90.8555109500885, 'writerate': None, 'indatatype': 'list'},
 ↪  '0': {'indatasize': 98280, 'process': 84.27067589759827, 'outdatasize': 0, 'terminate':
 ↪  9.5367431640625e-07, 'write': 0, 'read': 2.6909921169281006, 'outdatatype': '',
 ↪  'readrate': 130.80677486406955, 'total': 86.96833109855652, 'writerate': None,
 ↪  'indatatype': 'list'}, '2': {'indatasize': 98000, 'process': 93.43654346466064,
 ↪  'outdatasize': 0, 'terminate': 1.1920928955078125e-06, 'write': 0, 'read':
 ↪  2.696721076965332, 'outdatatype': '', 'readrate': 130.1580660299457, 'total':
 ↪  96.14008617401123, 'writerate': None, 'indatatype': 'list'}}, 'SimpleProcessingPE4':
 ↪  {'3': {'indatasize': 280, 'process': 7.08356499671936, 'outdatasize': 0, 'terminate':
 ↪  4.9114227294921875e-05, 'write': 0, 'read': 0.015041828155517578, 'outdatatype': '',
 ↪  'readrate': 132.96256142019337, 'total': 7.100219964981079, 'writerate': None,
 ↪  'indatatype': 'list'}}}
Trying to write performance data to csv file
Workflow benchmark written to disk
Performance data written to csv files
```

Listing 26: Log of AstroPartition (last 10 lines)

From the tails, we can see the performance data dictionary flushed to PDB. Also, last three records show that the performance data were recorded to file system as CSV files successfully.

Relational performance data was stored in three CSV files: *SUBMISSION_ID-wf.csv*, *SUBMISSION_ID-pe.csv* and *SUBMISSION_ID-pei.csv*. The detailed contents are listed from Listing 28 to Listing 33.

From List 28 and List 31, it can be observed that, the execution time of *AstroPartition* was about 96 seconds, while it took *AtsroIteration* about 323 seconds. In terms of time cost, *AstroPartition* overtook *AstroIteration* in our experiment. Further conclusions will be drawn after data analysis on PDB in Data Analysis Chapter.

Its memory log can be found as *SUBMISSION_ID.dat*, which contains 4122 lines of records for *AstroPartition* and 10360 lines for *AstroIteration*.

The heads and the tails of the memory log are demonstrated below to give a better clue of the change of RSS taken by runtime PEIs.

As displayed in Listing 34 and Listing 35, RSS taken by AstroPartition started from around 30 MiB and remained at around 50 MiB in the end of the execution. This could be caused by the partitioning in a network requiring extra space to maintain and manage its structure.

```
FilterColumns2 (rank 3): Processed 1051 iterations.
InternalExtinction3 (rank 0): internal extinction: [1051, '359.86958', '21.4146', 'Sc',
 ↪  0.090000004, 6.0, 0.16005368379989493]
InternalExtinction3 (rank 0): internal extinction: [1051, '359.86958', '21.4146', 'Sc',
 ↪  0.090000004, 6.0, 0.16005368379989493]
InternalExtinction3 (rank 0): internal extinction: [1051, '359.86958', '21.4146', 'Sc',
 ↪  0.090000004, 6.0, 0.16005368379989493]
InternalExtinction3 (rank 0): internal extinction: [1051, '359.86958', '21.4146', 'Sc',
 ↪  0.090000004, 6.0, 0.16005368379989493]
InternalExtinction3 (rank 0): Processed 1028 iterations.
{'read0': {'1': {'indatasize': 280, 'process': 7.1000189781188965, 'outdatasize': 0,
 ↪  'terminate': 1.4066696166992188e-05, 'write': 0, 'read': 0.01594710350036621,
 ↪  'outdatatype': '', 'readrate': 125.41462466548059, 'total': 7.117380857467651,
 ↪  'writerate': None, 'indatatype': 'unicode'}}, 'int_ext_graph':
 ↪  {'90df2138478f11e58583e41f13f00272': {'submitted': '2015-08-21 00:02:44', 'exec':
 ↪  322.94744396209717}}, 'FilterColumns2': {'3': {'indatasize': 294280, 'process':
 ↪  36.39006185531616, 'outdatasize': 294280, 'terminate': 1.7881393432617188e-05, 'write':
 ↪  7.080553293228149, 'read': 279.3691680431366, 'outdatatype': 'list', 'readrate':
 ↪  3.76562670594188, 'total': 322.87789487838745, 'writerate': 148.43472769355154,
 ↪  'indatatype': 'list'}}, 'InternalExtinction3': {'0': {'indatasize': 287840, 'process':
 ↪  32.57888460159302, 'outdatasize': 263480, 'terminate': 1.9073486328125e-06, 'write':
 ↪  6.301403522491455, 'read': 284.0029013156891, 'outdatatype': 'list', 'readrate':
 ↪  3.6232024223449555, 'total': 322.92328000068665, 'writerate': 149.3318110229428,
 ↪  'indatatype': 'list'}}, 'GetVOTable1': {'2': {'indatasize': 294280, 'process':
 ↪  306.80878615379333, 'outdatasize': 294280, 'terminate': 2.002716064453125e-05, 'write':
 ↪  7.018192768096924, 'read': 8.993653297424316, 'outdatatype': 'list', 'readrate':
 ↪  116.97137583692283, 'total': 322.85016107559204, 'writerate': 149.75365236155983,
 ↪  'indatatype': 'list'}}}
Trying to write performance data to csv file
Workflow benchmark written to disk
Performance data written to csv files
```

Listing 27: Log of AstroIteration (first 10 lines)

```
WF_SubmissionID, WF_Name, WF_Mapping, WF_ProcessorNum, WF_IterationNum, WF_GraphDescription,
 ↪  WF_MemoryProfile, WF_TotalTime, WF_Submitted
15278efe46ae11e5af97e41f13efbb4a, int_ext_graph, multi, 4, 1,
 ↪  /exports/work/inf_dir/anqilu/dispel4py/graphs/ 15278efe46ae11e5af97e41f13efbb4a,
 ↪  /exports/work/inf_dir/anqilu/dispel4py/memory/ 15278efe46ae11e5af97e41f13efbb4a.dat,
 ↪  96.23444318771362, 2015-08-19 21:08:39
```

Listing 28: Relational performance data in PDB: The AstroPartition runtime workflow characteristics

Compared with the memory footprints of *AstroPartition*, from Listing 36 and Listing 37, we can see that, memory space occupied by runtime PEIs were usually released by the end of the enactment.

As illustrated in Listing 38, the DOT language description of the *AstroPartition* and *AstroIteration*, stored to *SUBMISSION_ID.dot*, is presented as below. As they shared the same graph, description for each workflow instances is the same.

```
PE_PEID, PE_SubmissionID, PE_TotalTime, PE_ReadTime, PE_WriteTime, PE_ProcessTime,
 ↪  PE_TerminateTime, PE_InDataSize, PE_OutDataSize, PE_InDataType, PE_OutDataType,
 ↪  PE_ReadRate, PE_WriteRate
SimpleProcessingPE5, 15278efe46ae11e5af97e41f13efbb4a, 91.32130940755208, 2.692301591237386,
 ↪  0, 88.62236801783244, 1.0331471761067708e-06, 294280, 0, list, , 130.49577930924576, 0
SimpleProcessingPE4, 15278efe46ae11e5af97e41f13efbb4a, 7.100219964981079,
 ↪  0.015041828155517578, 0, 7.08356499671936, 4.9114227294921875e-05, 280, 0, list, ,
 ↪  132.96256142019337, 0
```

Listing 29: Relational performance data in PDB: The AstroPartition runtime wrapper PEs characteristics

```
PEI_Rank, PEI_PEID, PEI_SubmissionID, PEI_TotalTime, PEI_ReadTime, PEI_WriteTime,
 ↪  PEI_ProcessTime, PEI_TerminateTime, PEI_InDataSize, PEI_OutDataSize, PEI_ReadRate,
 ↪  PEI_WriteRate
1, SimpleProcessingPE5, 15278efe46ae11e5af97e41f13efbb4a, 90.8555109500885,
 ↪  2.6891915798187256, 0, 88.1598846912384, 9.5367431640625e-07, 98000, 0,
 ↪  130.5224970337221,
0, SimpleProcessingPE5, 15278efe46ae11e5af97e41f13efbb4a, 86.96833109855652,
 ↪  2.6909921169281006, 0, 84.27067589759827, 9.5367431640625e-07, 98280, 0,
 ↪  130.80677486406955,
2, SimpleProcessingPE5, 15278efe46ae11e5af97e41f13efbb4a, 96.14008617401123,
 ↪  2.696721076965332, 0, 93.43654346466064, 1.1920928955078125e-06, 98000, 0,
 ↪  130.1580660299457,
3, SimpleProcessingPE4, 15278efe46ae11e5af97e41f13efbb4a, 7.100219964981079,
 ↪  0.015041828155517578, 0, 7.08356499671936, 4.9114227294921875e-05, 280, 0,
 ↪  132.96256142019337,
```

Listing 30: Relational performance data in PDB: The AstroPartition runtime wrapper PEIs characteristics

```
WF_SubmissionID, WF_Name, WF_Mapping, WF_ProcessorNum, WF_IterationNum, WF_GraphDescription,
 ↪  WF_MemoryProfile, WF_TotalTime, WF_Submitted
90df2138478f11e58583e41f13f00272, int_ext_graph, multi, 4, 1,
 ↪  /exports/work/inf_dir/anqilu/dispel4py/graphs/90df2138478f11e58583e41f13f00272,
 ↪  /exports/work/inf_dir/anqilu/dispel4py/memory/90df2138478f11e58583e41f13f00272.dat,
 ↪  322.94744396209717, 2015-08-21 00:02:44
```

Listing 31: Relational performance data in PDB: The AstroIteration runtime workflow characteristics

```
PE_PEID, PE_SubmissionID, PE_TotalTime, PE_ReadTime, PE_WriteTime, PE_ProcessTime,
 ↪  PE_TerminateTime, PE_InDataSize, PE_OutDataSize, PE_InDataType, PE_OutDataType,
 ↪  PE_ReadRate, PE_WriteRate
read0, 90df2138478f11e58583e41f13f00272, 7.117380857467651, 0.01594710350036621, 0,
 ↪  7.1000189781188965, 1.4066696166992188e-05, 280, 0, unicode, , 125.41462466548059, 0
FilterColumns2, 90df2138478f11e58583e41f13f00272, 322.87789487838745, 279.3691680431366,
 ↪  7.080553293228149, 36.39006185531616, 1.7881393432617188e-05, 294280, 294280, list,
 ↪  list, 3.76562670594188, 148.43472769355154
InternalExtinction3, 90df2138478f11e58583e41f13f00272, 322.92328000068665,
 ↪  284.0029013156891, 6.301403522491455, 32.57888460159302, 1.9073486328125e-06, 287840,
 ↪  263480, list, list, 3.6232024223449555, 149.3318110229428
GetVOTable1, 90df2138478f11e58583e41f13f00272, 322.85016107559204, 8.993653297424316,
 ↪  7.018192768096924, 306.80878615379333, 2.002716064453125e-05, 294280, 294280, list,
 ↪  list, 116.97137583692283, 149.75365236155983
```

Listing 32: Relational performance data in PDB: The AstroIteration runtime wrapper PEs characteristics

```
PEI_Rank, PEI_PEID, PEI_SubmissionID, PEI_TotalTime, PEI_ReadTime, PEI_WriteTime,
 ↪  PEI_ProcessTime, PEI_TerminateTime, PEI_InDataSize, PEI_OutDataSize, PEI_ReadRate,
 ↪  PEI_WriteRate
1, read0, 90df2138478f11e58583e41f13f00272, 7.117380857467651, 0.01594710350036621, 0,
 ↪  7.1000189781188965, 1.4066696166992188e-05, 280, 0, 125.41462466548059,
3, FilterColumns2, 90df2138478f11e58583e41f13f00272, 322.87789487838745, 279.3691680431366,
 ↪  7.080553293228149, 36.39006185531616, 1.7881393432617188e-05, 294280, 294280,
 ↪  3.76562670594188, 148.43472769355154
0, InternalExtinction3, 90df2138478f11e58583e41f13f00272, 322.92328000068665,
 ↪  284.0029013156891, 6.301403522491455, 32.57888460159302, 1.9073486328125e-06, 287840,
 ↪  263480, 3.6232024223449555, 149.3318110229428
2, GetVOTable1, 90df2138478f11e58583e41f13f00272, 322.85016107559204, 8.993653297424316,
 ↪  7.018192768096924, 306.80878615379333, 2.002716064453125e-05, 294280, 294280,
 ↪  116.97137583692283, 149.75365236155983
```

Listing 33: Relational performance data in PDB: The AstroIteration runtime wrapper PEIs characteristics

```
MEM 32.449219  2015-08-19 21:08:39  read   SimpleProcessingPE5  0
MEM 32.449219  2015-08-19 21:08:39  read   SimpleProcessingPE4  3
MEM 32.449219  2015-08-19 21:08:39  read   SimpleProcessingPE5  2
MEM 32.449219  2015-08-19 21:08:39  read   SimpleProcessingPE5  1
MEM 32.652344  2015-08-19 21:08:39  write  SimpleProcessingPE4  3
MEM 32.785156  2015-08-19 21:08:39  write  SimpleProcessingPE4  3
MEM 34.816406  2015-08-19 21:08:39  write  SimpleProcessingPE4  3
MEM 36.835938  2015-08-19 21:08:39  write  SimpleProcessingPE4  3
MEM 36.835938  2015-08-19 21:08:39  write  SimpleProcessingPE4  3
MEM 36.835938  2015-08-19 21:08:39  write  SimpleProcessingPE4  3
```

Listing 34: Memory log of AstroPartition (first 10 lines)

```
MEM  50.128906  2015-08-19 21:10:10  read    SimpleProcessingPE5  0
MEM  50.113281  2015-08-19 21:10:10  write   SimpleProcessingPE5  1
MEM  50.113281  2015-08-19 21:10:10  process SimpleProcessingPE5  1
MEM  50.113281  2015-08-19 21:10:10  read    SimpleProcessingPE5  1
MEM  50.128906  2015-08-19 21:10:10  write   SimpleProcessingPE5  0
MEM  50.128906  2015-08-19 21:10:10  process SimpleProcessingPE5  0
MEM  50.128906  2015-08-19 21:10:10  read    SimpleProcessingPE5  0
MEM  50.128906  2015-08-19 21:10:11  write   SimpleProcessingPE5  0
MEM  50.128906  2015-08-19 21:10:11  process SimpleProcessingPE5  0
MEM  50.128906  2015-08-19 21:10:11  read    SimpleProcessingPE5  0
```

Listing 35: Memory log of AstroPartition (last 10 lines)

```
MEM  32.218750  2015-08-21 00:02:44 read    InternalExtinction3 0
MEM  32.218750  2015-08-21 00:02:44 read    FilterColumns2      3
MEM  32.218750  2015-08-21 00:02:44 read    GetVOTable1         2
MEM  32.218750  2015-08-21 00:02:44 read    read0               1
MEM  32.367188  2015-08-21 00:02:44 write   read0               1
MEM  32.550781  2015-08-21 00:02:44 write   read0               1
MEM  32.550781  2015-08-21 00:02:44 write   read0               1
MEM  32.550781  2015-08-21 00:02:44 write   read0               1
MEM  32.550781  2015-08-21 00:02:44 write   read0               1
MEM  32.550781  2015-08-21 00:02:44 write   read0               1
```

Listing 36: Memory log of AstroIteration (first 10 lines)

```
MEM  32.867188  2015-08-21 00:08:06  read    InternalExtinction3 0
MEM  34.257812  2015-08-21 00:08:07  process GetVOTable1         2
MEM  34.257812  2015-08-21 00:08:07  write   GetVOTable1         2
MEM  34.257812  2015-08-21 00:08:07  read    GetVOTable1         2
MEM  53.320312  2015-08-21 00:08:07  process FilterColumns2      3
MEM  53.320312  2015-08-21 00:08:07  write   FilterColumns2      3
MEM  53.320312  2015-08-21 00:08:07  read    FilterColumns2      3
MEM  32.867188  2015-08-21 00:08:07  process InternalExtinction3 0
MEM  32.867188  2015-08-21 00:08:07  write   InternalExtinction3 0
MEM  32.867188  2015-08-21 00:08:07  read    InternalExtinction3 0
```

Listing 37: Memory log of AstroIteration (last 10 lines)

```
digraph request
{
node [shape=Mrecord,            style=filled, fillcolor=white];
GetVOTable10[label="{ {<in_input>input} | GetVOTable1 | {<out_output>output} }"];
FilterColumns21[label="{ {<in_input>input} | FilterColumns2 | {<out_output>output} }"];
read02[label="{ read0 | {<out_output>output} }"];
InternalExtinction33[label="{ {<in_input>input} | InternalExtinction3 }"];
GetVOTable10:out_output -> FilterColumns21:in_input;
FilterColumns21:out_output -> InternalExtinction33:in_input;
read02:out_output -> GetVOTable10:in_input;
}
```

Listing 38: DOT language description of AstroPartition and AstroIteration

Chapter 5

Data Analysis

In this chapter, we will analyze the characteristics captured from workflow instances from 2 basic aspects: timings and memory. Performance database (PDB) of the *Astro* workflow, enacted on Eddie cluster, is used for study. Since the data streamed in a quite different way, as shown in Figure **??** and Figure 4.5, we do not compare them with data stream properties.

5.1 Timings Analysis

With timings analysis on PDB, we aim to figure out the timing performance of a workflow instance in operation level. Also, execution time of two workflows will be compared, as it is a vital key to estimate workflow efficiency.

Main operations performed by a PE are read, process, write and terminate. We are trying to figure out time cost on each operation for each PEI.

As shown in Figure 5.1, for *AstroPartition*, *SimpleProcessingPE5* cost more time than *SimpleProcessingPE4* on average (PEI 0: SimpleProcessingPE5 running on rank 0, PEI 1: SimpleProcessingPE5 running on rank 1, PEI 2: SimpleProcessingPE5 running on rank 2, PEI 3: SimpleProcessingPE4 running on rank 3). Both partitions spent most time on process operation, among four basic operations.

However, since the graph was partitioned to two sub-graph, dispel4py treated each partition as a single PE, thus complexity of underlying process was hidden. Detailed internal enactment status could not be captured due to this reason. From interception on the workflow, it was not possible to access benchmarks of original PEs under wrappers.

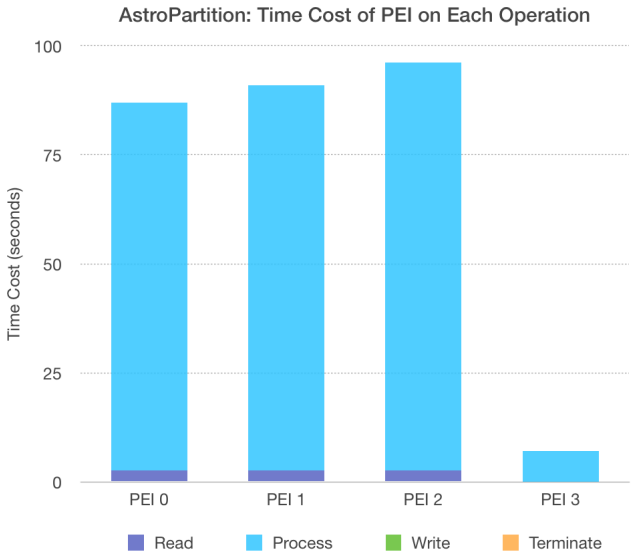

Figure 5.1: Time cost of each four operations of four PEIs in AstroPartition

Figure 5.2 displays the time cost of PEIs for *AstroIteration* (PEI 0: InternalExtinction3 running on rank 0, PEI 1: read0 running on rank 1, PEI 2: GetVOTable1 running on rank 2, PEI 3: FilterColumns2 running on rank 3).

Compared to AstroPartition, time cost of internal operations of each PE is clearer and better shown in this case.

InternalExtinction PE and *FilterColumns* PE spent the most of time on reading inputs. The second time costing operation for these two PEs was processing. Writing operation took them the least of time. In contrast, *GetVOTable* PE spent most time on processing input data, while it used much less time on writing and reading. As the producer in the pipeline, *read* PE read original data from an external source, and passed the data to next PE after processing on it. The bar chart shows that processing actually took it much more time than reading and writing. Though, compared to others, the processing time cost of *read* PE was the least among all PEs. Terminating time for each PE was so short which could be neglected.

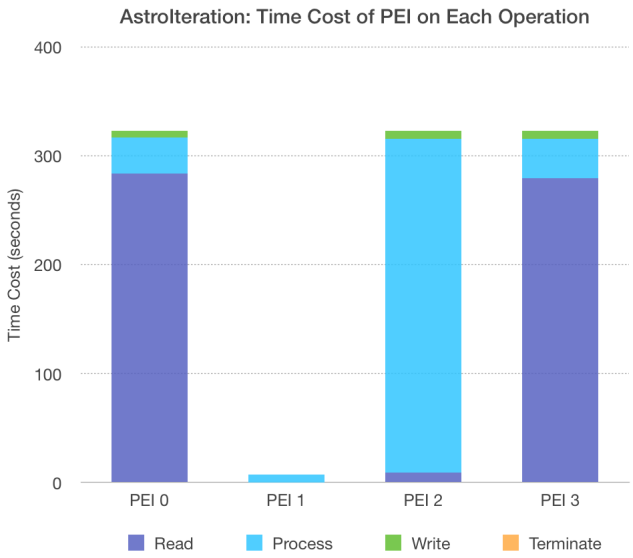

Figure 5.2: Time cost of each four operations of four PEIs in AstroIteration

In Figure 5.3, we compare timing parameters between *AstroPartition* and *AstroIteration* to see how differently pipeline workflows perform for partitioned graph and original graph. It can be assumed that, in Multi-Processing mapping, with the same computational resource allocated, the partitioned pipeline workflow performed better than non-partitioned one in consideration of execution time. This conclusion will help users to make decisions in future work.

	AstroPartition	AstroIteration
Execution Time	96.23444318771362	322.94744396209717
The Most Time-Consuming PEI	SimpleProcessingPE5 on rank 1	InternalExtinction3 on rank 0
The Least Time-Consuming PEI	SimpleProcessingPE4 on rank 3	read0 on rank 1

Figure 5.3: Comparison between AstroPartition ans AstroIteration

5.2 Memory Analysis

From the memory footprints tracked by profiling framework, the trend of memory usage of each PEI can be drawn out.

For *AstroPartition*, the memory performance of *SimpleProcessingPE5* partition is illustrated in Figure 5.4, Figure 5.5 and Figure 5.6. Each line chart shows memory cost, along time trend,

on all three operations (i.e. process, read and write) of a specific PEI. Also, Figure 5.7 presents the RSS changes of *SimpleProcessingPE4*, which only had one instance in runtime. In all of the four charts, there is a dramatic increase in the beginning of time, and then the memory cost keeps steady at a specific level.

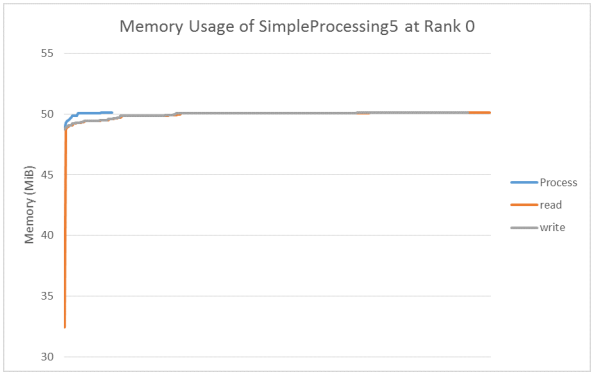

Figure 5.4: During enactment of AstroPartition, the memory usage trend of SimpleProcessing5 at rank 0 process

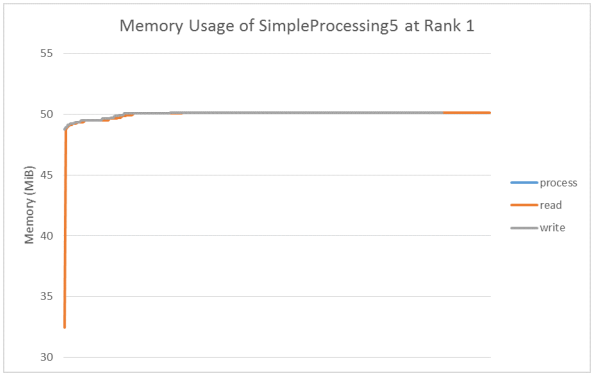

Figure 5.5: During enactment of AstroPartition, the memory usage trend of SimpleProcessing5 at rank 1 process

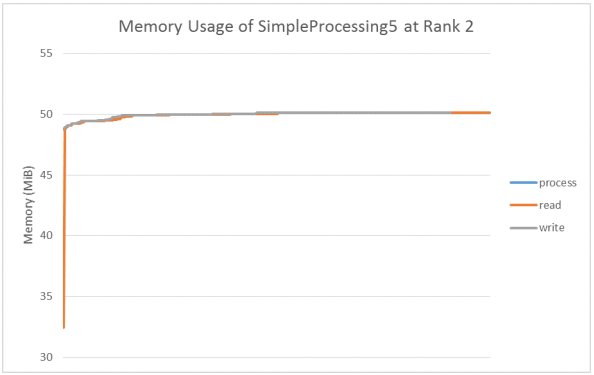

Figure 5.6: During enactment of AstroPartition, the memory usage trend of SimpleProcessing5 at rank 2 process

Since the ordinates of the line charts shown above are drawn with relatively large interval, from 30 MiB to 55 MiB, on axis, the memory change during a small period of time is not significant. This is because the memory usage of each PEI was relatively stable during the

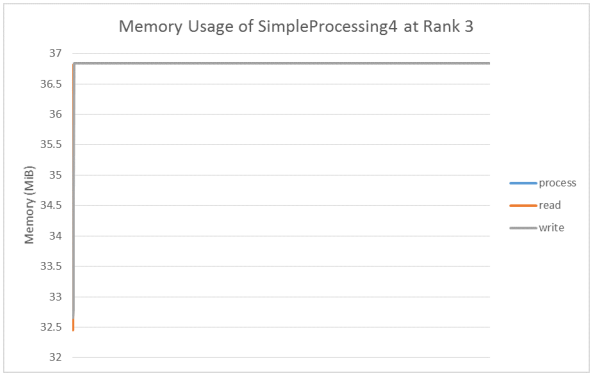

Figure 5.7: During enactment of AstroPartition, the memory usage trend of SimpleProcessing4 at rank 3 process

workflow execution. Thus we shrink the interval of y-axis to 1 MiB (shown in Figure 5.8) and 0.02 MiB (shown in Figure 5.9) respectively. The fluctuation of the memory cost in each operation is more clear.

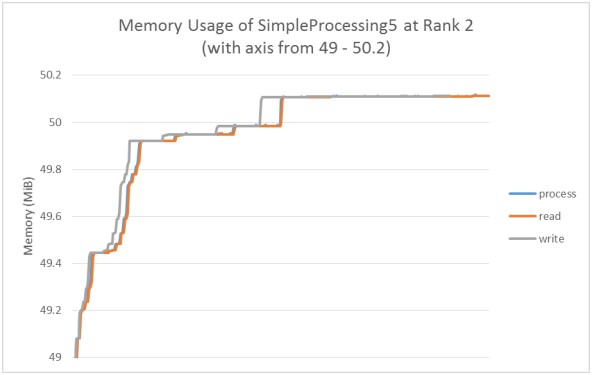

Figure 5.8: During enactment of AstroPartition, the memory usage trend of SimpleProcessing5 at rank 2 process, from 49 - 50.2 MiB

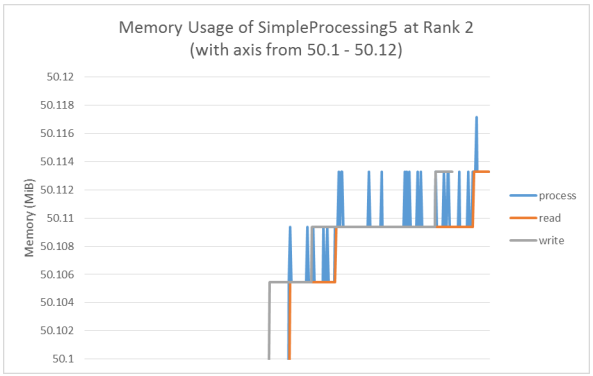

Figure 5.9: During enactment of AstroPartition, the memory usage trend of SimpleProcessing5 at rank 2 process, from 50.1 - 50.12 MiB

For *AstroIteration*, the memory trace is presented at Figure 5.10. (process 0: process operation of InternalExtinction3, read 0: read operation of InternalExtinction3, write 0: write operation of InternalExtinction3; process 1: process operation of read0, read 1: read operation of read0, write 1: write operation of read0; process 2: process operation of GetVOTable1, read 2: read operation of GetVOTable1, write 2: write operation of GetVOTable1; process 3:

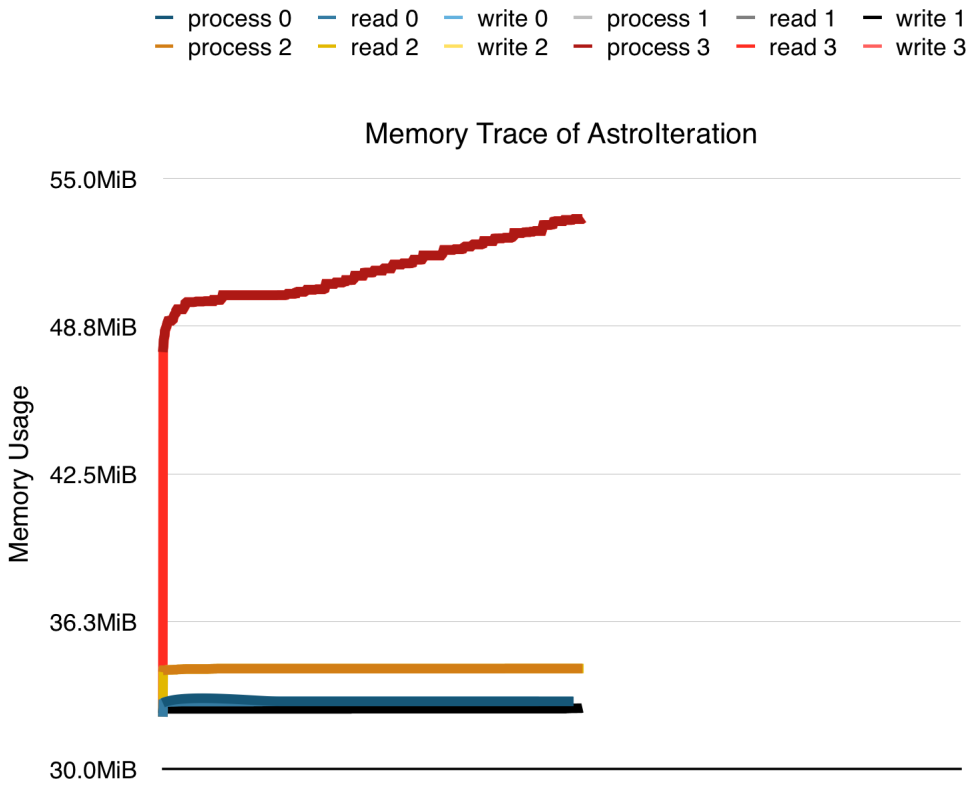

Figure 5.10: During enactment of AstroIteration, the memory usage trend of PEIs on process, write and read

process operation of FilterColumns2, read 3: read operation of FilterColumns2, write 3: write operation of FilterColumns2)

In this line chart, we can see that it took largest RSS to finish process operation of FilterColumns2, while write operation of read0 took the least space to complete. The memory usages of all operations, except for read and process operation of FilterColumns2, were stable during the whole execution time. In another word, there was barely dramatic changes on RSS during the enactment. The operations (i.e. read, write, process) of GetVOTable1 took the second highest average memory usage, coming next to FilterColumns2.

The overall situation of memory consuming of *AstroIteration* was better than that of *AstroPartition*, since only one PEI of the former workflow instance took a memory usage around 50 MiB with others keeping stable at 30 to 35 MiB. In contrast, the latter workflow instance, *AstroPartition* had three PEIs which continuously ran on processes taking 50-MiB memory.

Therefore, in terms of memory consumption, the *AstroIteration* overtook *AstroPartition*.

Chapter 6

Conclusion and Future Work

dispel4py scientific workflow system enables users to process large-scale data with modest computing facilities. It streams data to avoid unnecessary I/O cost, which would happen to interaction with file system. However, organizing the enactment of streaming workflows is difficult because it needs to allocate the whole workflow at once, which requires a good understanding of each PEI's enactment behaviour.

In order to help users and developers get a clear image on the enactment performance of dispel4py workflows, we proposed a profiling mechanism, which is made up of two parts: profiling framework (PF) and performance database (PDB). Working in Multi-Processing environment, PF intercepts workflow instance to gather relevant parameters. Once the workflow terminates, it performs simple operations on raw data collected and then flushes it to PDB. These actions are performed in background, thus do not have effect on workflow enactments. In order to demonstrate the feature and value of PDB, we implemented the profiling mechanism to two dispel4py workflows, on local server and cluster respectively. At last, benchmarks of these two workflows were studied. The data analysis showed the efficiency of them in different angles.

dispel4py profiling mechanism currently applies to Multi-Processing. We will adapt it to other mappings (MPI and Apache Storm). We will extend the range of experiments to a representative set of dispel4py workflows and scales. Work is underway to use dispel4py profiling mechanism to evaluate automatic optimization on instantiation of workflow, which involves decisions such as processes allocations.

Bibliography

[1] Filguiera, R., Klampanos, I., Krause, A., David, M., Moreno, A., and Atkinson, M. (2014, November). dispel4py: A python framework for data-intensive scientific computing. *Proceedings of the 2014 International Workshop on Data Intensive Scalable Computing Systems* (pp. 9-16). IEEE Press.

[2] Yu, J., and Buyya, R. (2005). A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4), 171-200.

[3] M. Gokhale, J. Cohen, A. Yoo, and W.M. Miller. (2008). Hardware Technologies for High-Performance Data-Intensive Computing. *IEEE Computer*, 41(4), 60-68.

[4] Callaghan, S., Maechling, P., Small, P., Milner, K., Juve, G., Jordan, T. H., ... and Brooks, C. (2011). Metrics for heterogeneous scientific workflows: A case study of an earthquake science application. *International Journal of High Performance Computing Applications*, 1094342011414743.

[5] Juve, G., Chervenak, A., Deelman, E., Bharathi, S., Mehta, G., and Vahi, K. (2013). Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3), 682-692.

[6] Mao, M., and Humphrey, M. (2011, November). Auto-scaling to minimize cost and meet application deadlines in cloud workflows. *In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (p. 49). ACM.

[7] Liew, C. S., Atkinson, M. P., Ostrowski, R., Cole, M., van Hemert, J. I., and Han, L. (2011). Performance database: capturing data for optimizing distributed streaming workflows. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 369(1949), 3268-3284.

[8] Tansley, S., and Tolle, K. M. (Eds.). (2009). The fourth paradigm: data-intensive scientific discovery (Vol. 1). Redmond, WA: Microsoft Research.

[9] Vahi, K., Harvey, I., Samak, T., Gunter, D., Evans, K., Rogers, D., ... and Jones, A. (2013). A case study into using common real-time workflow monitoring infrastructure for scientific workflows. *Journal of grid computing*, 11(3), 381-406.

[10] Jacobs, A. (2009). The pathologies of big data. *Communications of the ACM*, 52(8), 36-44.