Stream-based workflows with dispel4py library

Zheming Wang

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2015

Abstract

This paper is aiming to test the performance of dispel4py, which is a new framework for distributed data-intensive applications. The approach to measure the performance of dispel4py is to re-implement well-known parallel benchmarks which are one MPI benchmark and one Hadoop MapReduce benchmark as stream-based dispel4py workflows. The result obtained in this project mainly consists of running time of the original and translated benchmarks, which are the standard measurement for efficiency. The impact of the comparison of original and translated benchmarks will be used to improve the performance of dispel4py in the future.

Keywords: dispel4py, data-intensive, stream-based, MPI, Hadoop MapReduce, comparison

Acknowledgements

I would like to express my special thanks of gratitude to my supervisor Rosa Filgueira who gave me the golden opportunity to do this wonderful project on the topic “Stream-based workflows with dispel4py library”, which also helped me in doing a lot of research and I came to know about so many new things in parallel computing.
Secondly I would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or prefessional qualification except as specified. I have read and understood the University's plagiarism guidelines.

Contents

List of Figures

Chapter 1

Introduction

In the resent years, there has been an unstoppable growing trend of data-intensive computing which has been used in many domains, especially in science and engineering. The data-intensive computing has two obvious characteristics: extremely large scale of data and complex data types. In addition, the sources and usage of data are diverse (Hey et al., 2009). Data which could be captured by instruments, generated by simulations or sensor networks will be used for analysis, data mining, data visualization and data exploration. Due to these characteristics, the programming of data-intensive computing may become a tough problem for the researchers and scientists who are not familiar with programming. As a result, a new dynamic and agile framework, dispel4py, emerges in response to the need of data-intensive computing.

1.1 Motivation

Dispel4py is "a new Python framework for describing abstract stream-based workflows for distributed data-intensive applications" (Filguiera et al., 2014). By using dispel4py, scientists can merely concentrate on computation, avoiding being struggled with the complicated details of computing tools they use. So far, dispel4py has four different real-time mappings: Apache Storm (Storm, 2015), MPI, multi and simple, which help the local development be transited to scalable execution smoothly and enact in many different environments, including: PC, laptop, workstation and cluster. The dispel4py is based on python, which has been chosen by many scientists (Koepke, 2010), because of its dynamic data type system, automatic memory management, rich and comprehensive third-party libraries and easy installation in different environments.

However, as a new framework, dispel4py has not been tested properly because it has not

been compared with other parallel environments. Therefore, the motivation of this project is aiming at testing the performance of dispel4py by re-implementing two famous parallel benchmarks as stream-based dispel4py workflows, comparing with original ones and studying the performance of those benchmarks and workflows in different platforms.

1.2 Objective

The objective of this project is to develop a suite of benchmarks to measure the performance of dispel4py. Before building up the benchmarks of dispel4py, the original benchmarks should be selected. The benchmarks selected to be translated are one Integer Sort benchmark from NAS parallel benchmark (NPB) suite (Bailey et al., 1991) and one page rank benchmark from HiBench benchmark suite (Huang et al., 2010). The NPB suite is a MPI benchmark suite and programmed in C and Fotran and HiBench suite is a Hadoop benchmark suite programmed in Java. The result of the project shows that dispel4py is quite scalable framework which means the program develop in laptop can be directly run in a cluster without any modification. In addition, dispel4py works really well or even better with relatively small size of data than MPI or Hadoop MapReduce. On the other hand, some drawbacks of dispel4py also have exposed which should be improved in the future.

1.3 Structure of the document

In the remainder of this dissertation, it will start with background of the project in Chapter 2. After that, in the chapter of description of work undertaken, the conceptual design of the workflows, actual implementation of the workflows as well as the difficulties and their solutions will be introduced in detail. The third part of the thesis is evaluation, the performance of the original benchmark and translated dispel4py workflow will be compared and analyzed in this part. The dissertation will end with a conclusion part, which contains remarks and observations, unsolved problem, suggestion for further work.

Chapter 2

Background

In this Chapter, the background of the project will be introduced. First of all, it is a brief conception of workflow. Secondly, the frameworks which are used in the project: dispel4py, MPI, MapReduce will be illustrated. At the end of this chapter, two benchmark suites used to be translated in the project will be presented: NAS parallel benchmarks and HiBench.

2.1 Scientific Workflow

Scientific workflow system is designed to automate a repetitive cycle which is able to move data to a computer for analysis or simulation, execute the computations and store the output results (Deelman et al., 2009). It consists of four parts in general: composition, mapping, execution and provenance.

During composition period, the user or programmer can define the steps and dependencies of a workflow in three methods: textual, graphical and mechanism-based semantic models. After that, the workflow will be represented in a number of forms, including: ubiquitous directed graph variants (most common), Petri nets (Peterson, 1981), Unified Modelling Language (UML) (Fowler, 2004), Business Process Modelling Notation (BPMN) (White, 2008) and some other lesser widespread process modelling tools. Before the workflow mapping, the execution control model need to be specified at first. There are two major classes: control flows which represent a transfer of control from preceding task to the following one and data flows in which the data consumer starts after data producer has finished.

The second part of workflow is mapping and an executable workflow will be generated within this period according to its description. The mapping can be performed either by user or workflow system, while in more advanced case, the workflow is designed over the target execution environment.

Thirdly, after the composition and mapping finished, the workflow steps into execution stage. Every modern workflow has its own model of execution and fault tolerance.
Finally, as a critical component, data provenance is used to record the history of creation of data objects. The user can reproduce the result of workflow by the data provenance if it is complete.
There are some modern scientific workflows system, like: Taverna (Wolstencroft et al., 2013), Kepler (Altintas et al., 2004) and Pegasus (Deelman et al., 2015). The Taverna workflow suite is design to generate complicated analysis pipelines which is combined of distributed web services and/or tools in local. There are many suitable platforms to execute the pipelines, laptop or PC locally, and supercomputers, Grids or cluster which are large computation infrastructures. Kepler is able to seamlessly combine the conceptual workflow design with execution and runtime interaction, get access to the data stored locally and remotely as well as local and remote service invocation. With the help of Kepler, the procedure of design, execution, monitoring and re-running of scientific workflow will be significantly simplified. Pegasus workflow system which has been used in the HiBench benchmark suite, is able to take responsibility for workflow translation, job execution, data management, execution monitoring and failures handling. It can be divided into two parts: firstly mapping the high-level workflow by "submit host" and then executing in a cluster or other execution environments, which is quite similar to dispel4py.

2.2 dispel4py

2.2.1 Terminologies in dispel4py

The dispel4py is composed of a number of conceptions:

- Processing element (PE): The foundational unit of dispel4py, can be looked as a node in a workflow graph. Typically, it is the representation of a phase of scientific method or a data-transforming operator which contains an algorithm or a service. The input and output of a processing element is unfixed: the root PE may just has outputs; the tail PE may merely has inputs; the intermediate PE may inputs as well as outputs. Generally, it is defined as a class in python.

  There are many kind of PE available: Generic PE, Iterative PE, Consumer PE, Producer PE, Simple Function PE, and create_iterative_chain.

  a). Generic PE could has unlimited number of inputs and outputs, and it is the super

class of other PEs. Generic PE is the only option, if the programmer wants to use grouping.

b). Iterative PE only has one input named “input” and one output named “output”, it processes one and produces one data unit in each iteration.

c). Consumer PE normally is used as a tail PE, so it has 1 input named “input” and no output.

d). Produce is generally used as a root PE, so it has no input and 1 output named “output”.

e). Simple Function PE is similar to Iterative PE, only has 1 input and 1 output. The difference is it only implement “_process” method, which means it cannot store state between calls.

f). create_iterative_chain has 1 input named “input” and 1 output named “output”, which is used for pipeline of functions processing sequentially. Normally, it will create a composite PE.

Each PE (except Simple Function PE) has three methods: “_process”, “preprocess” and “postprocess” to process data. “preprocess”, as its name, it makes preparation for data processing and “postprocess” is normally invoked to process local stored data after last block has been processed.

- Instance: An executable copy of PE. Each PE can be translated into multiple instances because of either it is called in multiple methods or broadening the data throughput in parallel. The number of instances of a PE can be set manually, otherwise, it will be controlled by the dispel4py underlying framework by default.
- Connection: The “transportation bridge” between the output of previous PE and the input of latter PE. In fact, the rate of data production and consumption is different so that a buffer is necessary part of a connection. Once the buffer is full, temporarily stop the producer PE. On the contrary, if the buffer is empty, pause the consumer PE.
- Composite processing element: PE for sub-workflow or a set of sub-workflows, usually consists of several PEs. Composite PE is quite practical when the workflow is becoming increasingly complicated.
- Partition: A set of PEs which are executed in the same process. These PEs normally do not need high CPU and RAM demands, but the enormous data-streaming congregates here.

- Graph: The connections between PEs and the topology of the workflow. The type of graph is free, it could be pipeline, grid or tree.
- Grouping: The pattern of input communication between PEs, including: shuffle (random), group-by (specified), one-to-all and all-to-one. Shuffle is the default setting of grouping, which will distribute data randomly. Figure 2.1 shows the shuffle grouping between P2 and P3.

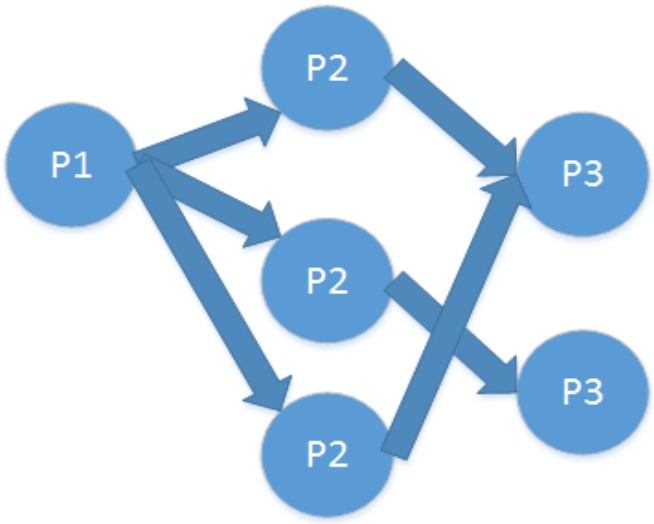


Figure 2.1: Shuffle grouping.

Group-by is similar to MapReduce, the data with same feature will be sent to same instance of PE, while in MapReduce, the tuples with same key will be sent to same reducer. Figure 2.2 shows the group-by grouping, the grouping feature is gender.

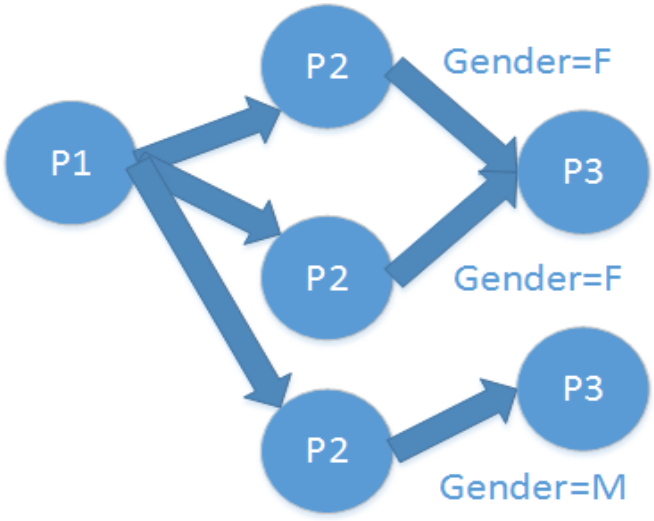


Figure 2.2: Group-by grouping.

One-to-all is usually used for broadcasting, like MPI_bcast, the instances of former PE send copies of their output to all connected instances of latter PE. Figure 2.3 shows the one-to-all grouping between P2 and P3.

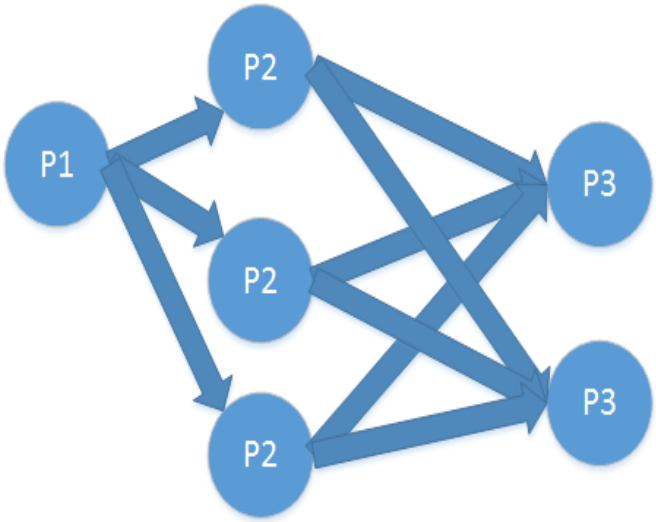


Figure 2.3: One-to-all grouping.

All-to-one is used for data aggregation in general, like MPI_reduce, the instances of former PE send all the data to one instance of latter PE. Figure 2.4 shows the all-to-one grouping between P2 and P3.

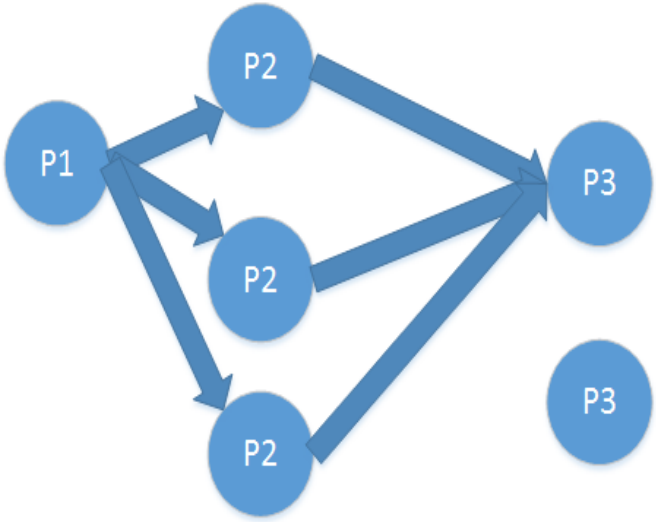


Figure 2.4: One-to-all grouping.

2.2.2 Mappings of dispel4py

The dispel4py has four mappings: Apache Storm, MPI, multi and simple. After describing the workflow, users could simply select one mapping via command line to execute their workflows.

Simple mapping is used for development and local testing. The PEs of the workflow graph are executed by a single process sequentially. The command line when using simple mapping is as follow (dispel4py org, 2015):

```
dispel4py simple <module>
[-a graph attribute within the module]
[-f file containing the input dataset in JSON format]
[-d input data in JSON format]
[-i number of iterations]
```

The parameter <module> typically is the path to the file or python module which creates the workflow graph. The parameter "-a" is the name of the graph attribute in the module. Normally, the "-a" parameter could be ignored, if there is only unique one graph in the dispel4py workflow. However, sometimes there could be more than one graphs, e.g. there is a composite PE. In that case, parameter "-a" should refer to the graph including composite PE rather than the composite PE. If "-f" parameter is provided, "-d" and "-i" will be ignored and if "-d" parameter is provided, "-i" will be ignored again. The "-i" parameter is used to define the number of iterations of root PEs, while the graph will only be executed once by default without "-i" parameter.

The multi mapping is used in a shared memory system by using python build-in multiprocessing package. The command line when using multi mapping is as follow:

```
dispel4py multi -n <number of processes> <module>
[-f file containing the input dataset in JSON format]
[-i number of iterations]
[-d input data in JSON format]
[-a attribute] [-s]
```

All parameters are as same as those in simple mapping, except the new argument "-s". The new parameter "-s" is designed to force the partitioning of the graph to be wrapped and executed in the same process. The MPI mapping is used in the MPI-powered parallel environment by using any MPI implementation. The command line when using MPI mapping

is as follow:

```
mpiexec -n <number mpi_processes> dispel4py mpi module
[-f file containing the input dataset in JSON format]
[-i number of iterations/runs]
[-d input data in JSON format]
[-a attribute]
[-s]
```

All parameter of MPI mapping are as same as those in multi mapping. N.B. The root PE in multi and MPI mapping is limited to be executed in one process.

The dispel4py workflow graph can be translated to a Storm topology and then executed in a Storm-powered cluster. The command line when using Storm mapping is as follow:

```
dispel4py storm -m {local,remote,create}
[-r resourceDir]
[-f file containing the input dataset in JSON format]
[-i number of iterations/runs]
[-d input data in JSON format]
[-a attribute]
[-s]
module [topology_name]
```

The module parameter here cannot be a file path, it should be a python module which create a workflow graph. The parameter "-m" specifies the execution mode of the Storm topology: local mode (local), submit to a Storm cluster (remote), create a Storm topology and resources in a temporary directory (create). The argument "-s" is also different from that in multi and MPI mapping, it now stands for saving Storm topology and resources.

2.2.3 Example of dispel4py workflow

In order to introduce dispel4py more practically, there is an example of dispel4py workflow. It firstly generates a random integer number, which will be checked whether it is a prime or not in the second step. In the case that the random number is a prime, a message will be printed out. The Figure 2.5 shows the code and graph of PrimeChecker.

```
from dispel4py.base import ProducerPE, IterativePE, ConsumerPE
from dispel4py.workflow_graph import WorkflowGraph
import random

class NumberProducer(ProducerPE):
    def __init__(self):
        ProducerPE.__init__(self)
    def _process(self):
        result= random.randint(1, 1000)
        return result

class IsPrime(IterativePE):
    def __init__(self):
        IterativePE.__init__(self)
    def _process(self, num):
        self.log("before checking data - %s - is prime or not" % num)
        if all(num % i != 0 for i in range(2, num)):
            return num

class PrintPrime(ConsumerPE):
    def __init__(self):
        ConsumerPE.__init__(self)
    def _process(self, num):
        self.log("the num %s is prime" % num)

producer = NumberProducer()
isprime = IsPrime()
printprime = PrintPrime()
graph = WorkflowGraph()
graph.connect(producer, 'output', isprime, 'input')
graph.connect(isprime, 'output', printprime, 'input')
```

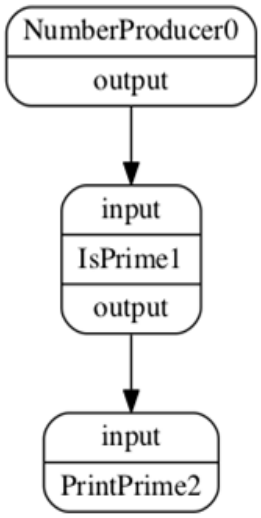


Figure 2.5: PrimeChecker.

If the file name is “prime.py”, it can be run with following command lines:

dispel4py simple prime.py -i 10

dispel4py multi prime.py -i 10 -n 4

mpiexec -n 4 dispel4py mpi prime.py -i 10

dispel4py storm prime.py -n 4 -i 10.

The default processes distribution of dispel4py is distributing equally. For example, if there are n nodes in graph and m processes available, every node will be executed by m/n processes and the other m%n processes are idle.

2.3 Message-Passing Interface (MPI)

MPI (Forum, 2015) is a message passing library interface specification for parallel programming. MPI its self is a standard or specification rather than a language. It comes up with a context "communication space" which is used for parallel message communication of processors. Data is moved from one process to another process efficiently through cooperative operations on each process. It could be easily employed either on distributed-memory multi computers or shared-memory multiprocessors with C or Fortran programming language. There are many version of implementation of MPI available, e.g. MPICH (Gropp et al., 1996) and Open MPI (Gabriel et al., 2004). Because of its portability and ease of use, it has been widely used in high performance computing, cooperates with OpenMP (Dagum and Enon, 1998), a standard for shared-memory programming, as hybrid parallel programming (Rabenseifner et al., 2009).

2.4 MapReduce

As a key component of Hadoop, MapReduce (Dean and Ghemawat, 2008) is a quite practical and effective programming model which has been widely applied to process large data sets with distributed algorithm on a cluster. Google (Lämmel, 2008) is one of the faithful users and active spreaders of MapReduce. Typically, a MapReduce program can be divided into two parts: a map function and a reduce function, both written by the programmer. In addition, all input data and output data is stored in Hadoop distributed file system (HDFS). Sometimes, the data will be processed in advance in map function by a combiner function. Moreover, MapReduce makes the users to just consider in a data-centric method: they only need to concentrate on how to apply transformations to the data sets, and leave the particular implements of fault tolerance, distributed execution and inter-machine network communication to be handled by the underlying runtime system. As a result, MapReduce is friendly to all programmers even though they do not have any experience of parallel programming.

2.5 NAS parallel benchmark (NPB)

NAS parallel benchmark is a small benchmark package developed by NASA in 1990s in order to help evaluate the performance of parallel supercomputers. The computational fluid dynamics (CFD) applications are the origin of these algorithmic benchmarks. Initially, the

original benchmark package was composed of 5 kernels and 3 pseudo applications:
5 kernels:

IS - Integer Sort, random memory access

EP - Embarrassingly Parallel

CG - Conjugate Gradient, irregular memory access and communication

MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive

FT - discrete 3D fast Fourier Transform, all-to-all communication

3 pseudo applications:

BT - Block Tri-diagonal solver

SP - Scalar Penta-diagonal solver

LU - Lower-Upper Gauss-Seidel solver

As the rapid development of software and hardware, the benchmark suite now has been extended to evaluate the performance of unstructured adaptive mesh, parallel I/O, multi-zone applications and computational grids. Furthermore, the problem size or the size of test data also has been extended to larger classes. Two popular programming models: MPI and OpenMP are available for the implementation of NPB. In the project, the version of NPB suite used for translation is 3.3.1.
The reasons for choosing Integer Sort benchmark are:

1. Enormous data size: Input data size could reach hundreds of MB or even above 1 GB.

2. Complicated data handling: The algorithm of Integer Sort is a combination of inter-process bucket sort and intra-process count sort.

2.6 HiBench

HiBench Suite is a comprehensive and realistic bigdata micro benchmark suite supported by Hadoop or Spark and developed by Intel. So far, it consists of 10 general micro workloads, including: 4 micro benchmarks: Sort, WordCount, TeraSort, Sleep; 1 SQL benchmark: Scan, Join, Aggregate; 2 web search benchmarks: PageRank, Nutch indexing; 2 machine learning benchmarks: Bayesian Classification, K-means clustering; 1 HDFS benchmark: enhanced DFSIO. This benchmark suite is designed for evaluating and characterizing the Hadoop framework in 5 different aspects: speed, throughput, the Hadoop distributed file system bandwidth, system resource utilizations and data access patterns. HiBench is an ideal benchmark suite for testing performance as it is easily configured and concisely

coded. The latest release of HiBench is 4.0, but in the project, the version of HiBench used is 2.2 (branch MRv2 in the github), because Spark support is unnecessary.
The reasons for choosing PageRank benchmark are:

1. Widely used in web search engines. Despite the implementation of PageRank in HiBench is not complex, it is still a quite practical algorithm.

2. Extremely enormous data size it could reach. According to Google (Google, 2015), there are over 60 trillion individual web pages included in Google's search engine. The links between these web pages could be 60 trillion times 60 trillion, an extremely enormous number.

Chapter 3

Description of work undertaken

This chapter will start with explaining how the original two benchmarks which are Integer Sort and PageRank work. After that, the conceptual design and actual implementation of dispel4py workflows will be introduced in full detail with help of pseudo code.

3.1 Original benchmarks

In this section, how the original benchmarks work will be thoroughly explained, which will lay a foundation for the conceptual design of translated workflows.

3.1.1 Integer Sort

The Integer Sort benchmark is from NAS parallel benchmark suite, implemented in C programming language with MPI. It tests both integer computation speed and communication performance of the running platform.
The algorithm of this benchmark is a combination of bucket sort and count sort. The algorithm assumes that, there is a large size array of integers which follows Gaussian distribution (Grün and Hillebrand, 1998) , the size could be up to 2^29. In the real MPI implementation, each process generates its own initial array as a part of the total large size array. The array in the first process will slightly modify 2 integers at the beginning of every iteration. After that, every process proceeds to internal bucket sort. Each bucket represents a range of integer and the element in the array will be put into the bucket if it belongs to the range the bucket represents. In the meantime, some testing elements will be put into the end of bucket array in specific process. Figure 3.1 shows the whole integer array and inside of a process.

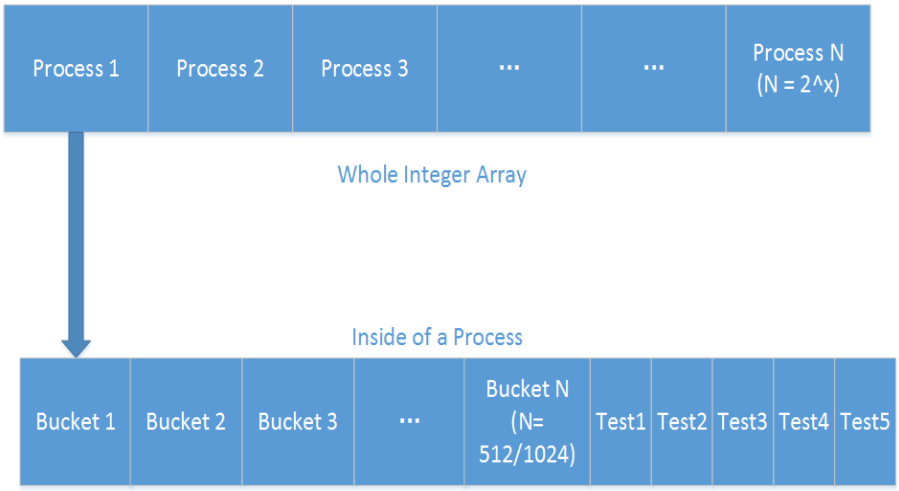


Figure 3.1: Whole array and inside of a process.

As bucket set part finishes in every process, every process will call the MPI_Allreduce with SUM operation to aggregate the size of each bucket in each process and the result of aggregation stored in an array named "bucket size totals". Bucket size totals[i] means totally how many elements belong to this bucket.

The next step is to redistribute the buckets, which is aiming to minimize the range of number in a process as small as possible. For example, if there are 8 processes and the range of key is 1 to 1024. The expected range of the first process is 1 to 128 after redistribution. The original Integer Sort benchmark invokes the collective operation MPI_alltoallv to carry out the redistribution operation: every process sends the its buckets to corresponding destination (process) and receives the buckets sent from other processes or itself. The Figure 3.2 shows the procedure of bucket size aggregation and bucket redistribution conceptually.

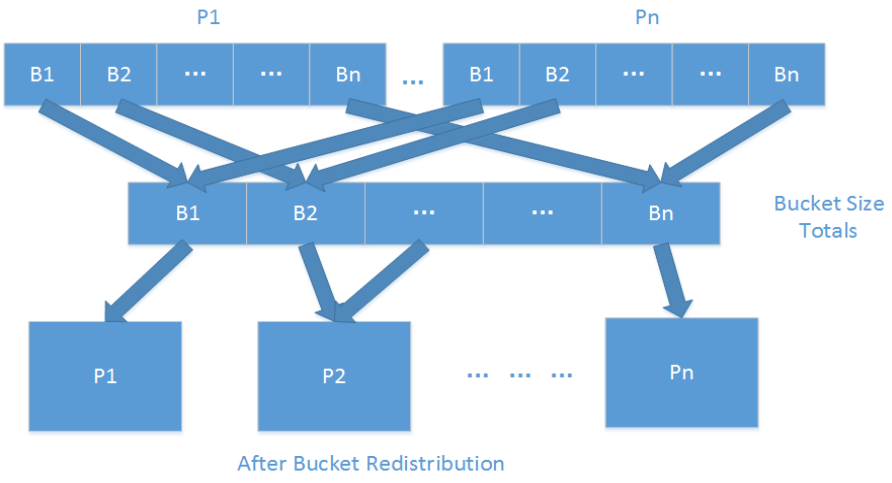


Figure 3.2: Bucket size aggregation and bucket redistribution.

Since the bucket distribution finishes, the range of numbers in a process has been much smaller than before, which means the memory cost is now acceptable for count sort. Therefore, the next step is count sort, which directly counts how many times each integer occurs in the "new" sequence.

The final step of the Integer Sort called partial verify is to probe whether the test keys have been computed correctly. The 5 test elements previously stored at the end of bucket size array come into use. Firstly, find out which process should the test key belongs to now. Secondly, compare the actual rank of the test element to its algorithmically expected rank which has been predefined. If these ranks are all the same, partial verify passes, or it fails.

In the original Integer Sort benchmark, these previous steps except sequence generation will be looped for 10 times and the total running time of each process will be recorded. The maximum running time will be selected as the running time of the IS benchmark for measurement.

In addition, there is another untimed step named full verify. In this step, every process will fully check that whether all integers have been sorted correctly. In other words, every process needs to make sure that no former key is large than the next key. After the intra-process check, it is the inter-process check, which is designed to check that is there any last key in the former process is larger than the first key in next process. As the full verify passes, it means all integers have been sorted correctly.

3.1.2 PageRank

PageRank (Page et al., 1999) is an extremely famous web search engine algorithm firstly put forward by Larry Page and Sergey Brin in 1998 used to measure the relative importance of a web page. HiBench suite implements PageRank algorithm by using MapReduce. The algorithm assumes that the network is a strongly connected directed graph, web pages are treated as nodes in this graph. The link relation between two web pages can be stored as an adjacency matrix. The size of the adjacency matrix depends on the number of web pages, n web pages will generate an n*n matrix. If web page A has n hyperlink(s) to web page B, then the value of adjacency matrix[A][B] is n ($n > 0$).

However, matrix is not an appropriate data structure for MapReduce which seeks for key-value tuples. As a result, the adjacency matrix should be transformed into key-value tuples: key represents the source of a hyperlink, while value represents the destination of that hyperlink. One tuple represents one hyperlink, which means if there are a few hyperlinks between two pages, there are corresponding number of tuples. The Figure 3.3 shows the

transformation process from an adjacency matrix to key-value tuples.

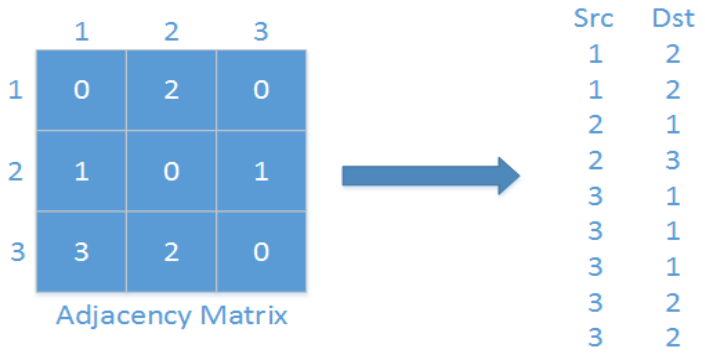


Figure 3.3: Adjacency matrix to tuples.

These tuples will be stored in several edge files as one part of input of PageRank benchmark. Another part of input is the initial rank values of each web page. All rank values are equal which is 1 divided by the number of web pages and these rank values are stored as tuples (node-initial rank) with the edge files in HDFS.

After the previous preparation steps done, the main data manipulation part of PageRank begins. The main part is composed of two stages or two MapReduce jobs. In the first stage, the mappers do nothing on the tuples, simply passes the tuples to the reducers. The data received by reducers has been shuffled and sorted. The job of reducers is: firstly count the out-degree of each node; secondly divide the initial ranks by their own out-degrees; thirdly output the new subrank tuples(destination-divided initial rank) as well as initial rank tuples(node-initial rank) to a temporary file in HDFS. The Figure 3.4 shows the procedure of stage one:

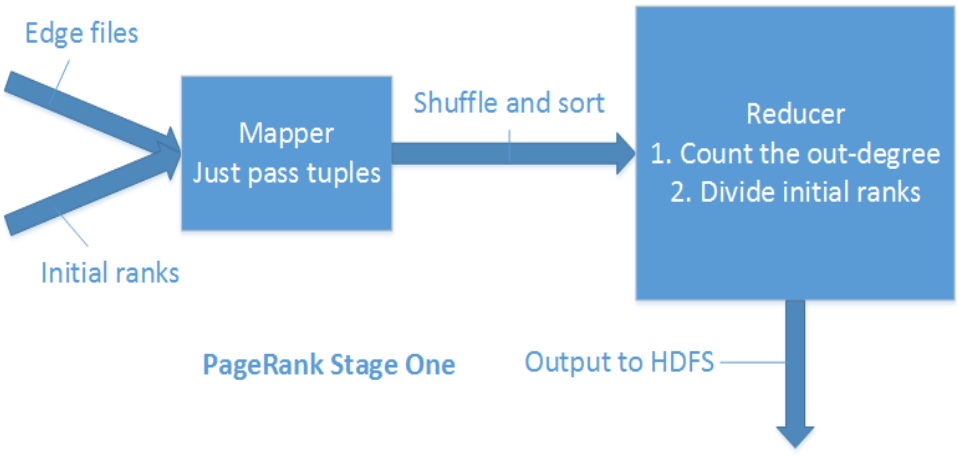


Figure 3.4: PageRank stage one.

In the second stage, the mappers do nothing as the same as the mappers in the first stage. The job of reducers is: aggregate all subrank values of each node at first; after the aggregation, new rank = new rank $\times\ e + \frac{1-e}{\text{number of nodes}}$; finally, compare the new rank value of each node to its initial rank value: if the absolutely difference is smaller than a predefined threshold which is $\frac{1}{10\times\text{number of node}}$, the rank value of this node has converged, otherwise, it is not converged and next iteration will begin if current iteration is not the last iteration. If there is a next iteration, the new rank values will replace the previous rank values as initial rank values to be input.

For the random factor e above, it represents the possibility that the web surfer will stay in the destination web page after redirection, whereas 1-e stands for the possibility that the web surfer will enter a new URL rather than stay in this web page. Therefore, the factor $\frac{1-e}{\text{number of nodes}}$ means although a new URL has been entered, the web surfer goes back the current web page, a quite tiny possibility. In the original PageRank benchmark, e=0.85. The Figure 3.5 shows the procedure of stage two:

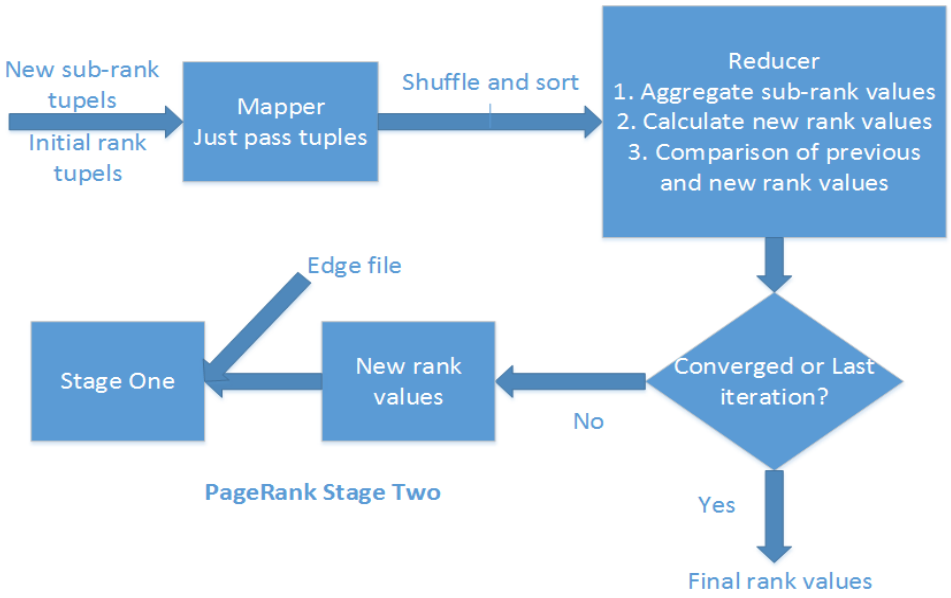


Figure 3.5: PageRank stage two.

From the aspect of linear algebra, the procedure of PageRank can be summarize as this formula: $V' = eMV + (1-e)a$,where V' is the vector of new rank values, e is random factor mentioned before, M is the adjacency matrix, V is the vector of previous rank values and a is a vector consists of a series of $\frac{1}{\text{number of nodes}}$.

3.2 The dispel4py workflows

In the last section, the details of how the original Integer Sort and PageRank benchmarks work have been explained. In this section, the conceptual design and actual implementation of dispel4py workflows will be introduced in same order, first Integer Sort first, then PageRank.
In order to compare the original and translated benchmarks fairly, all operations of the original benchmark will be translated in same way and no optimization will be applied, for example: the original PageRank benchmark does not use combiner, so the translated benchmark should not use any combining strategy either.

3.2.1 Design of Integer Sort workflow

According to the original Integer Sort benchmark, the translated benchmark has been divided into 6 main parts (PEs): initializer, bucket setter, bucket size aggregator, bucket redistribution sender, bucket redistribution receiver and partial verifier. Moreover, there are three more PEs defined: timer, full key sorter and full verifier and one initial array generator program ArrayGenerator.py (written in python).
N.B. The "process" below in the workflow except in ArrayGenerator.py is not real process. That "process" stands for conceptual or simulative process, which is used to simulate the real process in MPI.

- **ArrayGenerator.py:** This python script is used to generate the initial whole array which follows Gaussian distribution. Due to the limitation of root PE which only can be executed by 1 process (or) to avoid input duplication, this part has been separated from the Initializer PE. In original Integer Sort benchmark, the array of each MPI process is generated by each process itself, which is parallel generation. However, to generate 2^25 integers which is just class B of problem size, it will take up about 250MB in 1 process (or). Obviously, it is not a wise choice to generate such a large array every time when execute the workflow. As a result, the generator has been separated from the initializer PE as an individual python script.

- **Initializer:** This PE is the root PE of the workflow/graph, it has 1 input "input" and 1 output "output". The input is the number of current iteration, and the output is subarray with its rank and current iteration to bucket setter PE. The function of initializer PE is to read and pass the initial whole array "process by process".

- **Bucket setter:** This PE is primarily used to put the element into its corresponding bucket, and bucket sort starts from this PE. It has 1 input "input" and 3 outputs "output", "time" and "bucket_size". The input is the output from Initializer PE, which is a subarray with its rank and current iteration. The first output "time" is to timer PE, used to makes the timer start. The second output "bucket_size" is to bucket size aggregator PE, used to sum up the total size of each bucket. The last output "output" is to bucket redistribution sender, used for bucket redistribution. The work of this PE done is: slightly modify the first subarray; count how many integers in each bucket; put the integers into its bucket.

- **Bucket size aggregator:** This PE plays a role in the workflow as the role of MPI_Allreduce function plays in the original benchmark. It has 1 input "bucket_size" with all-to-one grouping from bucket setter PE and 1 output "bucket_size_totals" to bucket redistribution sender PE. All bucket size arrays will be sent to this PE for sum-up. The result will be stored in an array named "bucket size totals" and sent to bucket redistribution sender PE.

- **Bucket redistribution sender:** This PE is a part of bucket redistribution process. It has 2 inputs, one is from bucket setter PE which is "input" and another one is from bucket size aggregator PE which is "bucket_size_totals" with one-to-all grouping. The unique output "output" of this PE will be sent to the bucket redistribution receiver PE. The function of this PE is to divide the original array in each process into many subarrays according to the array "bucket size totals" sent from bucket size aggregator PE and then send the subarrays with their new process ranks to bucket redistribution receiver PE.

- **Bucket redistribution receiver:** This PE is another part of bucket redistribution process. This PE has 1 input "input" from sender PE with group-by grouping and 1 output "output" to partial verifier PE. It cooperated with bucket redistribution sender, plays a role in the workflow as the role of MPI_Alltoallv function plays in the original benchmark. All subarrays sent from sender PE will be grouped by their destinations (their new process ranks) and the subarrays with same new process rank will generate new array of that process rank. After all new arrays have been generated, these new arrays will be sent to the next partial verifier PE.

- **Partial verifier:** This PE is the last component of timed part in the workflow, which means as this PE stops, the timer will stop as well. This PE has 1 input "input" from

bucket redistribution receiver PE, 1 output "time" to timer, 1 optional output "output" to final key sorter for testing the correctness of sorting. The function of this PE is obvious as its name, probe whether the keys have been sorted correctly. Firstly, because the range of number of the new arrays has been much smaller, intra-process count sort begins in this PE: count how many times each number occurs in the new array only, do not generate final sorted array whose keys rank from low to high. With the help of counting result, key probing happens after counting. In the end, this PE sends an output to timer which makes the timer stop. If the current iteration is not equal to the last iteration, this workflow finishes. Otherwise, the optional output to final key sorter will be sent for testing correctness of sorting.

- **Final key sorter:** This PE is an optional PE only used for testing correctness of sorting. It has 1 input "input" from partial verifier and 1 output "output" to full verifier. The input mainly consists of a new array and counting result of this new array. At first, according to the counting result, the final sorted array will be generated here and then check the final array to make sure no previous key is larger than its next key. The output is composed of the maximum key and minimum key of final array, which will be used for inter-check.

- **Full verifier:** This PE is also an optional PE used for testing correctness of sorting. It only has 1 input "input" with all-to-one grouping, no output. All maximum and minimum keys of final arrays are gathered here. The only job this PE does is to check is there any maximum key in previous process larger than the minimum key in the next process.

- **Timer:** As its name, this PE is the timer of workflow, used to record the running time of the workflow. It only has one input interface "input" with all-to-one grouping, but the source of input could be from partial verifier or bucket setter. Running time of all processes will be recorded, added up with previous time if existed and written to a time file after every iteration finishes.

Overall, the design of Integer Sort workflow can be summarized as Figure 3.6

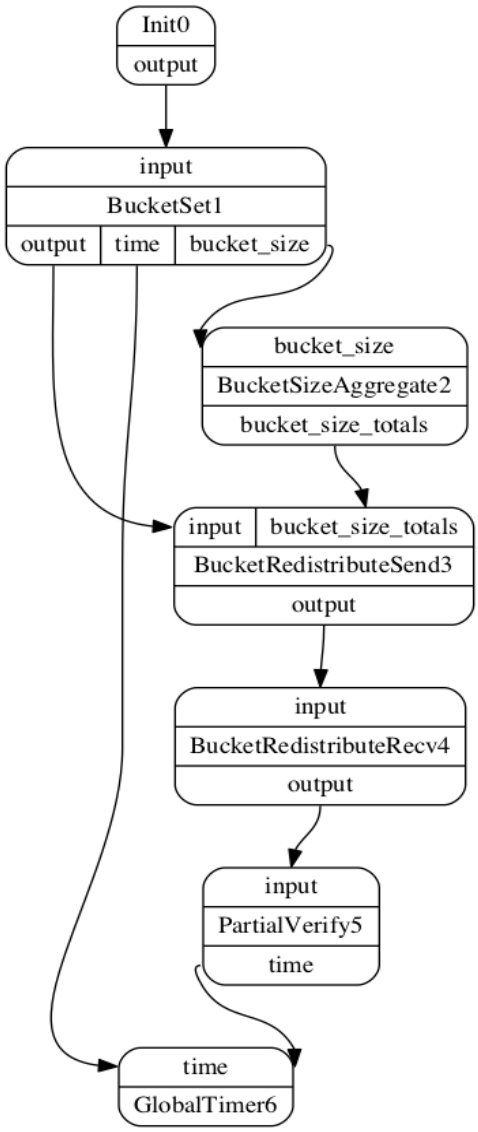


Figure 3.6: Design of Integer Sort workflow.

As Figure 3.6 shows, the Optional PEs (final key sorter and full verifier) are not included in the workflow when testing performance of workflow. There are two main reasons why these two PEs are abandoned. Firstly, these two PEs only work in the last iteration, they do not work in the previous iterations. It is a waste of processor to import these two PEs. Secondly, even in the last iteration, the running time of these two PEs is untimed. Therefore, once the correctness of sorting has been confirmed, these two PEs will be removed from the workflow. In total, there are 7 PEs in the workflow when testing its performance.

3.2.2 Actual implementation of Integer Sort

- **ArrayGenerator.py:** This data generator can be described as the following pseudo code:

```
randlc(x, a)
    r23 = 2^-23; t23 = 2^23; r46 = 2^-46; t46 = 2^46
    a1 = (float)(int)(r23 * a); x1 = (float)(int)(r23 * x)
    a2 = a - t23 * a1; x2 = x - t23 * x1
    t1 = a1 * x2 + a2 * x1; t2 = (float)(int)(r23 * t1)
    z = t1 - t23 * t2; t3 = t23 * z + a2 * x2
    t4 = (float)(int)(r46 * t3); x = t3 - t46 * t4
    return x * r46
find_my_seed(kn,np,nn,s,a)
    nq = nn / np; mq = 0; t1 = a
    while nq > 1 do: {mq++; nq/=2}
    for 1 to mq do: {t2 = randlc(t1, t1)}
    an = t1; kk = kn; t1 = s; t2 = an
    for 1 to 100 do:
        ik = kk / 2
        if 2 * ik != kk do:
            t3 = randlc(t1, t2)
        if ik == 0: break
        t3 = randlc(t2, t2); kk = ik
    return t1
create_seq(seed, a)
    k = max key / 4
    for 1 to number of keys do:
        x = randlc(seed, a)
        x +=randlc(seed, a) for 3 times
        write k * x to file
#main method
for rank as every process:
    create_seq(find_my_seed(rank, num procs, 4 * total keys * ...
        min procs, 314159265.00, 1220703125.00), 1220703125.00)
```

This generator mainly consists of 3 methods: randlc, find_my_seed and create_seq. Function randlc is intended to generator a uniform pseudorandom double precision number in the range (0,1) following Gaussian distribution. The variable “x” and “a” are two references, which means they can be modified in method. Additionally, “a1”,

"x1", "t2" and "t4" will be converted to integers first and converted back to floats. Second function find_my_seed is designed to create a random number which is the first random number for the subsequence belonging to processor rank "kn", and which is used as seed for process "kn" to generate random numbers. Variable "kn" stands for the rank of process, "np" stands for the number of processes and "nn" represents total number of random numbers. Variable "s" and "a" are two constants, which are 314159265.00 and 1220703125.00 respectively. Function create_seq is much simpler than the former two functions. It calls randlc for 4 times and adds all return of randlc up. Then divide the sum by 4 and multiplied by the maximum key predefined, the result will be written to the disk.

- **Initializer:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
iteration = "input"
open array file
for rank as every process (every line in file) do:
    key array = line
    write "output" [rank, key array, iteration]
```

Firstly, the current iteration will be sent into initializer PE as input. Secondly, the array file generated before will be read line by line. Thirdly, output a list to bucket setter PE.

- **Bucket setter:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
rank, key array, iteration = "input"
write "time" [rank, 'start', iteration]
if rank == 0 then: modify some integers in key array
put the test element at the end of bucket size array
for i as every key in key array do:
    bucket size[i] +=1 if key belongs to bucket i
for i as every bucket (i > 0):
    bucket ptrs[i] = bucket ptrs[i - 1] + bucket size [i - 1]
for i as every key in key array:
    key buff1[bucket ptrs[i]] = key if key belongs to bucket i
    bucket ptrs[i] += 1
write "bucket_size" [bucket size]
write "output" [rank, bucket size, key buff1, iteration]
```

After receiving input from initializer PE, send the “time” output to timer PE to start the timer for this process. If “rank” is 0 which means it is the first process, modify a few numbers in its “key array”. Before putting the integers into buckets, test elements will be put at the end of “bucket size” if these test elements belong to this process after bucket redistribution. Then, it is the main stage of bucket setter PE: Firstly, calculate size of each bucket, stored in “bucket size”; Secondly, “bucket prts” will be used to point out where is the start of each bucket; Thirdly, according to “bucket ptrs”, every integer in “key array” will be put into “key buff1”, which can be looked as the procedure of pouring elements into buckets. When the previous steps finishes, bucket setter PE will send “bucket size” output to bucket size aggregator PE and also send “output” output to bucket redistribution sender PE. N.B. In this PE, the initial sequence “key array” has been converted to “key buff1”.

- **Bucket size aggregator:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
1 bucket size = "bucket_size" (grouping='global')
2 for i as every bucket:
3     bucket size totals[i] += bucket size[i]
4 postprocess:
5 write "bucket_size_totals" [bucket size totals]
```

What is different from the previous PEs is grouping mode of input from bucket setter PE is “global”, which means all data will come together in one instance of PE. Hence, this PE only needs one process (or) to be executed. A list “bucket size totals” is defined to sum up all “bucket size” s. In order to ensure every “bucket size” has been summed, this PE overrides method “postprocess” to send output “bucket_size_totals” to bucket redistribution sender PE.

- **Bucket redistribution sender:** This PE is implemented as a generic PE, its pseudo code is as follow:

```
rank, bucket size, key buff1, iteration = "input"
store rank, bucket size, key buff1 in three lists
bucket size totals = "bucket_size_totals" (grouping='all')
j = 0
while both inputs arrived do:
    pop one rank, bucket size, key buff1 out in each list
    for i as every bucket:
        global accumulator += bucket size totals[i]
        local accumulator += bucket size[i]
        if glocal accumulator > (j + 1) * num keys then:
            send count[j] = local accumulator
            if j != 0 then:
                send displ[j] = send displ[j - 1] + send count[j ...
                    - 1]
                process bucket ptr1[j] = process bucket ptr2[j - ...
                    1] + 1
            process bucket ptr2[j] = i
            j += 1; local accumulator = 0
while j < num procs do:
    send count[j] = 0
    process bucket ptr1[j] = 1
    j += 1
last send start = send displ[num procs-1]
last send end = send displ[num procs - 1] + send count[num procs ...
    - 1]
for i as every process (except last process) do:
    write "output" [i, key buff1[send displ[i] upto send ...
        displ[i+1]], process bucket ptr1, process bucket ptr2, ...
        bucket size totals, iteration]
write "output" [num procs, key buff1[last send start upto last ...
    send end], process bucket ptr1, process bucket ptr2, bucket ...
    size totals, bucket size totals, iteration]
```

The grouping mode of input from bucket size aggregator PE is “all”, which means the input will be copied to all instances of this PE. This is a heavy-loaded PE because when the input from bucket size aggregator PE has not arrived, all input from bucket setter PE needs to be stored locally. Therefore, this PE has to create three lists to store the input from bucket setter PE. Once the “bucket_size_totals” arrives, this PE begins to work. Pop one element out respectively from those three list which could be treated as pop a conceptual process out. Meanwhile, defining two accumulator variables: global

accumulator and local accumulator. Every time, the global accumulator will be added up with “bucket size totals[i]”, and similarly, the local accumulator will be added up with “bucket size[i]”. Once the global accumulator surpasses the threshold, the local accumulator will be recorded as “send count[j]” and reset to zero, while “send displ[j]” stands for the starting position of every subarray. Additionally, there are two lists created as array of pointer: “process bucket ptr1” is used to point out the “first bucket” of each process after redistribution, while “process bucket ptr2” is used to point out the “last bucket”. As all subarrays have been divided properly, send “output” s to bucket redistribution receiver PE.

- **Bucket redistribution receiver:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
1 rank, subarray, process bucket ptr1, process bucket ptr2, bucket ...
      size totals, iteration = "input" (grouping=[0])
2 key buff2[rank] = key buff2[rank] + subarray
3 postprocess:
4 for i as every process in key buff2 do:
5     min value = minimum integer in this process
6     max value = maximum integer in this process
7     write "output" [i, key buff2[i], process bucket ptr1, ...
          process bucket ptr2, min value, max value, bucket size ...
          totals, iteration]
```

The grouping mode of input from bucket redistribution sender PE is grouping by the first element in input which is rank of process. All subarrays belongs to same process will be gathered in one instance of this PE to create the new array of each process. After all arrays generated, the maximum and minimum key of new array will be calculated for further check. Then send “output” to partial verifier PE. As this PE terminates, the bucket redistribution has finished.

- **Partial verifier:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
rank, key buff2, process bucket ptr1, process bucket ptr2, min ...
    value, max value, bucket size totals, iteration = "input"
m = total lesser keys
j = total local keys
for i as 0 to j do:
    key buff1[key buff2[i] - min value] += 1
for i as 0 to (max value - min value) do:
    key buff1[i + 1] += key buff[i]
for k as every test element do:
    if k is in the range [min value, max value] then:
        key rank = key buff1[k - min value -1] + m
        check has k been computed correctly
write "time" [rank, 'stop', iteration]
if iteration == max iteration then: (only for correctness)
    write "output" [rank, key buff1, key buff2, j,0, min value, ...
        max value]
```

There is some preparation needed before probing test keys. Firstly, figure out how many lesser keys as “m” and the total number of local keys as “j”. Secondly, count how many keys for each value and result is stored in new “key buff1”. Then convert “key buff1” to a list which refers the rank of every key value. As the preparation part finishes, the key probing parts begins. If the test element is in the range [min value, max value], the actual rank of test element is the rank in local process plus the total lesser keys. Compare the actual rank to the algorithmically expected rank. After that, send “time” to timer PE to stop the timer of this process. Moreover, if the correctness of sorting has not been confirmed, this PE will send optional output “output” to final key sorter PE.

- **Timer:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
rank, op, iteration = "time" (grouping='global')
if op == 'start' then:
    start[rank] = current time
else op == 'stop':
    elapsed[rank] = current time - start[rank]
postprocess:
if iteration == 1 then:
    write elapsed to time file
else:
    read previous time from time file
    for rank as every process do:
        elapsed[rank] += previous[rank]
    rewrite elapsed to time file
    if iteration == max iteration:
        print total running time
```

The grouping mode of input from bucket setter PE and partial verifier PE is “global”, so it is the second PE that only needs one process in the workflow. It has two lists “start” used to record start time of every process, “elapsed” used to record the time cost of every process. If it is the first iteration, the time file will be created and then write “elapsed” to time file. Otherwise, read the time file to get the previous time cost of each process, then for each process, add its previous time cost to “elapsed[rank]” and rewrite the new “elapsed” to the time file.

N.B. The original benchmark selects the maximum running time among the processes as its running time. However, during the performance test of translated benchmark, because of the buffering and delay between different nodes in the cluster, the maximum running time could be double of the minimum when the number of nodes is small. This difference between maximum and minimum running time becomes smaller as the number of nodes increases. Therefore, average running time replaces maximum to make the measurement of running time more accurate.

- **Final key sorter:** This optional PE is implemented as a generic PE and its pseudo code is as follow:

```
rank, key buff1, key buff2, total local keys, total lesser keys, ...
    min value, max value = "input"
for key in key buff2 do:
    key buff1[key - min value] -= 1
    location = key buff1[key - min value] - total lesser key
    key array[location] = key
first value = key array[0]; last value = key array[last]
j = 0
for i as 1 to total local keys do:
    if key array[i - 1] > key array[i] then:
        j += 1
if j == 0 then:
    write "output" [rank, first value, last value, total local ...
        keys]
```

After receiving input from partial verifier PE, it will create a new array named “key array”. And then, according to the counting result in “key buff1”, keys in “key buff2” will be put into “key array” properly. After checking all keys in “key array” follows the rule that former key is not larger than its next key, the “output” will be sent to full verifier PE. The reason for sending number of local keys is in some extreme case, the distribution of keys is unbalanced, which will lead to last few processes do not get any buckets in redistribution. those first keys in “empty” process are 0 rather than an expected meaningful number.

- **Full verifier:** This optional PE is implemented as a generic PE and its pseudo code is as follow:

```
rank, first key, last key, total local keys = "input" ...
    (grouping='global')
store first key, last key, total local keys in three lists
postprocess:
j = 0
for rank as every process do:
    if rank > 0 and total local keys[rank] > 0 then:
        if last key[rank - 1] > first key[rank] then:
            j += 1
if j==0 then:
    print "Fully Verified!"
```

In order to gather all input, the grouping mode of input is “global”. All minimum keys, maximum keys and numbers of local keys will be stored three list “first keys”, “last keys” and “total local keys” correspondingly. The inter-process check happens in “postprocess” method: firstly check is the current process “empty”, secondly check is its last key not larger than the first key in the next process. If all checks pass, the correctness of sorting is confirmed and these two optional PEs are no longer needed in the workflow.

Overall, the initializer, bucket size aggregator and timer PE will be executed by one process. The other PEs, bucket setter, bucket redistribution sender/receiver, partial verifier, they can be executed by multi processes. In other words, they can be executed in parallel. Here is a simple example of processes distribution of Integer Sort workflow running with 32 processes:
BucketRedistributeRecv4: [8, 9, 10, 11, 12, 13, 14, 15]
BucketSizeAggregate2: [7]
BucketRedistributeSend3: [25, 26, 27, 28, 29, 30, 31]
BucketSet1: [18, 19, 20, 21, 22, 23, 24]
GlobalTimer6: [17]
Init0: [16]
PartialVerify5: [0, 1, 2, 3, 4, 5, 6]

3.2.3 Design of PageRank workflow

The design of PageRank workflow is more concise than Integer Sort workflow. The workflow is divided into 5 PEs: initial reader, stage one, stage two, checker and timer. The data for PageRank workflow will be directly copied from HDFS, so there is no need for a data generator.

- **Initial reader:** This PE is the root PE of workflow. It has 1 input “input” which is the current iteration and 2 outputs, one to stage one PE which is “output” and another one to timer which is “time”. The main tasks of this PE are: read the edge file and initial rank values into workflow, pass them to the stage one PE and start the timer.

- **Stage one:** As its name, this PE does the same job as the mapper and reducer in stage one of the original Hadoop benchmark. It has 1 input “input” from initial reader PE and 1 output “output” to stage two PE. In stage one PE, the initial rank value of each

node is divided by its out-degrees as subrank value. Then tuples (destination-subrank value) and (node-initial rank value) will be sent to stage two PE.

- **Stage two:** New rank values are calculated in this PE. It has 1 input "input" from stage one PE and one output "output" to checker PE. The task of this PE is to aggregate all subrank values of each node in order to generate new rank values. After the aggregation, the new rank value will be dealt with random factor e. Finally, the initial and new rank values will be sent to checker PE.

- **Checker:** This PE is used to check whether all rank values have been converged. It has 1 input "input" from stage two PE and 1 output "time" to timer PE. If it is already converged or the current iteration is the final iteration, the new rank values will be written into disk. Otherwise, replace the previous rank values by the new rank values and starts next iteration. In the end, send an output to timer PE to stop the timer.

- **Timer:** The simplest PE in the workflow. It only has 1 input "time" which could come from initial reader PE or checker PE. Once receive the input from initial reader PE, start the timer, while receive the input from checker PE, stop the timer. As the timer PE of Integer Sort PE, the total time cost so far will be store in a time file.

In summary, the design of PageRank workflow can be illustrated as Figure 3.7:

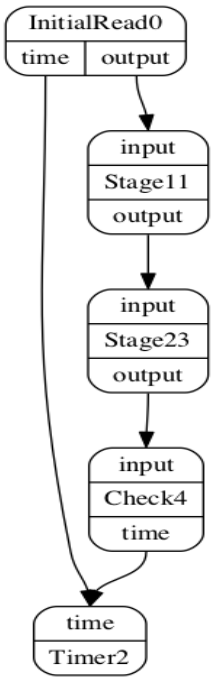


Figure 3.7: Design of PageRank workflow.

As a supplement to Figure 3.7 shows, in the first iteration, the initial rank values are generated by initial reader PE. In the following iterations, the initial reader will read the new rank values instead of generating inside.

3.2.4 Actual implementation of PageRank

- **Initial reader:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
iteration = "input"
if iteration != 1 then:
    read new rank vector
    if rank value starts with 'f' then: stop workflow
write "time" ['start', iteration]
read edge file
for line in edge file do:
    src, dst = line.split
    write "output" [src, dst, iteration]
if iteration == 1 then:
    for i as 1 to number nodes do:
        write "output" [i, 'v' + initial rank, iteration]
else:
    read new rank vector
    for line in new rank value file do:
        i, rank = line.split
        write "output" [i, rank, iteration]
```

The current iteration will be sent in as input. If current iteration is not the first iteration, before reading the edge file, it is necessary to check the new rank vector. If the rank values start with 'f', which means already converged, no need to iterate one more time. Otherwise, it is not converged, new iteration starts. Firstly, send "time" to timer PE to start timer. Secondly, read the edge file, split every line into "src" and "dst", then send "output" to stage one PE. Thirdly, if it is the first iteration, generate initial rank which is 1 divided by number of nodes and send "output" to stage one PE. For other iterations, read the new rank vector, split every line into "i" and "rank", send "output" to stage one PE. There is no need to add 'v' to the new rank value, as 'v' has already been added when the new rank value written to the disk.

- **Stage one:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
src, value, iteration = "input" (grouping=[0])
if value starts with 'v' then:
    cur rank[src] = value removed 'v'
else:
    append value to dst node list[src]
postprocess:
for every src do:
    write "output" [src, 's' + cur rank[src], iteration]
    outdeg = length of dst node list[src]
    current rank = cur rank[src] / outdeg
    for dst in dst node list[src]:
        write "output" [dst, 'v' + current rank, iteration]
```

The input from initial reader will be grouped by “src”. There are two dictionaries “dst node list” which can be looked as a sparse matrix transformed from edge file and “cur rank” used to store all current rank values. Obviously, this PE is quite heavy-loaded: if there are 10000 nodes, which means there will be 10000 lists in “dst node list”. Therefore, stage one should have the priority to get more process to share the pressure of load if there is more process available. If “value” starts with ‘v’, this “value” is an initial rank value, or it is a destination node of “src” which will be appended to “dst node list[src]”. After all input received, for each node, its initial rank value will be sent via “output” to stage two PE and then divided by the length of “dst node list[src]”. For all destination nodes of “src”, send “output” to stage two PE. The ‘s’ and ‘v’ is used to distinguish it is an initial rank value or subrank value.

- **Stage two:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
node, value, iteration = "input" (grouping=[0])
if value starts with 'v' then:
    next rank[node] += value removed 'v'
else:
    previous rank[node] = value removed 's'
postprocess:
    for every node do:
        next rank[node] = next rank[node] * mixing_c + random_coeff
    write "output" [previous rank, next rank, iteration]
```

The grouping mode is same as stage one PE, grouped by the first value which stands

for node. If the value starts with 's', store it in the dictionary "previous rank". Otherwise, add it up to another dictionary "next rank[node]". In the "postprocess" method, every element in dictionary "next rank" will be processed according to the formula: new rank = new rank $\times e + \frac{1-e}{\text{number of nodes}} (e = 0.85)$. Finally, send these two dictionaries via "output" to checker PE.

- **Checker:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
previous rank, next rank, iteration = "input" (grouping='global')
for every node in previous/next rank do:
    p rank[node] = previous rank[node]
    n rank[node] = next rank[node]
postprocess:
changed = 0
for i as 1 to number of node do:
    if abs(p rank[i] - n rank[i]) > threshold then:
        changed += 1
if change ==0 or iteration == max iteration:
    for i as 1 to number of nodes do:
        write [i, 'f' + n rank[i]] to new rank vector
else:
    for i as 1 to number of nodes do:
        write [i, 'v' + n rank[i]] to new rank vector
write "time" ['end', iteration]
```

The grouping mode of input is "global", all input from stage two PE will be collected in one instance of PE. It has two large list "p rank" and "n rank" to store previous and next rank value of each node. After all input received and stored, it is the comparison between previous rank value and next rank value of each node. The threshold of difference depends on the number of nodes, which is $\frac{1}{10 \times \text{number of nodes}}$. In addition, the variable "changed" is defined to count how many nodes are not converged. Thirdly, if "changed" is 0 or this is the final iteration, write every next rank value with its node to the disk, 'f' here means finished. Hence if there is a next iteration, the initializer PE will firstly read the new rank vector and all values in the vector start with 'f'. As a result, the next iteration will be stopped. If "changed" is not 0 and this iteration is not the last one, write every next value started with 'v' to the disk. In the end, send the output "time" to timer PE to stop the timer.

- **Timer:** This PE is implemented as a generic PE and its pseudo code is as follow:

```
operation, iteration = "time" (grouping='global')
if operation == 'start' then: start time = current time
else:  end time = current time
postprocess:
if iteration == 1:
    write (end time - start time) to time file
else:
    read previous time from time file
    total time = end time - start time + previous time
    write total time to time file
print total time
```

  The grouping mode of input is obviously “global”. Once, it receives input from initial reader PE, timer will be started and stopped when receives input from checker PE. Similar to the timer PE in Integer Sort workflow, it will read the previous time cost from time file and then rewrite the sum of previous time cost and current time cost to the time file.

In sum, initial reader, checker and timer PE should be executed by one process and stage one and stage two PE could be executed by multi processes in parallel. Here is a simple example of process distribution of PageRank workflow running with 8 processes:
Timer2: [5]
Check4: [6]
Stage23: [3, 4]
InitialRead0: [7]
Stage11: [0, 1, 2]

Chapter 4

Evaluation

In this chapter, the performance of the original benchmark to the translated workflow will be compared and then analyzing their performance. The details of platforms used in the testing will be introduced in Section 4.1. In Section 4.2, the criteria of evaluation will be illustrated. Section 4.3 presents and analyzes the performance of Integer Sort benchmarks. The following Section 4.4 presents and analyzes the performance of PageRank benchmarks. In the last Section 4.5, the problems met in the project and their solutions will be discussed.

4.1 Platforms

The evaluation takes place in three different platforms:

Linux visual machine (local): the configuration of the visual machine is Intel i5-4200H 2.8GHz, 2 cores, 4GB RAM, 80GB storage and powered by Ubuntu 14.04.

EDDIE cluster: EDDIE cluster is constructed by relatively standard machines, which run an industry standard Linux based operating system. The node used for testing has 8 cores which consists of two Intel Xeon E5620 quad-core processors with 2GB RAM each core by default. All these nodes are connected by a GigabitEthernet network with a 10 Gigabit network core.

DIR cluster: DIR machine is an Open Nebula linux cloud designed for data-intensive workloads. Backend nodes use mini ITX motherboards with low powered Intel Atom processors with plenty of space for hard disks. Each VM in our cluster had 4 virtual cores - using the processor's hyperthreading mode, 3GB of RAM and 2.1TB of disk space on 3 local disks. We used 14 VMs for our evaluations, where MPI, Hadoop, and dispel4py have been installed.

4.2 Criteria of evaluation:

For same processes and size of data, the test will be run 7 times. However, when calculating the average running time, the maximum and minimum running time are excluded. Running time is a practical standard to measure the efficiency of a benchmark.

The test data of Integer Sort consists of 4 classes of problem size, which are S (2^16 integers, about 390KB, maximum key is 2^11, 512 buckets in each process, used for local test), W (2^20 integers, about 6MB, maximum key is 2^16, 1024 buckets in each process, used for cluster test), A (2^23 integers, about 60MB, maximum key is 2^19, 1024 buckets in each process, used for cluster test), B (2^25 integers, about 250MB, maximum key is 2^21, 1024 buckets in each process, used for cluster test).

The translated Integer Sort workflow needs at least 7 nodes to be executed, so the test will start from 8 processes. Locally, the test ends at 128 processes and in the cluster ends at 32 processes.

The test data for PageRank benchmark could be defined manually. In the local test, the numbers of nodes are: 50 (10KB), 500 (150KB), 5000 (1.7MB), while in the DIR cluster, the numbers of nodes are: 5000 (1.7MB), 10000 (3.5MB), 50000 (21MB).

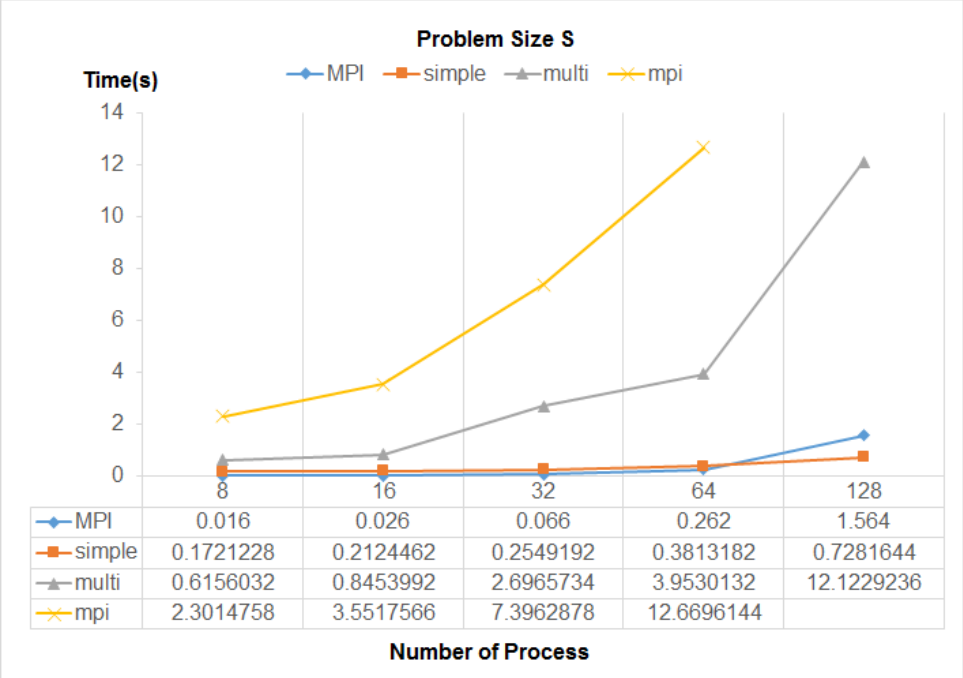
The translated PageRank workflow needs at least 5 nodes to be executed. In local, it will be run by 8 processes or 4 mapper + 4 reducers. In the DIR cluster, these two benchmarks will be run with 6 and 8 processes as a few nodes which have Hadoop installed were unconnected at that moment.

4.3 Integer Sort benchmark

4.3.1 Local performance

Problem size S (390KB) is used in the local test and the translated workflow is executed by 3 mappings: simple, multi and MPI. Storm mapping is unavailable in the EDDIE cluster, so it is excluded in the local test.

The Figure 4.1 shows the average running times of problem size S with 8/16/32/64/128 processes. For Integer Sort benchmark, the upper case “MPI” in figures stands for the original MPI benchmark and lower case “mpi” stands for dispel4py MPI mapping. Unfortunately, the visual machine is unable to run the workflow with 128 processes by MPI mapping.



	8	16	32	64	128
MPI	0.016	0.026	0.066	0.262	1.564
simple	0.1721228	0.2124462	0.2549192	0.3813182	0.7281644
multi	0.6156032	0.8453992	2.6965734	3.9530132	12.1229236
mpi	2.3014758	3.5517566	7.3962878	12.6696144	

Figure 4.1: Problem size S with 8/16/32/64/128 processes.

From the Figure 4.1, it is obvious that when executed by single processor, the running time increases as the number of processes increases. This is because all processes needed are spawned by the only processor. There are two interest points in the graph: 1. the running time of simple mapping increases slower than the original MPI, which leads to that when running with 128 processes, the time costs of simple mapping is less than original MPI; 2. the running time of MPI mapping is about twice more than multi mapping. Therefore, although the visual machine is unable to run the workflow with 128 processes by MPI mapping, it is possible to predict the time cost could be about 36 seconds.

4.3.2 Cluster performance

Problem size W (6MB), A (60MB) and B(250MB) are used in the cluster test and the translated workflow is executed by MPI and multi mapping. The Figure 4.2 shows the average time cost of problem size of W with 8/16/32 processes. Unfortunately, in the EDDIE cluster, the translated Integer Sort workflow only can be run with 8/16 processes with MPI mapping and 8/16/32 processes with multi mapping. Once the number of processes is over 32 (for MPI mapping) or 64 (for multi mapping), it is not runnable. MPI mapping will be stopped by an exception “Connection refused” and for multi mapping, there is only processes distribution, no execution.

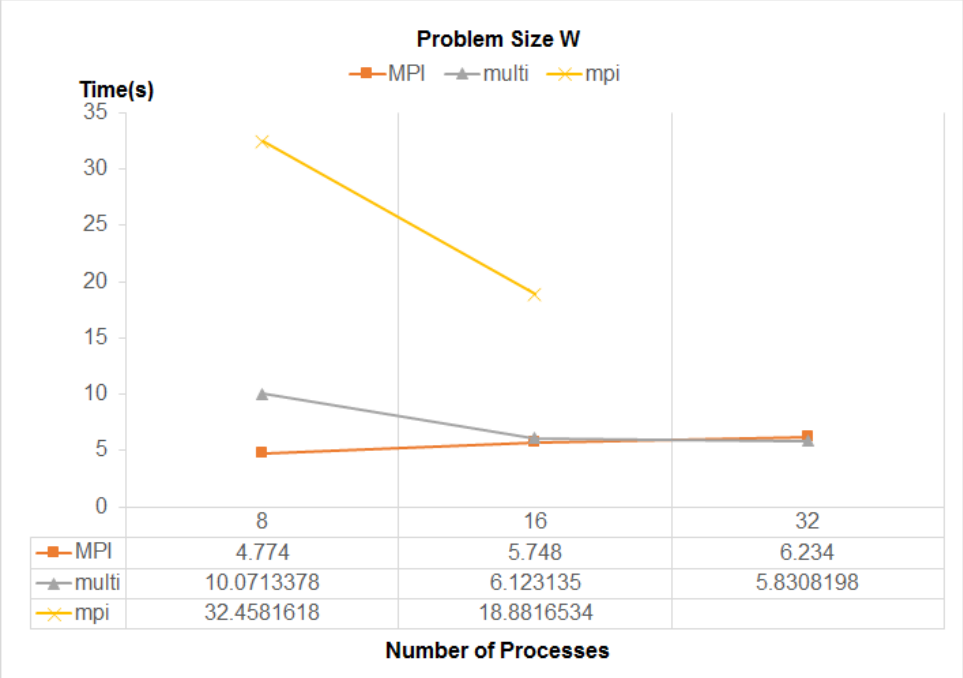


Figure 4.2: Problem size W with 8/16/32 processes.

From the Figure 4.2, as the number of processes increases, the time cost of MPI increases as well. This may be caused by the processing time is less than communication time. Therefore, as the number of processes increases, the communication time increases more than the processing time decreases, which makes the total running time increases. In respect of translated workflow, the running time decreases as the number of processes increase. Further, comparing the running time of MPI and multi mapping with 8/16 processes, it still follows the rule in the local test: the time cost of MPI mapping is about twice more than multi mapping. As a prediction, the time cost of MPI mapping with 32 processes could be about 17 seconds. There is a remarkable point that, with 32 processes, the multi mapping is ever more efficient than the original MPI.

The Figure 4.3 shows the average time cost of problem size A with 8/16/32 processes.

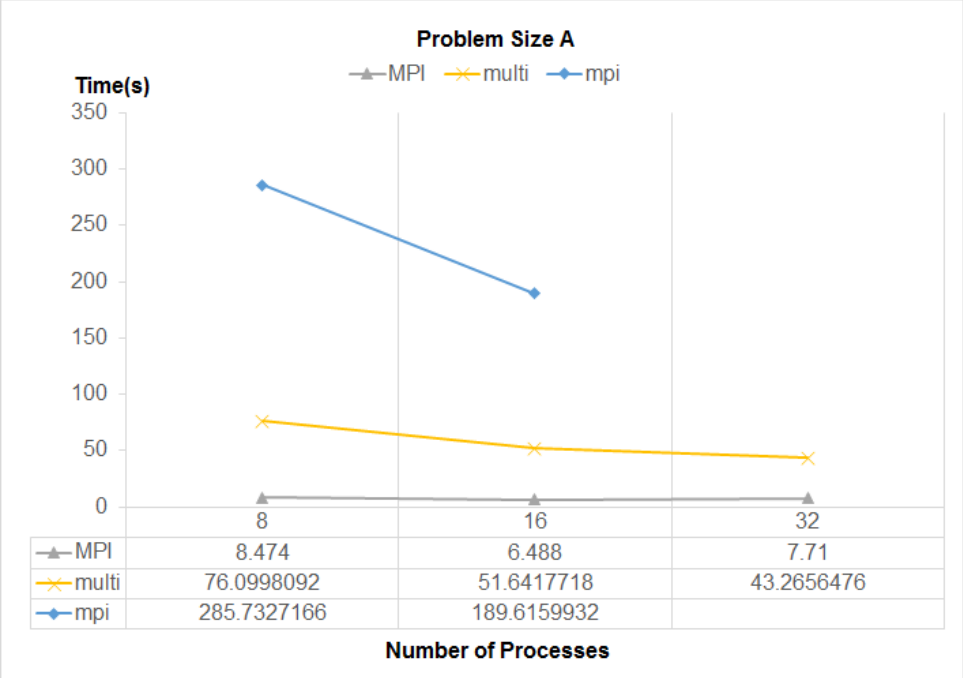


Figure 4.3: Problem size A with 8/16/32 processes.

From the Figure 4.3, the gap of performance begins to emerge. The time costs of MPI is still persisting under 10 seconds, while the time cost of translated workflow has arisen a lot. In addition, the time cost of MPI mapping is twice more than multi mapping as before. The Figure 4.4 shows the average time cost of problem size B with 8/16/32 processes.

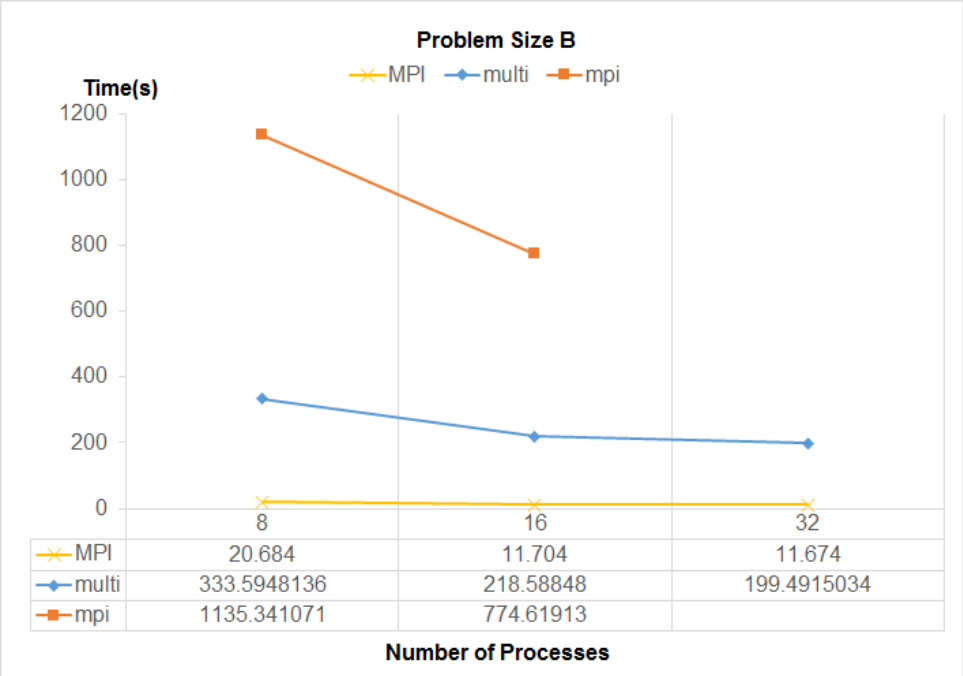
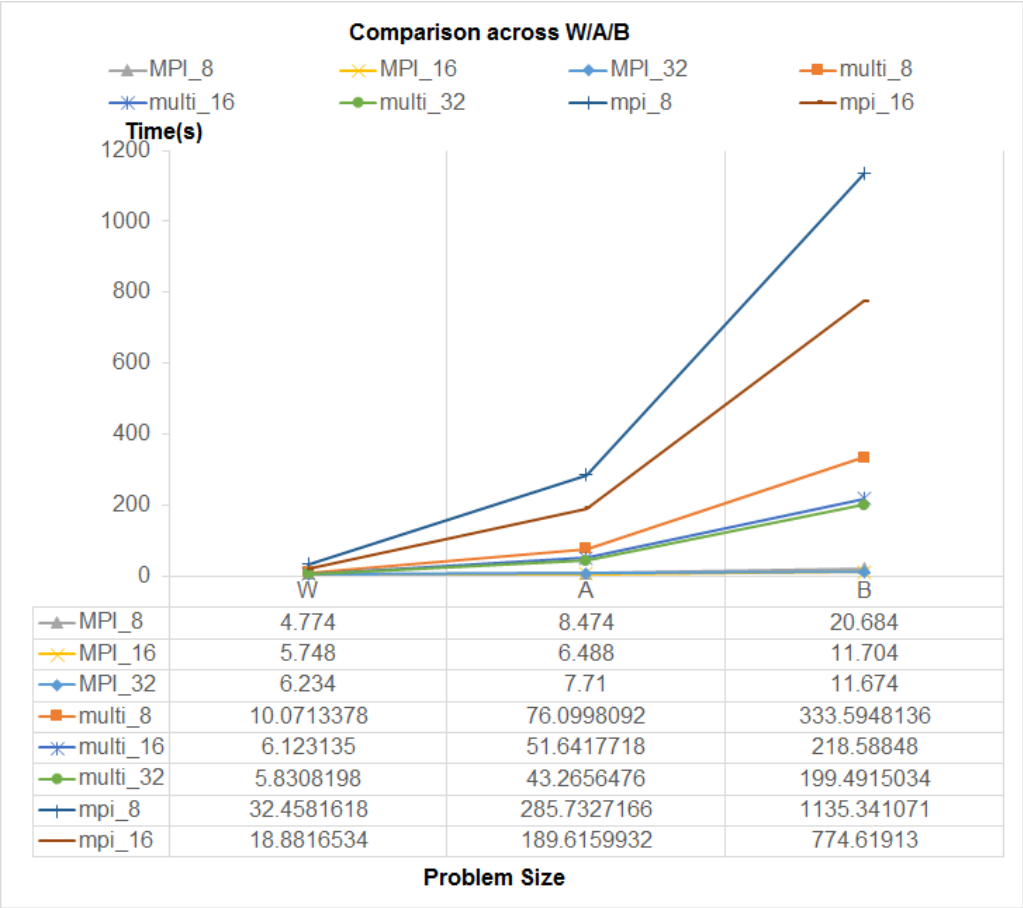


Figure 4.4: Problem size B with 8/16/32 processes.

From the Figure 4.4, the gap of performance becomes much bigger than that in problem size A. The running time of MPI mapping even overpasses 1000 seconds, while the original MPI is about 20 seconds. Basically, the time cost of MPI mapping is still twice more than multi mapping. The Figure 4.5 shows the comparison across problem size W/A/B



	W	A	B
MPI_8	4.774	8.474	20.684
MPI_16	5.748	6.488	11.704
MPI_32	6.234	7.71	11.674
multi_8	10.0713378	76.0998092	333.5948136
multi_16	6.123135	51.6417718	218.58848
multi_32	5.8308198	43.2656476	199.4915034
mpi_8	32.4581618	285.7327166	1135.341071
mpi_16	18.8816534	189.6159932	774.61913

Figure 4.5: Comparison across problem size W/A/B.

Problem size A is 8 times as big as W, and B is 4 times as big as A. The Figure 4.5 shows that both of MPI and multi mapping, the increase of running time is proportional to the increase of data size. Moreover, when the data size is relatively small as problem size W (6MB), the performance of dispel4py could be even better than original MPI.

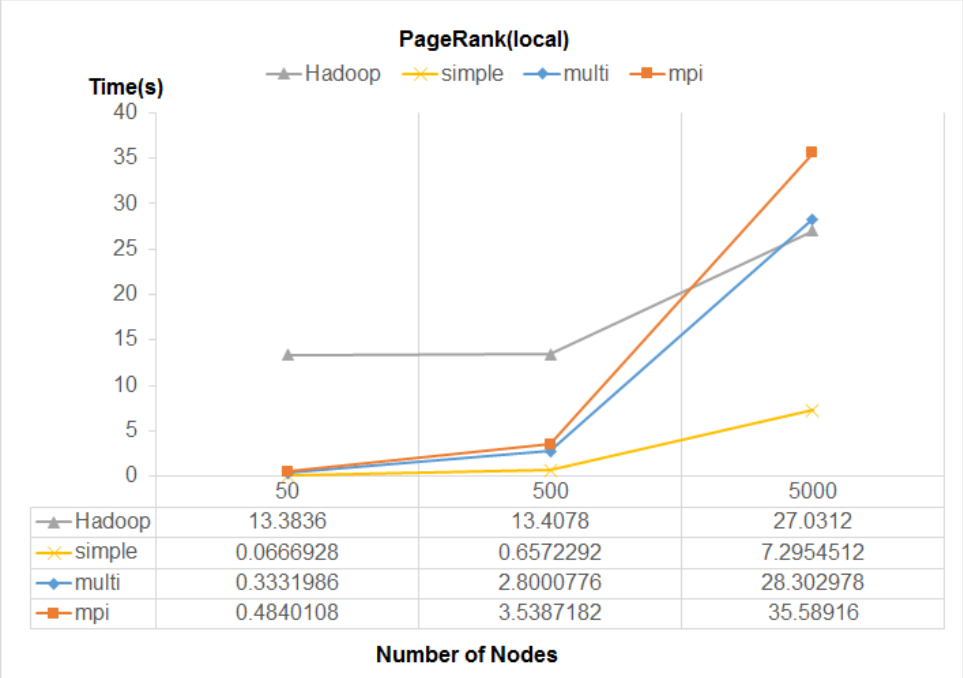
In summary, the performance of dispel4py workflow is comparable to or even better than MPI with relatively small problem size, e.g. problem size W. However, as the size of input data increases, the running time of dispel4py workflow increases proportionally. In addition, although the workflow cannot be run with MPI mapping with 32 processes, according to the relation between MPI mapping and multi mapping in the Integer Sort workflow, the running time of MPI mapping with 32 processes could be predicted as 3 times of running

time of multi mapping with 32 processes.

4.4 PageRank benchmark

4.4.1 Local performance

In the local test, the translated workflow will be executed by 3 mappings as the local test in Integer Sort: simple, multi and MPI. The Figure 4.6 shows the local performance of two benchmarks. Hadoop stands for the original PageRank benchmark.



	50	500	5000
Hadoop	13.3836	13.4078	27.0312
simple	0.0666928	0.6572292	7.2954512
multi	0.3331986	2.8000776	28.302978
mpi	0.4840108	3.5387182	35.58916

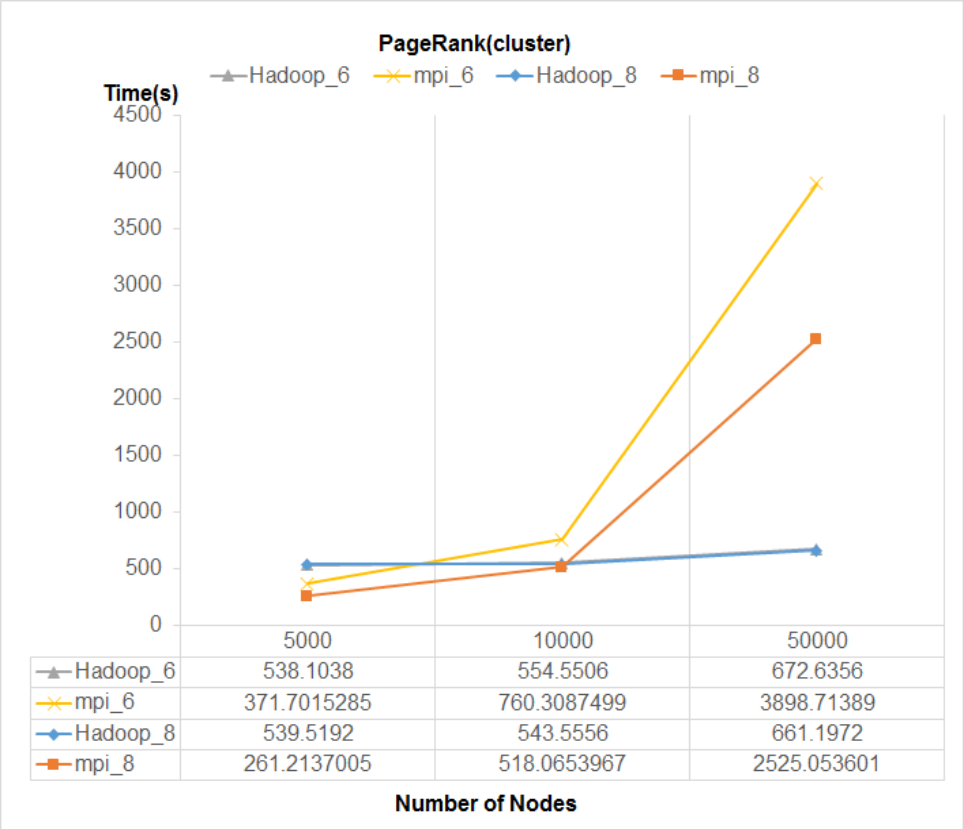
Figure 4.6: Local PageRank performance.

From the Figure 4.6, it is obvious that the translated dispel4py workflow mapped by simple works better than original one when data size is relatively small. However, what cannot be ignored is, the increase of time cost is still proportional to the increase of data size for dispel4py workflow. In other words, once the data size is 10 times as big as before, the time cost is also 10 times as big as before.

4.4.2 Cluster performance

In the cluster test, the multi nodes available in the DIR cluster is 4, while the minimum nodes needed for PageRank benchmark is 5. As a result, the cluster test is only run by MPI

mapping. The Figure 4.7 shows the performance of two benchmarks in the cluster with 6 and 8 processes. 6 processes are equally divided into 3 mappers and 3 reducers and 8 processes are divided in the same way.



	5000	10000	50000
Hadoop_6	538.1038	554.5506	672.6356
mpi_6	371.7015285	760.3087499	3898.71389
Hadoop_8	539.5192	543.5556	661.1972
mpi_8	261.2137005	518.0653967	2525.053601

Figure 4.7: PageRank performance in the cluster.

From the Figure 4.7, the running time of two benchmarks decreases as the number of processes available increases. The running time of translated workflow is initially less than original Hadoop one. The turning point is 10000 (3.5MB) nodes, where the running time of translated workflow begins to be more than original Hadoop one. When the number of nodes reaches 50000 (21MB), the average running time of translated workflow has arisen to over 1 hour with 6 processes and about 40 minutes with 8 processes. This proves the rule once again that for translated workflow, the increase of running time is proportional to the increase of data size.

Similar to the result of Integer Sort benchmark, the dispel4py workflow is comparable to or even better than Hadoop MapReduce with relatively small data size, like 50, 500, 5000 nodes. Moreover, the increase of running time is proportional to the increase of data size as well.

4.5 Problems and solutions

In this section, the problems met in the procedure of design, implementation and testing will be discussed as well as their solutions.

4.5.1 Global grouping

As we explained before in Section 2.2.1, all-to-one grouping is mainly used for data aggregation .This problem firstly happened in the bucket size aggregator PE of Integer Sort workflow. When this PE is executed by one process, everything is fine. However, as the number of available processes increases, the framework will distribute more processes to each PE except the root PE. Although the global grouping is able to gather all input in one instance of PE, the other "idle" instances of PE are active, which means each "idle" instance of PE will also output a "bucket size totals" list. Because those "idle" instances of PE do not have any input data, their "bucket size totals" lists actually are lists of 0, which will make fatal mistakes in the workflow.
Therefore, the solution of this problem is, when the number of available processes increases, reset the number of executing processes of each PE manually and strictly. Moreover, there is another benefit of manual setting, which is make full use of processes available.

4.5.2 Iterations

The "-i n" argument which has been explained in Section 2.2.2 in the command line is actually intended to iterate the root PE for n times. For instance, using the "-i 3" to execute the PageRank workflow, the initial reader will read the edge file for 3 times and create 3 initial rank vectors, which is totally different from the original benchmark. This also will lead to data from different iterations mixed up.
There are two solutions for this problem. Option one is add one more dimension to every data container in the each PE to indicate which iteration the data belongs to. However, this is a bad solution, as the memory will be easily run out if the number of iteration is large. Option two is better, use a loop in shell script and every time send the current iteration to the root PE as input.

4.5.3 Different versions of MPI compiler

This problem could be divided into two problems and mainly for the Integer Sort workflow. First problem is, for the original benchmark, the executable made by local MPI compiler

cannot be run in the EDDIE cluster because of the versions of MPI compiler are different. Second problem is, for the translated workflow, the MPI implementation of anaconda in the cluster is MPICH2, which is incompatible with OpenMPI.
The solution for the first problem is compiling the source code by the MPI compiler in the cluster although the MPI implementation in local and in the EDDIE cluster both are Open-MPI. For the second problem, the solution is just abandoning the OpenMPI, although the parallel environment in the EDDIE cluster is OpenMPI, MPICH2 is still runnable.

4.5.4 Bug in the data generator

This problem is caused by the PageRank data generator of the original Hadoop benchmark. As only Hadoop version of PageRank benchmark is needed, HiBench v2.2 has been choosen instead of v4.0. However, unfortunately, in the release of v2.2, there is a bug remains in the PagaRank data generator. The error data generated by this generator leads to, regardless of number of nodes, the benchmark only iterates once.
The solution is quite simple, just copy the java file of PageRank data generator from the v4.0 latest release and rebuild it. It costs more time to find the bug than fixing it, as it is a bug in the original benchmark.

4.5.5 DIR is a “real” distributed memory system

The DIR cluster is used to test the performance of PageRank workflow. However DIR is a little different from EDDIE cluster: the nodes in DIR are able to store data locally. In this case, the edge file, rank vector and time file will be stored separately: edge file is in the node of initial reader PE, rank vector is in the node of checker PE and time file is in the node of timer PE. Unfortunately, the distribution of nodes will change in every iteration, for example, the node 1 is used by checker PE in the first iteration and used by timer PE in the second iteration. The timer PE in the second iteration will throw an exception because there is no timer file locally. The solution of this problem is synchronization: Before running the PageRank workflow, firstly send a copy of edge file to each node; Secondly, remove all rank vectors and time files in each node; Thirdly, after every iteration, pull the rank vector and time file from each node back to the master node, in fact, there is only one rank vector and one time file; Fourthly, synchronize the rank vector and time file to every node. The time cost of synchronization is not included in the running time cost.

Chapter 5

Conclusion

In summary, according to the performance test result of Integer Sort and PageRank benchmarks and the experience of developing with dispel4py, some conclusions could be made as follow:

1. The dispel4py workflow is comparable to or ever better than MPI or Hadoop MapReduce when the size of input data is less than about 10MB. Therefore, if the data unit in data stream is around or less than 10MB, dispel4py could be a desirable option to handle the data stream.

2. The running time of dispel4py workflow will increase proportionally as the size of input data increases and remarkably decrease as the number of processes available increases (in the cluster).

3. In general, the multi mapping works better than MPI mapping either in local or in the cluster.

4. The dispel4py framework is really scalable, as the python script developed in laptop can be run in a cluster without any modification. The rare few things which need to be modified are the path of input and output, the size of input data and the running shell.

However, in the procedure of using dispel4py, some drawbacks of dispel4py also have exposed:

1. There is no support for parallel input in dispel4py contemporarily, while MapReduce could read data parallel by multi mappers and MPI could generate data parallel in all processes. The root PE in the workflow is limited to be executed by 1 process. This could be a bottleneck for dispel4py especially when the size of input data is hundreds of MB.

2. The time of communication between different PEs should be limited. For example, when testing translated PageRank workflow in the DIR cluster with the data size of 50000 nodes, the time of processing iterations of stage one PE or stage two PE is counted by tens of millions. This could easily lead to running out of memory, which will stop the execution and throw an exception “connection refused” or “connection reset by peer”. On the other hand, the delay of enormous times of communication and pressure of buffering is unneglectable, which could significantly take negative effect to the performance of dispel4py.

Therefore, in the future, the performance of dispel4py could be improved in some methods:

1. Add the support for parallel input. This is really important for a parallel processing framework to improve its data throughput.

2. Make the partition mechanism cleverer. The PEs with high data throughput but low CPU and RAM demand could be wrapped and executed in one process automatically. This is also a method to reduce the inter-process communications between PEs.

3. Compress the size of data in the communication, which could reduce the time cost within communication and memory cost of buffering.

4. For programmer, it would be better to make full use of local memory to reduce the inter-process communications, for example, combine data locally before sending it.

Bibliography

Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., and Mock, S. (2004). Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE.

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. (1991). The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73.

Dagum, L. and Enon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Deelman, E., Gannon, D., Shields, M., and Taylor, I. (2009). Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540.

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., da Silva, R. F., Livny, M., et al. (2015). Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35.

dispel4py org (2015). Enactment of dispel4py workflows. *Available at: http://dispel4py.org/documentation/enactment.html [Accessed at 19 August 2015]*.

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

Forum, M. (2015). A message-passing interface standard version 3.1. *Available at: http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf [Accessed at 19 August 2015]*.

Fowler, M. (2004). *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al. (2004). Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer.

Google (2015). How search works. *Available at: http://www.google.com/ insidesearch/howsearchworks/thestory/ [Accessed at 19 August 2015]*.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828.

Grün, T. and Hillebrand, M. A. (1998). Nas integer sort on multi-threaded shared memory machines. In *Euro-Par'98 Parallel Processing*, pages 999–1009. Springer.

Hey, A. J., Tansley, S., Tolle, K. M., et al. (2009). *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft Research Redmond, WA.

Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. (2010). The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE.

Koepke, H. (2010). reasons python rocks for research (and a few reasons it doesn't), 2010. *URL http://www. stat. washington. edu/˜ hoytak/blog/whypython. html*.

Lämmel, R. (2008). Google's mapreduce programming modelâĂŤrevisited. *Science of computer programming*, 70(1):1–30.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: bringing order to the web.

Peterson, J. L. (1981). Petri net theory and the modeling of systems.

Rabenseifner, R., Hager, G., and Jost, G. (2009). Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE.

Storm, A. (2015). Storm, distributed and fault-tolerant realtime computation. *URL http://storm.apache.org/ [Accessed at 19 August 2015]*.

White, S. A. (2008). *BPMN modeling and reference guide: understanding and using BPMN*. Future Strategies Inc.

Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., et al. (2013). The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, page gkt328.