Evaluation of mpi4py for Natural Language Processing Scenarios

---

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Manvi Saxena

May, 2018

Evaluation of mpi4py for Natural Language Processing Scenarios

_____

Manvi Saxena

APPROVED:

_____

Dr. Edgar Gabriel, Chairman
Dept. of Computer Science, University of Houston

_____

Dr. Thamar Solorio
Dept. of Computer Science, University of Houston

_____

Dr. Peggy Lindner
The Honors College, UH

_____

Dean, College of Natural Sciences and Mathematics

Acknowledgements

First of all, I would like to thank my advisor without whom this thesis would not have been possible. I thank him for his encouraging words, endless support, and long debugging sessions. Thank you, Dr. Gabriel.

I thank Dr. Solorio and Dr. Lindner for extending their help in completing my thesis.

Last but not the least, I want to thank my family and all of my friends for being there for me all the time. Thank you everyone.

Evaluation of mpi4py for Natural Language Processing Scenarios

---

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Manvi Saxena

May, 2018

Abstract

Many Natural Language Processing (NLP) applications operating on large data sets are written in programming languages that do not have bindings in the Message Passing Interface (MPI) specification. Yet, with increasing problem sizes, these applications also necessitate some form of parallel and distributed processing. The goal of this thesis is to evaluate the utilization of MPI with a non-traditional HPC programing language, Python, for NLP application scenarios.

The current thesis is divided into two parts. The first part evaluates the performance and functionality of the mpi4py, a python module for MPI binding, using multiple point-to-point benchmarks with native C-based MPI benchmarks using an InfiniBand and a Gigabit Ethernet network interconnect. The results show that in many instances communication performance of the Python benchmarks was on par with their C-based counterparts.

In the second part of the thesis, a few application scenarios used in Natural Language Processing (NLP) such as word count, n-gram count, and tfidf were developed, and mpi4py module was used to distribute data on different nodes for these scenarios and to evaluate performance. The results demonstrate that the application of mpi4py module in NLP scenarios can greatly improve execution time.

Contents

List of Figures

Chapter 1

Introduction

Parallel computing is a computing method in which multiple processors are concurrently used to solve a problem. In traditional (serial) programming, a single processor executes program instructions in a step-by-step manner. Some operations, however, have multiple steps that do not have time dependencies and therefore can be separated into multiple tasks to be executed simultaneously. For example, adding a number to all the elements of a matrix does not require that the result obtained from summing one element be acquired before summing the next element. Elements in the matrix can be made available to several processors, and the sums performed simultaneously, with the results available faster than if all operations had been performed serially.

Parallel computations can be performed on shared-memory systems with multiple central processing units (CPUs), distributed-memory clusters made up of smaller shared-memory systems, or single CPU systems. Coordinating the concurrent work of the multiple processors and synchronizing the results are handled by program calls to parallel libraries; these tasks usually require parallel programming expertise.

Parallel computers can be broadly classified into three types by the way they are connected through memory system or through network interconnect. They are as follows:

- Shared memory systems: Memory can be accessed simultaneously by number of processes for communication. Fig. 1.1 shows a schematic of shared memory system.

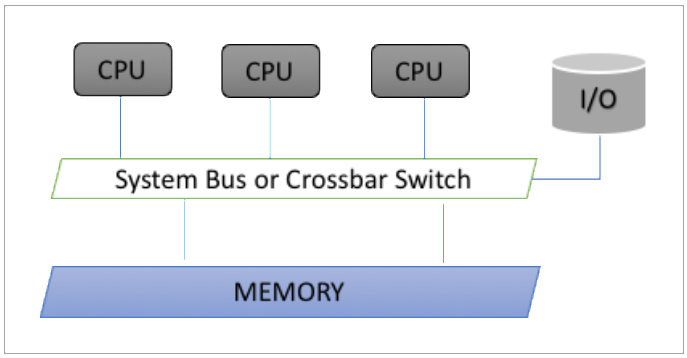

Figure 1.1 Shared memory system for three processors

- Distributed memory systems: Memory system where each processor has its own private memory space. For communication, task must request and receive data from respective processors. A schematic of distributed memory stems shown in Fig. 1.2.

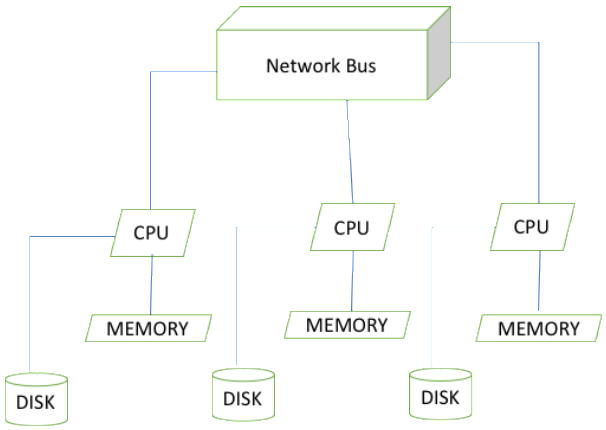

Figure 1.2 Distributed memory system of three computers

- Hybrid systems: Memory architecture where shared memory systems are connected via network, Fig. 1.3.

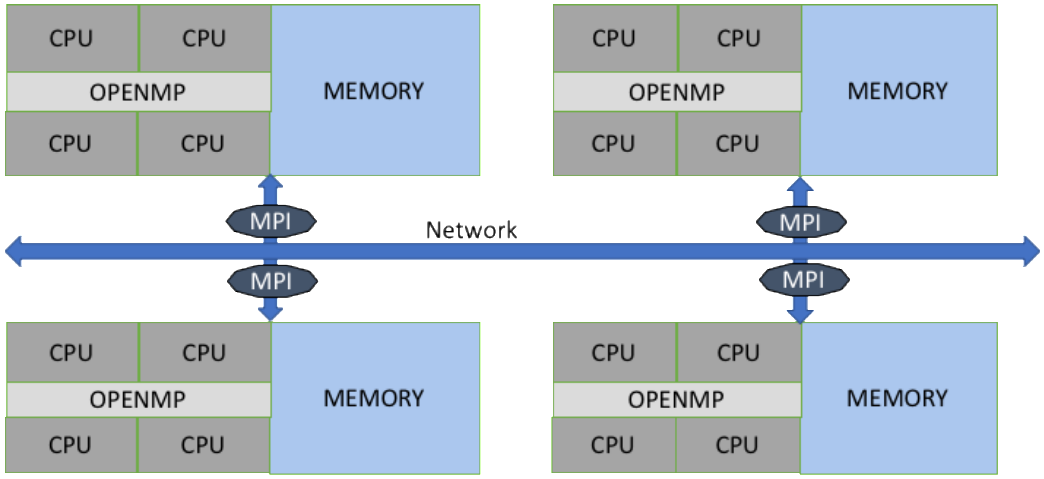

Figure 1.3 hybrid memory for quad-core processor-based computer system

The uses of parallel computing are very wide ranging from home gaming system to national astronomical observatory for the purpose of collision risk assessment [1,2,3,4]. Other areas where parallel computing is currently used are:

- Weather prediction
- Oceanography and astrophysics
- Artificial intelligence and automation
- Seismic exploration
- Genetic engineering
- Weapons research and defense
- Medical applications
- Energy resource exploration

1.1. Big Data and NLP

Big data is a term coined for solving problems that require analyzing very large amounts of un-structured data. In contrast, the High Performance Computing (HPC) deals with small size dataset, which are structured and mainly used for computational purposes. Big data stands for data of very large volume, velocity, variety (of different data types), veracity (trustworthiness of data), and value (meaningfulness) of data. Many technologies have been developed to store, query and manage large sets of data such as relational database, data warehouse, and parallel databases [14]. But, due to exponential growth of data, big data technologies are now using cluster computing instead of single node computing.

Apache Hadoop [13] is an example of big data technology. It is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs.

Apache Spark [15] is a more generic model of cluster computing, it provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance. It is capable of streaming, batch and interactive workloads.

Natural Language Processing

Natural Language Processing (NLP) is the field of science which deals with interactions between human language and computers. It is based on study of mathematical and computational modeling of various aspects of language and the development of a wide range of systems. These include spoken language systems that integrate speech and natural language; cooperative interfaces to databases and knowledge bases that model aspects of human-human interaction; multilingual interfaces; machine translation; and message understanding systems [16]. It includes areas such as artificial intelligence, computer science, and computational linguistics. This field covers computer understanding and manipulation of human language. It provides ways for computers to analyze, understand, and derive meaning from human language. It deals with following programming functionalities that can process large natural language corpora [17]:

- Understanding natural language to decipher its meaning,
- Generating natural language to present textual, audio or video,
- Provides connecting link between language and machine perception, and
- Facilitate dialog between human and machine.

Information retrieval (IR) is required to implement any of above mentioned features. IR deals with the process of automated searching of information from provided resources. IR represent document with following models:

- **Set-theoretic** models represent documents as sets of words or phrases. Similarities are usually derived from set-theoretic operations on those sets.

- **Algebraic models** represent documents and queries usually as vectors, matrices, or tuples. The similarity of the query vector and document vector is represented as a scalar value.
- **Probabilistic models** treat the process of document retrieval as a probabilistic inference. Similarities are computed as probabilities that a document is relevant for a given query.
- **Feature-based retrieval models** view documents as vectors of values of feature *functions* (or just *features*) and seek the best way to combine these features into a single relevance score, typically by learning to rank methods.

The current thesis will use algebraic model (Vector Space model) and feature based retrieval models.

1.2. Programming Models

A programming model is an abstraction of the underlying computer system that allows for the expression of both algorithms and data structures [5]. They allow programmers to develop codes which are faster, efficient and portable. In High Performance Computing (HPC), users focus on writing computation intensive programs for parallel processing, and in big data programming users focus on writing data-driven parallel programs that can be executed in parallel and distributed environments. The programming models used in HPC and big data are briefly described below.

1.2.1 High Performance computing (HPC)

Message Passing Interface (MPI) is the standard communication protocol for parallel programming. It is used for communication by passing messages to different process in distributed system. It provides a specification for how its features must behave in any implementation [7].

MPI includes both point-to-point and collective operations. In point to point operations, communication refers to transmittance of a message or data between a pair of processes. Whereas, collective communication refers to communication with participation of all the processes within the communicator. It involves group of processes to send and receive message.

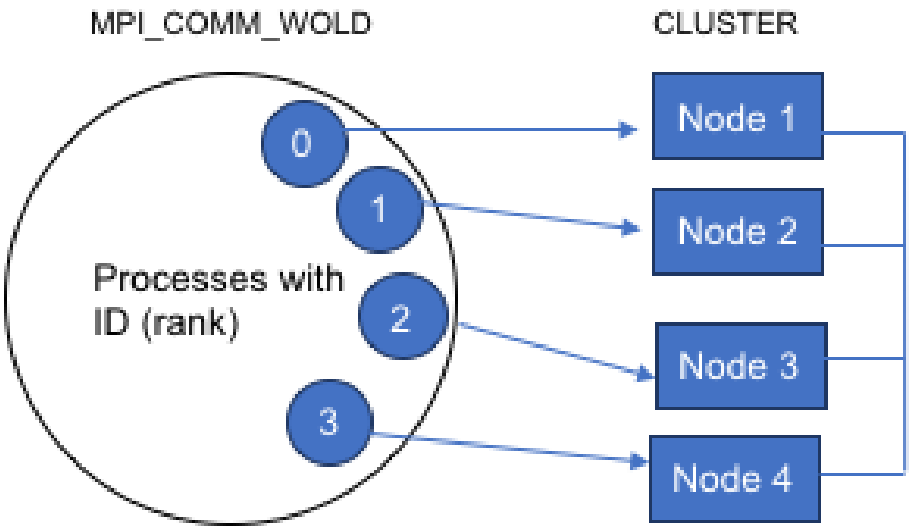

Figure 1.4 Schematic of MPI communication in a cluster of four nodes

Open Multiprocessing (OpenMP) is a directive based multi-processing programming model which works on fork and join execution model of multi-threading. It targets shared memory architecture, where all processors share main memory. It launches a single process which in turn can create n number of thread as directed by user [9].

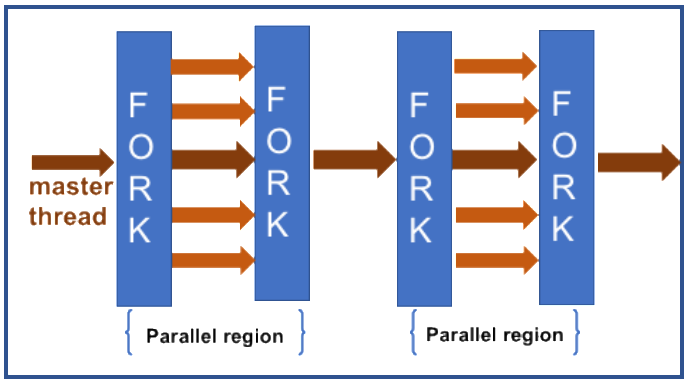

Figure 1.5 Schematic of Fork Join Model of OpenMP

Compute Unified Device Architecture (CUDA) is a parallel programming architecture majorly used in Graphics Processing Units (GPU). It was created by Nvidia. In CUDA, parallel part of program is executed on GPUs [11]. It supports programming languages such as C, C++ and Fortran. OpenACC [18] and OpenCL [19] are frameworks used for heterogeneous programming using CPUs and GPUs. It enables general purpose computing on graphical processing units which is termed as General Purpose Graphical Processing Unit (GPGPU).

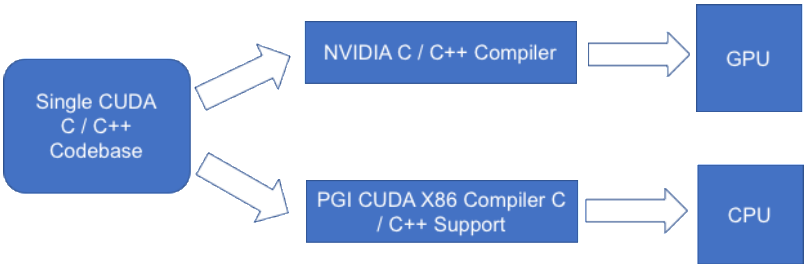

Figure 1.6 Schematic of Computer Unified Device Architecture (CUDA)

Other programming models include partitioned global address space (PGAS) as Co-Array Fortran (CAF), Unified Parallel C, and the DARPA-initiated, vendor-based, languages Chapel, Fortress and X10 [5].

1.2.2 Big Data Technology

MapReduce [6] is a framework to write parallel application for big data application. It can be termed as de-facto paradigm for writing applications that can be solved with map and reduce functions. It uses divide and conquer approach to divide data, perform operation with map function and then reduce it to get result.

Hadoop is one of the famous examples of this paradigm. It is written in Java, and therefore its map and reduce functions are provided with Mapper and Reducer interfaces. The input to Mapper is key-value pair, map function performs logic on each key-value pair. Reducer perform logic on set of values for each key.

Functional Programming is a declarative style of programming. It describes the logic of computation in form of expressions and therefore avoids any side effect. It is becoming popular choice for next generation big data processing technology. Spark and Flink are few popular examples of functional programming.

Spark [20] framework provides data centric programming interface on its resilient distributed dataset (RDD) structure. Tasks can be represented in parallel as RDD

transformations. The programs are dependency flows which will be executed in parallel on distributed environment.

Flink [21] is an open source framework for distributed and high performing data streaming applications. It provides bounded and unbounded data sets. Bounded datasets are finite and unchanging, while unbounded data sets are infinite datasets that are appended to continuously. It provides both streaming and batch processing execution model. It provides real time (record level) streaming as compare to micro batch streaming in spark.

Structured query language is the most famous database query language, based on relational algebra. It consists of data definition, manipulation and control features. It supports primitives such as create, insert, update and delete for modifying data in tables.

- **Hive** [22] is a SQL-like query interface built on Hadoop environment. It takes SQL like statement for querying and internally convert it into MapReduce jobs connected as a Directed Acyclic graph.

- **Cassandra** [23] is a distributed NoSQL database, which supports SQL like statements to query data from cluster of computers. Key features of Cassandra are decentralized, scalability, replication factor, fault tolerant, tunable consistency and map reduce support. It uses KEYSPACE and COLUMNFAMILY compared to DATABASE and TABLE in SQL.

- **Apache SQL** [24] is a Spark module for querying structured data. It is built on the DataFrame model and consider input data sets as table like structure. It provides SQL like statement to query data from data frames.

Actor Model [25] is a programming model for concurrent computation. The primitive unit of computation in this model is actor. Actors have states and they communicate with messages and perform action or computation based on that message. Actors are completely isolated and never share memory. They have private state and address where they send and receive messages. Actor model is a reactive programming model, where programmers write code for reacting logic in response to events and state changes [12].

- **Akka** [26] is actor-based toolkit and runtime for concurrent and distributed applications on Java virtual machine. It is written in Scala. Concurrency is message based, asynchronous and no mutable data is shared. Apache Spark is built on Akka.

- **Storm** [27] is open source distributed real time computation system. It supports real time processing of unbounded streams of data. Based on Actor based model, it provides two types of processing actors.

  (i) Spouts: It is data source of stream and is continuously collecting or generating data for processing.

  (ii) Bolts: It is processing entity which contains processing logic such as transformation, redirection, aggregation, partitioning etc.

1.3. Challenges

1.3.1. Motivation for using Python MPI

Python is a high-level programming language which contains built-in libraries, for example natural language processing toolkit (NLTK), scikit-learn, numpy, pandas, matplot and pybrain suitable for programming language used in data science applications. It is primarily used in data analytical application. Goal of this thesis was to evaluate possibility of getting high performance from NLP based applications. Therefore, the primary choice of language was the one which is best for NLP application, and which provides binding for MPI.

For MPI library in python, there are many options such as - pypar [15], MYMPI [10], MPI Python [1], and mpi4py [5, 6]. In the current thesis it was decided to investigate the mpi4py library, since it is the most complete, widely used, and actively developed library among the available solutions.

1.3.2 Benefits of using big data applications in HPC setting

With rise in real time analytics for operational, tactical and strategic decisions, there is a high demand for obtaining the best possible performance from such applications. For real time decision making, once the data is arrived, the performance of application is most often the limiting factor. In industries like finance and trading data-based decisions must be made quickly, in real time, therefore performance of the applications is very critical.

In this thesis, it is aimed to get high performance for such application.

1.4. Goal of the Thesis

For the last few years big data and high-performance computing (HPC) communities have been working on similar problems and have developed solutions based on fundamentally different technologies. The dominant programing models in big data are based on the Map Reduce paradigm, although more generic models such as the one offered by Apache Spark are becoming popular. Many of the applications developed for big data environments are based on Java, Scala and Python programming languages, whereas software in the HPC are primarily developed in programming languages such as MPI, Open MP, or CUDA. C, C++ and FORTRAN.

It has been said [13] that most data science applications would benefit from utilizing an HPC programming model and environment from the performance perspective over the big data approaches such as Hadoop MapReduce or Apache Spark. Yet, it is often unrealistic to expect application groups to convert their codes to an HPC environment for practical reasons. For example, applications in the domain of natural language processing (NLP) are often written in Python, and benefit from having a wide range of well tested tools and libraries that they are built on. Rewriting an application code in this domain using C/C++ would not just mandate converting the code written by the application group from Python to C/C++, but also replacing all the libraries used by the original code.

The goal of the current thesis are two folds: (i) evaluate the MPI with a non-traditional HPC programming language (mpi4py for Python) in terms of their latency and bandwidth usages as compared to their C counterpart, and (ii) evaluate mpi4py in natural language processing scenarios from performance and functionality perspectives.

The organization of the remainder of the thesis is following: Chapter 2 will provide a brief background on Python, MPI, mpi4py, NLTK and other python libraries relevant to current work. Chapter 3 will discuss the results related to the first goal of the thesis, i.e., evaluation of the mpi4py- MPI for python in terms of their latency and bandwidth usages as compared to their C-counterpart. Chapter 4 will present the results related to the second goal of the thesis, i.e. evaluation of mpi4py for natural languages processing scenarios such as word count, n-gram, term frequency inverse document frequency (tfidf), and lastly Chapter 5 will summarize the thesis and outline the scope for future work.

Chapter 2

Background

2.1. Python

Python is a high level, general purpose programming language originally designed by Guido van Rossum in 1985–1990. It is the language of choice for data analysis, backend web development, applications related to artificial intelligence, scientific computing, and to some extent for productive tools, game and desktop app development. It is a highly readable programming language and supports English words as keywords.

Python is widely popular among the data science community, because it contains rich and powerful scientific tools and libraries, such as – numpy, scikit-learn, pandas, matplotlib and pybrain. Other key features of Python making it useful in general-purpose computing are:

- **Interpreted Language:** It is processed at runtime by Python interpreter and thus provides rapid prototyping.
- **Dynamic type:** It can infer the type of data without explicit declaration and thus offers a concise source code, better reuse of module and metaprogramming which ultimately provides the capability to execute many programming behaviors at runtime.

- **Automatic memory management:** It implements automatic garbage collection. Python's memory manager periodically looks for any objects that are no longer referenced by the program and clears memory.
- **Object Oriented:** It encapsulates code and data within objects and therefore provides easy reusability of code.
- **Supports imperative, functional and procedural styles of programming.** Python program may contain statements, expressions and/or functional call.
- **Interactive Mode:** It provides interactive prompt where code snippets can be tested and debugged.
- **Portable:** It can run on wide variety of hardware platforms.
- **Extendable:** low - level modules can be added easily to Python interpreter.
- **Scalable:** It provides better structure and support for large programs.
- **Databases:** Python provides interfaces to all major commercial databases.

2.2. Message Passing Interface (MPI)

Message Passing Interface (MPI) is a standard and portable communication protocol. It provides message-passing specification for communication among processes within a parallel program running on a distributed memory system. The MPI bodes following advantages:

- **High performance:** MPI supports low-latency, high bandwidth interconnects like Ethernet [28], Intel's Omni-Path [29], Mellanox's InfiniBand [30] and works on all

different types of CPU architectures. MPI systems can provide performance up to or above a teraflop or 10^{12} floating point operations per second.

- **Scalability:** It refers to system's capacity to grow or extend to deal with increasing work load.
- **Portability:** MPI programs works on all different types of CPU architectures and network interconnects (Ethernet, Omni-Path and InfiniBand) without requiring any changes to code.
- MPI supports both Point-to-Point and collective communications.
- **Point-to-Point** communication refers to transmittance of a message or data between a pair of processes. One process send data with MPI_SEND and other process receives data with MPI_RECV. Both processes are aware of source and destination, unique tag associated, type and amount of data within message. Figure 2.1 is a typical example of point-to-point communication

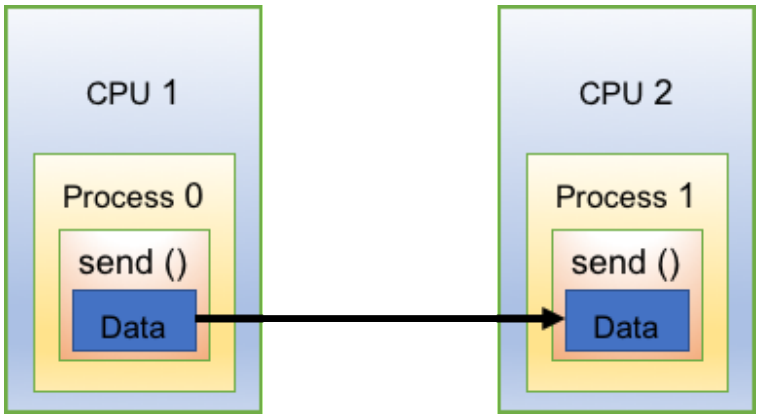

Figure 2.1 Point to point communication

Collective communication refers to communication with participation of all the processes within the communicator. It involves group of processes to send and receive message. Collective communication provides various benefits in parallel communication [31].

1) Offer a higher-level of abstraction for often occurring communication patterns,

2) Separate desired data movement from actual implementation,

3) Allow numerous optimizations internally,

4) Simplify code maintenance and readability,

5) Reduce communication costs compared to (trivial) linear algorithms,

6) Essential for scalability of applications at large process counts.

MPI functions used for collective communication are listed below.

a) MPI_BCAST (Broadcast from one process to all processes within a group)

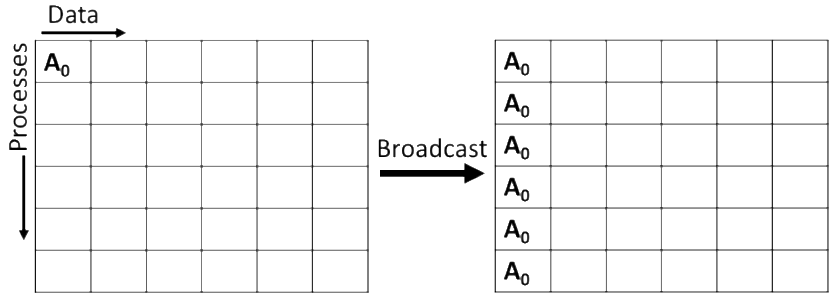

Figure 2.2 Broadcast operation send data to all processes

b) MPI_GATHER (Gather data from all processes within group to one process)

c) MPI_SCATTER (Scatter data from one process to all processes within a group)

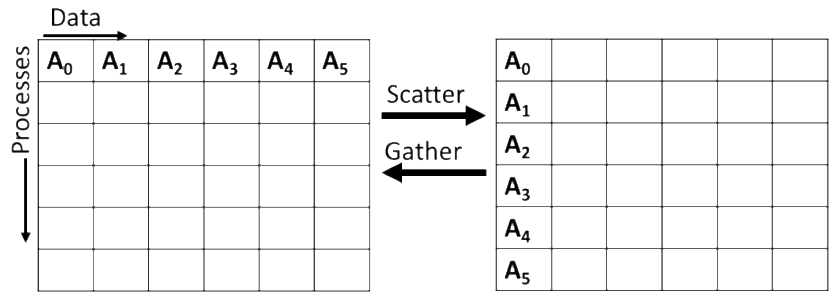

Figure 2.3 Scatter operation divides data to all participating processes and gather operation collects data for all participating processes

d) MPI_ALLGATHER (Similar to gather where all processes receive the result)

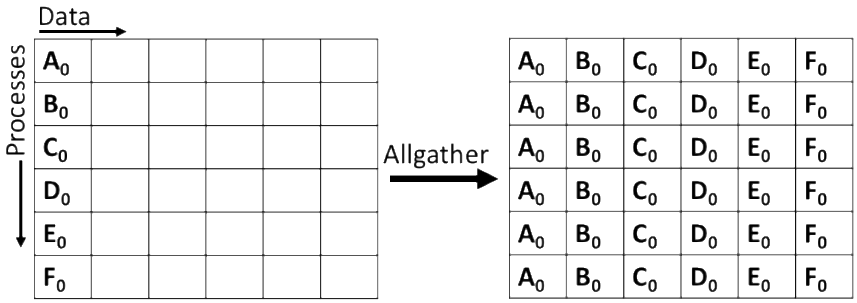

Figure 2.4 Allgather operation collects data from all processes and broadcast complete set of data to all participating processes

e) MPI_ALLTOALL (scatter or gather data from all processes to all process of a group)

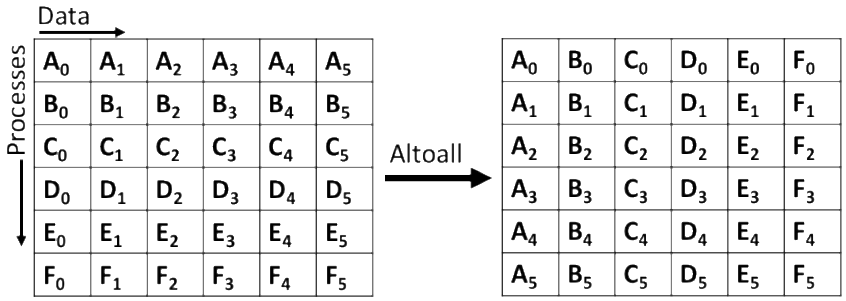

Figure 2.5 Alltoall operation divides data from all processes and then broadcast complete set of data to all processes

MPI is used in Supercomputing or high-performance computing. Super computers can give performance up to 180 petaflops (1015 floating point operation per second), for example Google Deep Learning Cloud [32]. MPI bindings were originally written for C, C++ and Fortran. Although, currently many others languages are providing bindings for MPI such as Python, R, .NET, Java and MATLAB.

MPI for Input/Output operations on File

MPI also provides specification for multiple processes to read and write simultaneously to common file referred to as MPI IO. It is similar to POSIX IO, calls to the MPI functions for reading or writing must be preceded by a call to MPI function to create/open file with MPI_File_open() and must end with closing a file with MPI_File_close. It provides a file pointer (to locate offset) which can be independent for a process or can be shared among the processes. In MPI IO, all participating processes can participate in reading or writing a portion of a common file. Three common ways to position pointer for multiple processes to simultaneously read/write to file are:

a) Use of individual file pointers using MPI functions- MPI_File_seek(), MPI_File_read()
b) Calculation of byte offset and read at location with MPI_File_read_at()
c) Using shared file pointer with MPI_File_seek_shared(), and MPI_File_read_shared()

2.2.1. MPI in Python

There are multiple libraries supporting MPI from within a Python code, including pypar [36], MYMPI [37], MPI Python [38], and mpi4py [39, 40]. We decided to investigate in this paper the mpi4py library, since it is the most complete, widely used and actively developed library among the available solutions.

2.2.1. a MPI4PY

It is Python package which provides bindings for Message Passing Interface (MPI). It is standard for message passing that allows Python program to exploit multiple processors on distributed environment. It provides object-oriented interface that resembles the C++ bindings for MPI-2. It provides functionalities for point-to-point and collective communications of any pickable object and optimized communications for objects that have single-segment buffer interface such as NumPy arrays, bytes/string/array objects.

2.3. Natural Language Processing (NLP)

NLP is the field of science which deals with interactions between human language and computers. It includes areas such as artificial intelligence, computer science, and computational linguistics. This field covers computer understanding and manipulation of human language. It provides ways for computers to analyze, understand, and derive meaning from human language.

Need for NLP

There is a huge amount of data available on web. The size of the World Wide Web for indexed web pages is approximated as 4.59 billion pages as of Monday, 02 October, 2017 [33]. NLP is required for applications for processing large amount of text. Few notable examples of NLP are:

- Text classification
- Indexing and searching
- Automatic translation
- Speech recognition
- Information retrieval
- Automatic summarization
- Question answer bots
- Text generation
- Knowledge acquisition

2.3.1. Natural Language Toolkit (NLTK)

Natural Language toolkit (NLTK) is a set of programs and libraries that provides functionalities such as tokenization, stemming, classification, speech tagging, text parsing, and semantic reasoning for English. It is written in Python programming language and was developed by Steven Bird and Edward Loper at the University of Pennsylvania [34].

It is a very powerful tool to build applications that need human language data. It provides over 50 corpora and lexical resources, toy grammars, trained models. NLTK data can be downloaded from Python shell with download method:

```
>>> import nltk
>>> nltk.download()
```

2.4. NUMPY

NUMPY is a Python library for scientific computing, which supports computation for large and multidimensional arrays and matrices and provides large set of high-level mathematical function as built-in functionalities to be operated on arrays and matrices. Numpy provides following functionalities:

- N-dimensional array objects
- Scientific built-in functions
- Tools for integrating Python code with C/C++ and Fortran code
- Mathematical built-in functionalities for linear algebra, Fourier transform, and random numbers.

2.5. SCIKIT-LEARN

It is Python library for data analysis, data mining, machine learning. The scikit-learn project started by David Cournapeau. It is largely written in Python, with some core algorithms written in Cython to achieve performance [35]. It supports functionalities such as:

Classification: Categorizing the object on basis of its type.

a) Regression: Used for predicting or forecasting.

b) Clustering: Used to group similar objects into groups.

c) Dimensionality reduction: Used to reduce number of random variables from set.

d) Model Selection: Used to compare, validate and choose parameters and models.

Chapter 3

Microbenchmark

3.1. Goals of Benchmarks

Benchmarks such as ping-pong, transferring python objects, overlapping communication and computation, and application scenarios like word count were developed to evaluate the performance of Python MPI against C-MPI. The performance of parallel application is a combination of time spent in computation and communication. Both, performance of Python mpi4py with respect to computation [41, 42] and communication [41, 43, 44, 45] have been analyzed in open literature. Yet, there are many open questions with respect to the performance of mpi4py, because previous studies have either not used high speed network interconnects such as InfiniBand or have not covered programming aspects such as overlapping computation and communication which is important in parallel programming applications.

In this chapter, the performance of mpi4py is evaluated on both Gigabit Ethernet (GE) and InfiniBand network interconnect. Also, a benchmark is exclusively developed and studied to evaluate performance in case of overlapping computation and communication.

3.2. Terminology

Benchmark is a small program run in order to access the relative performance of object, library or test suits.

Latency: It is amount of time a message takes to reach from one designated location to its destination location.

Bandwidth: It is the amount of data that is transferred in a fixed amount of time.

Description of platform used in the study: All benchmarks in this thesis have been executed on Crill Cluster at University of Houston. It consists of 16 nodes with four 12 core AMD Opteron processors cores each (48 cores per node, 768 cores total) with 64 GB of main memory per node. The cluster is connected to a DDR and a QDR InfiniBand interconnect as well as Gigabit Ethernet switch.

These benchmarks were executed with mpi4py 2.0.0 on top of Open MPI [3.6] version 2.0.1. All measurements for each benchmark are executed three times and the average value per message length is being presented subsequently.

3.2.1 Ping-Pong Benchmark

Moving data between processes is one of the major bottlenecks in parallel computing. The Ping-Pong benchmark measures the latency and bandwidth of communication operation. This benchmark was used to compare performance of mpi4py against the performance obtained with similar ping-pong benchmark written in C-MPI [47]. Both C and Python benchmarks were run on same platform using the same MPI library for the actual communication. The difference obtained between these two benchmarks was used to quantify the penalty introduced by the mpi4py layer. The data transferred in python version of benchmark is allocated as numpy array [48], which allows the data to be transferred without requiring serialization (often referred to as pickling).

a) Comparison of mpi4py and C-MPI on InfiniBand

InfiniBand is an industry-standard specification that defines an input/output architecture used to interconnect servers, communications infrastructure equipment, storage and embedded systems. It is most common network interconnect for high performance computing or supercomputing areas [49].

Figures 3.1 and 3.2 show the throughput and execution time comparisons of mpi4Py and C- MPI on InfiniBand network interconnect, respectively. As is noted in the figures, the throughput and the execution time of ping-pong benchmark on InfiniBand network showed very little or no overhead caused by the mpi4py.

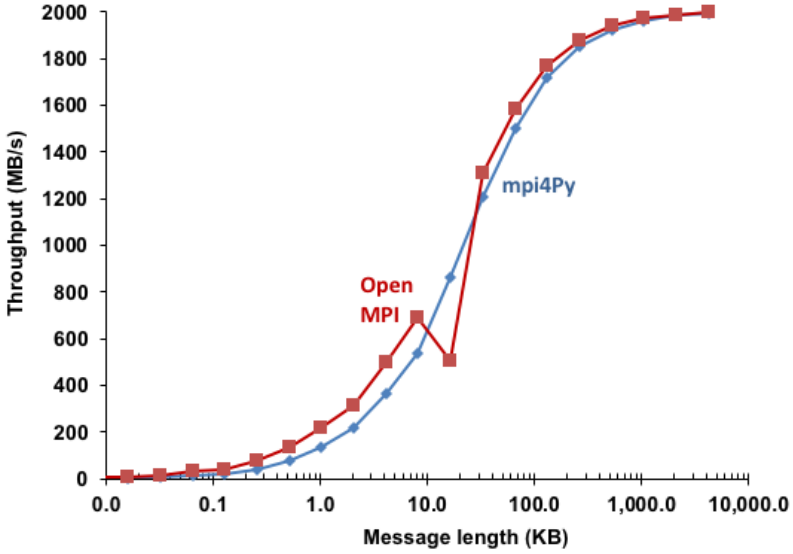

Figure 3.1 Comparison of throughput on InfiniBand network interconnect for mpi4py and open MPI

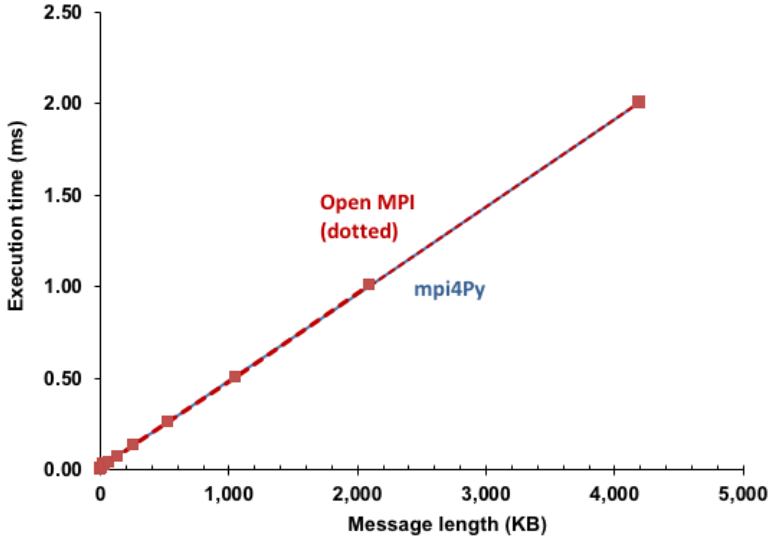

Figure 3.2 Comparison of execution time on InfiniBand network interconnect for mpi4py and open MPI

b) Comparison of mpi4py and Open MPI on Ethernet Network

Figures 3.3 and 3.4 compare the throughput of mpi4Py with Open MPI on Gigabit Ethernet network. It is observed that there is virtually no overhead when using mpi4py compared to C version of same benchmark on Ethernet network as well for numpy arrays.

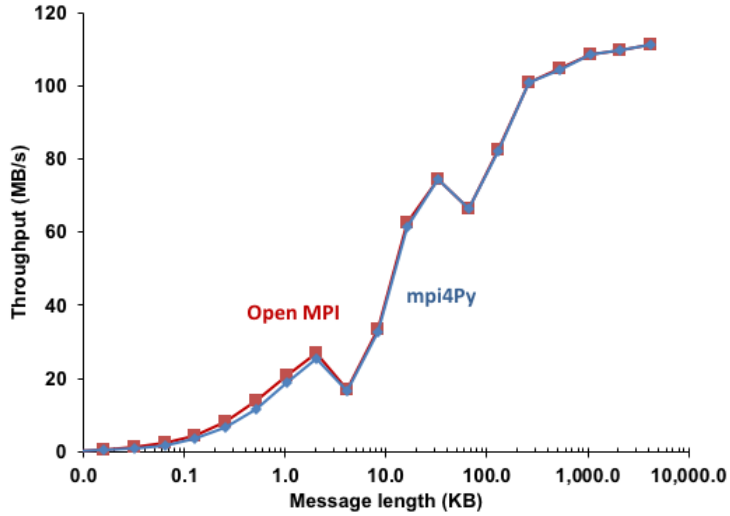

Figure 3.3 Comparison of throughput on Ethernet network interconnect for mpi4py and open MPI

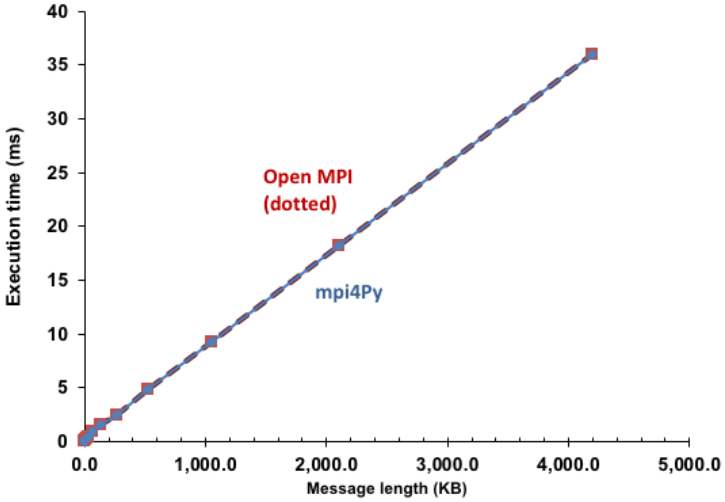

Figure 3.4 Comparison of execution time on Ethernet network interconnect for mpi4py and open MPI

3.2.2. Transferring Python Objects

The Python MPI, mpi4py has the ability to transfer raw data buffers. It also offers interfaces that allows transfer of Python's objects from one process to another. These interfaces, however, require serializing the objects before sending and de-serializing after receiving the corresponding number of bytes, and thus it is expected to have an overhead compared to direct buffer data transfer. To quantify this serializing and de-serializing overhead, ping-pong benchmark was developed to transfer data as Python list with a given number of elements.

Figures 3.5 and 3.6 present the time taken in transferring (execution time) the corresponding number of bytes of the Python list using the InfiniBand and the Gigabit Ethernet (GE) networks, respectively. In both figures, the number of bytes sent or received are shown, instead of the length of the list, in order to make the direct correlation to the benchmark numbers obtained with ping-pong version with numpy arrays. The maximum message size shown is approximately 4.7MB, which was the size in bytes of the Python list with 524,288 elements.

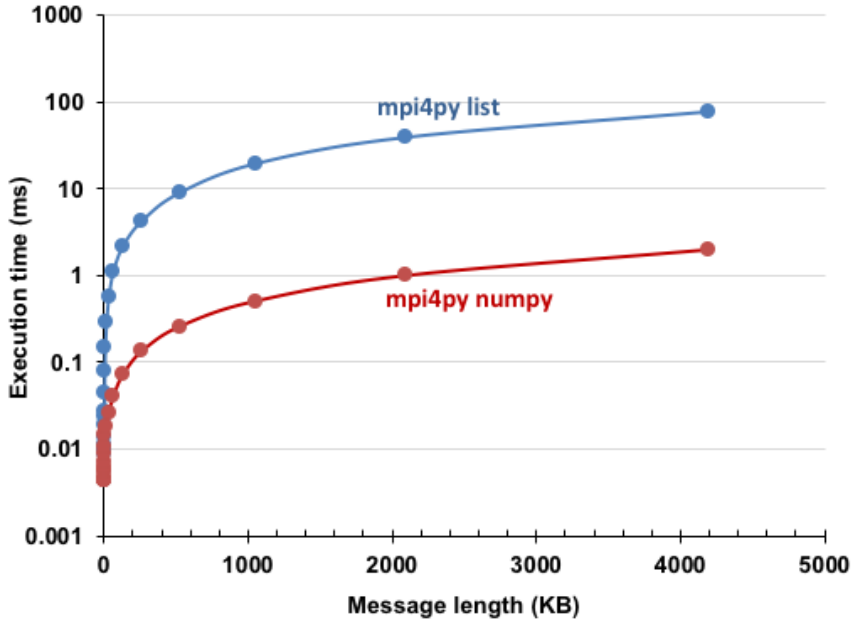

Figure 3.5 Execution time vs message length for mpi4py list and numpy arrays on InfiniBand

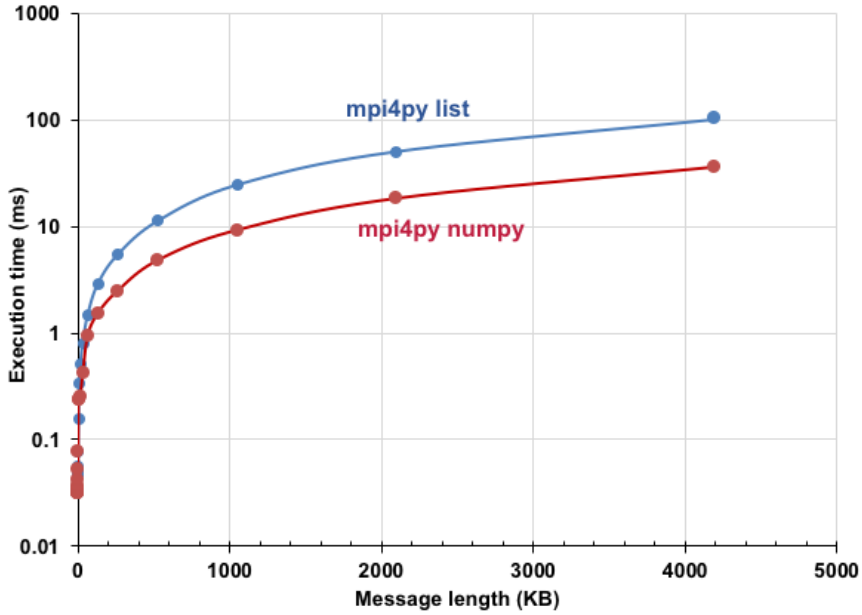

Figure 3.6 Execution time vs message length for mpi4py list and numpy arrays on GE

As expected (Figs. 3.5 and 3.6), serializing and de-serializing the Python list increases the time it takes to transfer the corresponding number of bytes significantly. Using the InfiniBand network, the execution time of the data transfer operation increases by more than a factor of 30 to nearly 100 milli second (ms). Over Gigabit Ethernet, the increase in the costs of the data transfer operation is somewhat less pronounced. In fact, it is noted that the execution time obtained when transferring Python objects over InfiniBand and Gigabit Ethernet are nearly identical, because the actual data transfer operation is only responsible for a small fraction of the overall costs.

3.2.3. Overlapping Communication and Computation in Benchmark

This benchmark was written to evaluate the ability of overlapping data transfer operations with compute operations. This is commonly used to hide the cost of data transfer operations, which could be essential to achieve speedup in applications having high communication costs, as seen for the data transfer operation of Python list in previous benchmark in section 3.2.2 (Figs. 3.5 and 3.6). In this, the benchmark is executing a compute function after posting the non-blocking Isend and Irecv operations. The compute operation is configured to take the equal amount of time as the data transfer itself. The costs of the data transfer are measured in advance before the overlap test. Thus, we expect to observe an overall execution time equal or larger than the time required to perform the data transfer only with the upper bound being sum of the time spent in the compute operation and communication combined.

Figures 3.7 and 3.8 show the execution time of a non-blocking Ping-Pong benchmark of mpi4Py and native open MPI over InfiniBand (Fig 3.7) and Gigabit Ethernet (Fig 3.8). In Figs 3.7 and 3.8, two values are represented - the first value represents the sum of the communication costs and computation costs, while the second value represents the actual measured time of the overlap benchmark. The difference between those two values is the time the application was able to save by overlapping communication and computation.

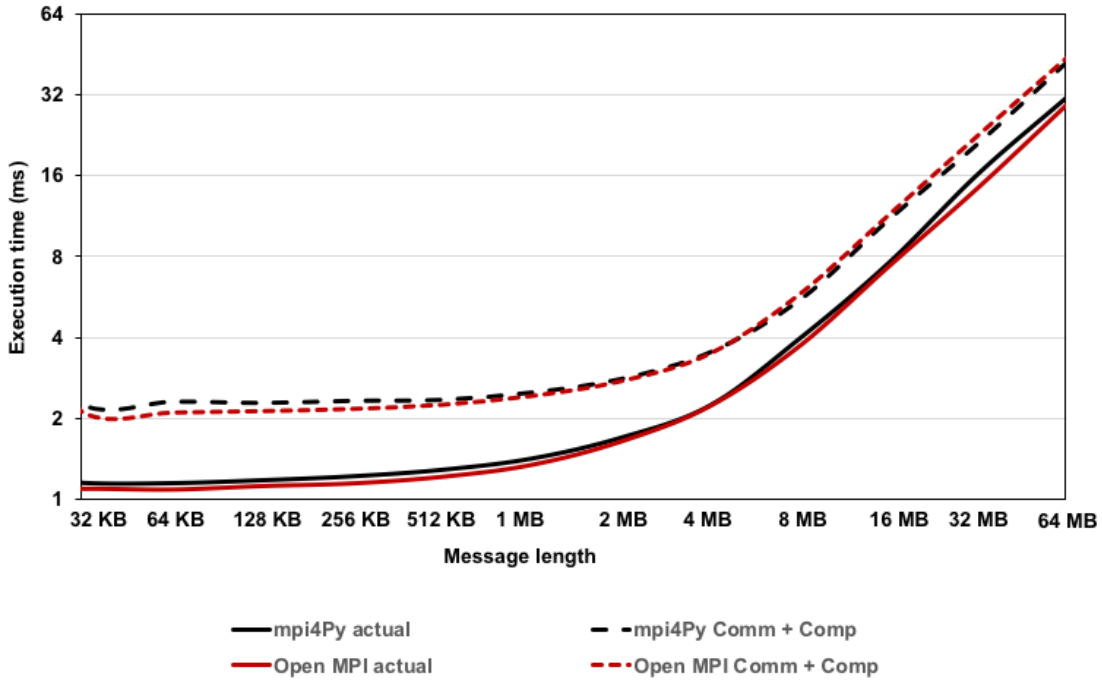

Figure 3.7 mpi vs native Open MPI on InfiniBand

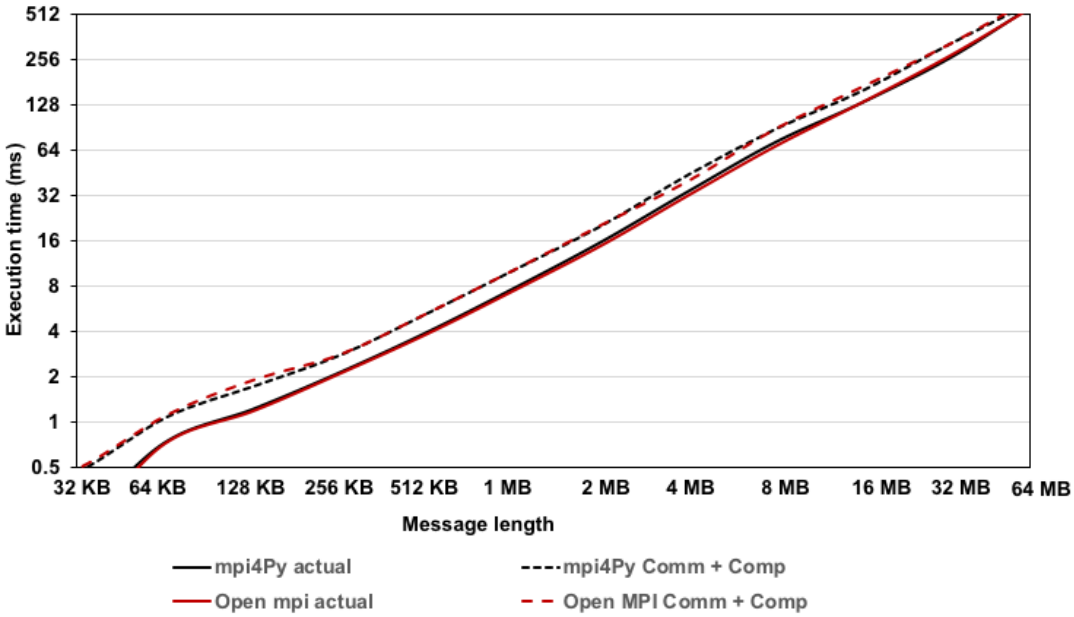

Figure 3.8 mpi vs native Open MPI on Gigabit Ethernet (GE)

The results in Figs. 3.7 and 3.8 indicate very similar behavior between C and Python version of the benchmark, namely some benefits from overlapping communication and computation but less than what could be expected in an ideal scenario. The limited benefits are due to the fact that the benchmark used did not incorporate calls to the MPI library during the compute sequence. Thus, the MPI library was not able to 'progress' pending communication operations continuously. However, evaluating the results from the C vs. Python perspective, there was no quantifiable degradation in the performance of the Python benchmark compared to C counterpart.

Chapter 4

Application Scenarios for Natural Language Processing

Natural Language Processing is an area of research and application that explores how computers can be used to understand and manipulate natural language text or speech to do useful things [50]. As a more general definition, we can say it refers to computer systems that can understand, process, and generate human language. The input to the systems might be text, spoken language, or keyboard input [51].

4.1. Common Applications of NLP

In the following subsection, a few typical application use-cases from NLP are being presented.

Machine translation: It is area under computational linguistics that deals with use of software for translating human language (in form of text or speech) from one language to another language. A few examples are: Pairaphrase, which is a web-based application for human language translation offered by Pairaphrase LLC, Easy translator, which is a web service for language translation offered by Xerox Corporation, and Watson language

translator, which provides cross platform applications for language translation offered by IBM company. Two models developed by IBM are research rule-based and statistical models. The rule-based engine is available as an on-site product, WebSphere translation server. The Bluemix or IBM Watson translation is available as an API for developers [52, 53, 54].

Fighting spam: Spam filter is generally a part of computer software or service that will filter coming messages/emails from irrelevant or inappropriate messages. Spam is generally referred to a term where a sender floods the internet with irrelevant messages with many copies of the same messages. The spam is usually an attempt to force the message on people who would not otherwise choose to receive it [55].

Information extraction (IE): As the name suggest, it refers to extracting meaningful information from un-structured or semi-structured machine-readable data. It mostly deals with processing large volumes of human language like text or speech. Due to increasing amount of unstructured information such as twitter posts, blogs, video cameras, audio recorders, and web documents, retrieval of meaningful information is becoming very significant. IE is driving force for crucial decision-making processes in many companies and government organizations.

Summarization: Summarization refers to the sum up of the main points from available data. In technical terms it can be referred as finding a compact description of a dataset [56]. A true summary succinctly expresses the gist of a document, revealing the essence of its content [57]. Text summarizer applications are intended to automatically construct summaries of a machine-readable datasets.

Question answering (QA): Question answering is a computer science discipline within the fields of information retrieval and natural language processing that is concerned with building systems that automatically answer questions posed by humans in a natural language [4.9]. Some examples are START (http://start.csail.mit.edu/index.php, by Boris Katz and his associates of the InfoLab Group at the MIT Computer Science and Artificial Intelligence Laboratory) and IBM Watson QA system.

4.2. Application Scenarios for Common NLP Applications

Application scenarios are the common sequence of events/actions/steps (as a part of application) that will be executed in order to process complete application. The first section (section 4.1) of the current chapter has already discussed the common applications of NLP. In the following (section 4.2.3), the common scenarios that may be used in any or all applications will be discussed.

4.2.1 Motivation for using parallel/distributed computing for NLP application scenarios

In today's internet era, there are overwhelming and ever-increasing amount of textual information that must be processed in a reasonable time frame. This scenario has led to a paradigm shift in the computing architectures and large-scale data processing strategies used in the Natural Language Processing field [59]. Many strategies have recently been proposed to process large amount of textual data with super-fast computing powers like NLP library with Apache Spark [60], for example Apache NLP package in R, and NLP toolkit in Python.

4.2.2 Common application scenarios with parallel and distributed computing

This section presents the execution time of four NLP scenarios (word-count, bigram, trigram, and tf-idf) under the high-performance computing (HPC) environment. For calculation of the execution time, a small and a large dataset was created from source www. gutenberg.org, and saved in pvfs2 file system. The small dataset consisted of 6 books with total of 145 KB data, and the large dataset consisted of 216 books with about 18 MB of total data. Each application scenario was executed 3 times and the average the three was used for plotting the results.

Terms used for performance metrics evaluation were:

Parallel Speed-up: Speed-up (S) is the ratio of execution time with one process (T_1) over execution time with 'P' processes (T_P). It is used to calculate how much faster does a problem run on 'P' processors compared to 1 processor.

$$S(P) = \frac{T(1)}{T(P)}$$

Where

S(P) is the speed-up for P processes

T(1) is execution time for one process

T(P) is execution time for 'P' processes

Parallel Efficiency: Efficiency is defined as speedup normalized by number of processors. It gives an indication of the effectiveness of the parallelization strategy used. Applications achieving a speedup value close to the number of processors used will have a parallel efficiency close to 1, while applications with the speed up value significantly lower than the number of processors used will have an efficiency close to zero.

$$E(P) = \frac{S(P)}{P}$$

Where

E(P) is parallel efficiency for 'P' number of processes

S(P) is the speed-up for 'P' number of processes

P is the number of processes

4.2.2.1 Word Count

Word count, also known as term frequency, is used in almost every natural language processing or data mining applications. Word count is used to determine the number of occurrences of the term in document. This is often used to determine the importance (or weight) of that term, assuming that frequently occurring words are of greater significance for the document than less frequently occurring once. Note, that there are multiple ways of determining the weight of a term. We present with TF-IDF another approach later in this section.

Steps involved in creating word count scenario for HPC environment were:

1. The word count program was developed to read a number of books assigned to each process based on its rank. For dividing books among each process two variables start_index and end_index were created and assigned values as per the rule:

   ```
   start_index = int(rank * length / size)
   end_index   = int((rank+1) * length / size)
   ```

   where,

   rank = rank of process

   length = total number of books (documents)

   size = total number of processes assigned.

2. Each process tokenizes the words using Python's NLTK package and stored its list of tokens.
3. Each process then counts the occurrence of each token within its local list using Counter class from collections library.

4. Counter object was shared globally among all processes with "Allreduce" operation in mpi4py library. The Allreduce operation is a collective operation where all participating processes are used to determine a global value. In the word count scenario described here, the operation is used to determine the number of occurrences of a term across all processes, by adding up the number of occurrences of that term on each process.

   <snapshot below>

```
counterSumOp = MPI.Op.Create(addCounter, commute=True)
global_counter = comm.allreduce(word_count, op=counterSumOp)
```

5. The global Counter object contains the word and count as dictionary for all the documents.
6. This global Counter object was then divided among processes based on its rank (based on below formula) to write to a shared file.

```
s_index = int(rank * counter_length / size)
e_index = int((rank+1) * counter_length / size)
seq = total_counter.items()[s_index:e_index]
```

7. Each process writes to shared file using "write_ordered" operation of mpi4py library. The "write_ordered" is the collective operation where all processes write to same file using shared file pointer. Shared file pointer specifies the location for writing data based on the rank and buffer size of each process participating in the communicator group.

```
<Code Snippet>
length = file_len(inputfile)
start_index = int(rank * length / size)
end_index   = int((rank+1) * length / size)
tokens = get_tokens()
word_count = Counter(tokens)
counterSumOp = MPI.Op.Create(addCounter, commute=True)
total_counter=comm.allreduce(word_count, op=counterSumOp)
counter_length = len(total_counter.items())
s_index = int(rank * counter_length / size)
e_index = int((rank+1) * counter_length / size)
seq = total_counter.items()[s_index:e_index]
fh.Write_ordered("%s\n" % seq)
```

Figures 4.1 and 4.2 compare the average (for 3 runs) execution time of the word-count program for the small and the large data set, respectively. Figures 4.3 and 4.4 present the parallel speedup and efficiency of the word-count program for the large dataset. It is evident that the execution time improved (i.e. decreased) greatly with the increased number of processes, especially for the first 6 to 8 processes after which the increment in the processes resulted in little or no improvement in the execution time.

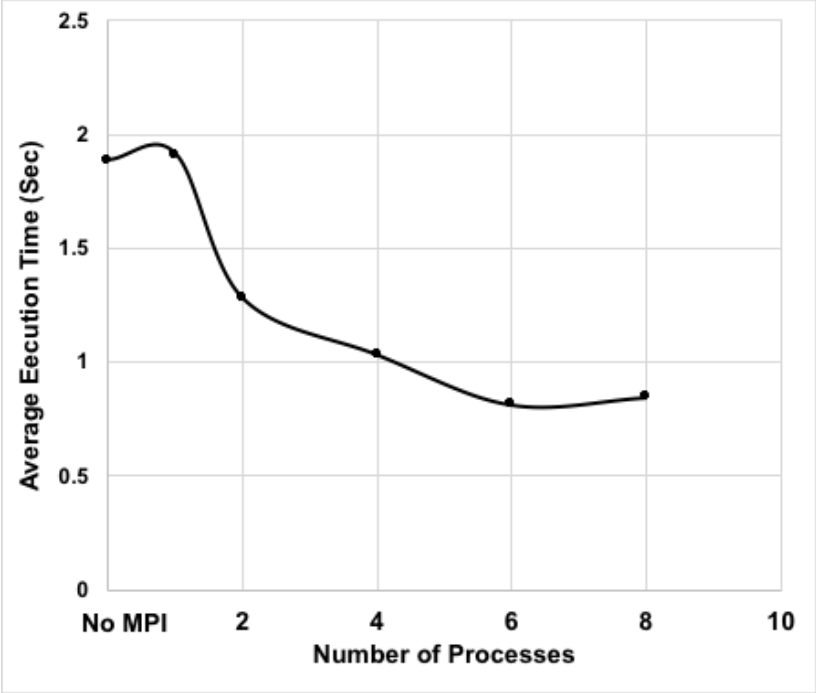

Figure 4.1 Average execution time of word count program for small data set

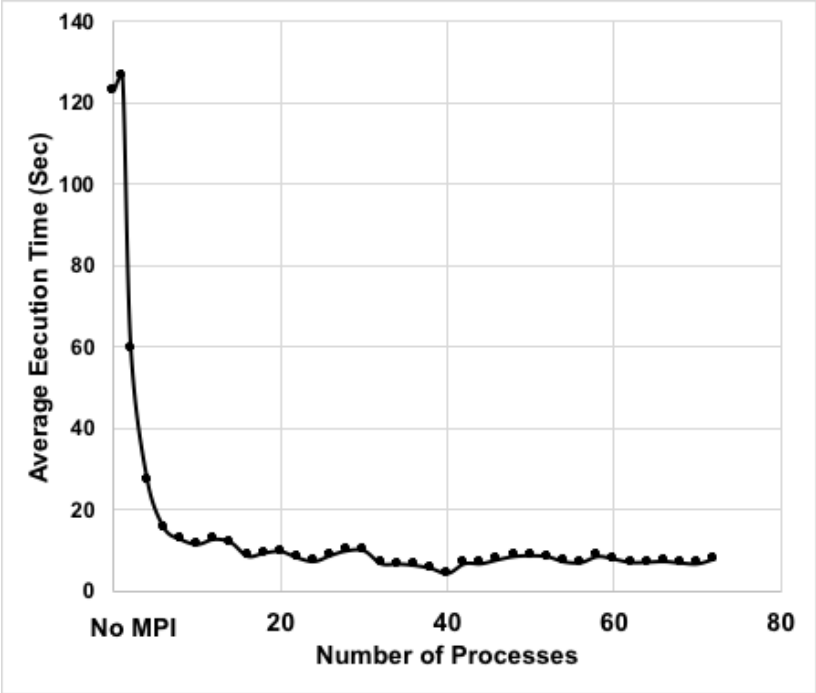

Figure 4.2 Average execution time of word-count program for large data set

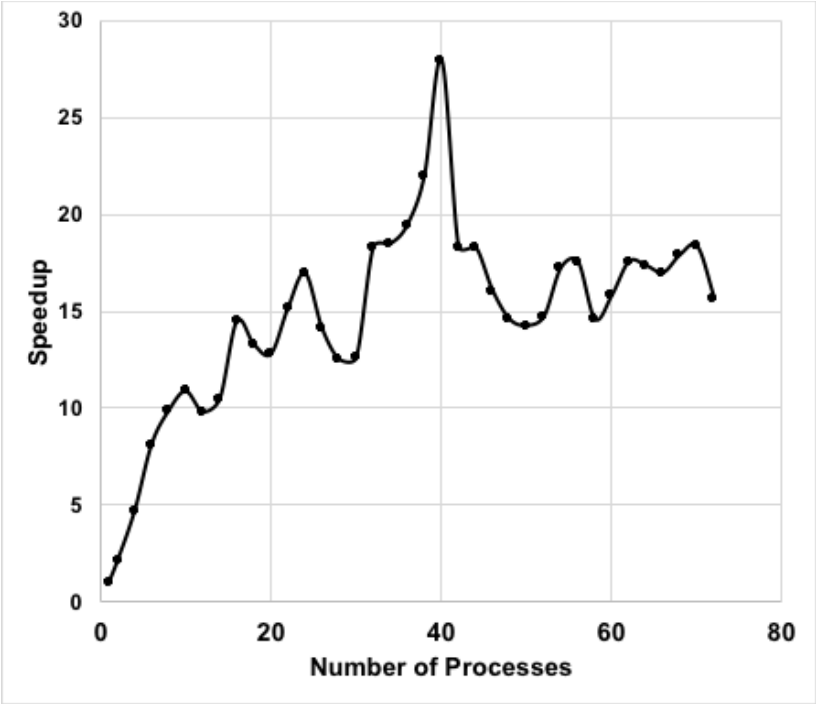

Figure 4.3 Parallel speedup of word-count scenario for large dataset

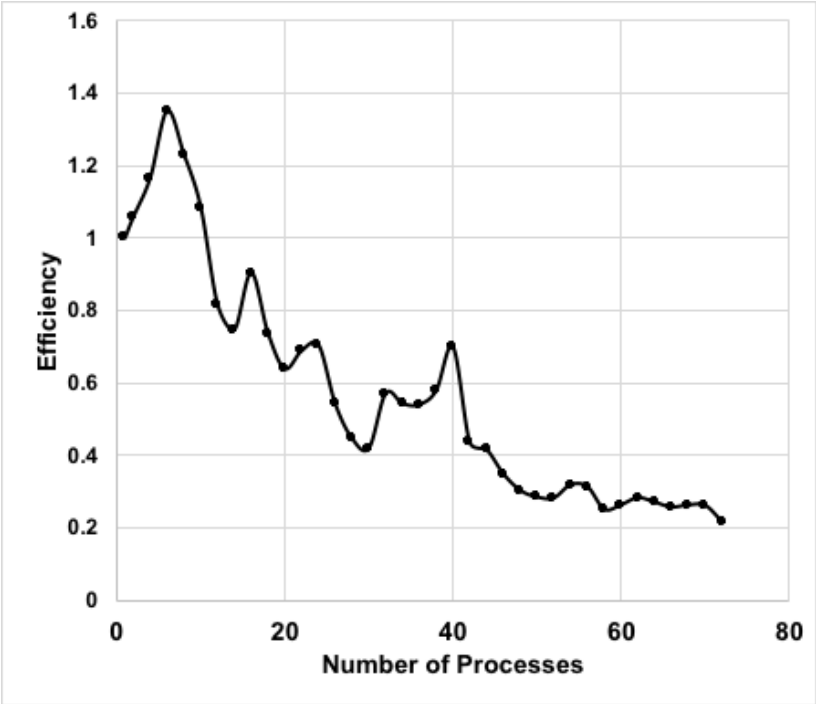

Figure 4.4 Parallel efficiency of word-count scenario for large dataset

4.2.2.2 N-Gram

It is sequence of n co-occurring or continuous words in a document. N-grams are most commonly used in developing language models. Companies are developing web scale n-gram models that can be used in a variety of tasks such as spelling correction, word breaking and text summarization [61]. Text Summarization can also be achieved by N-gram models [62]. n-grams is also used for developing features for supervised Machine Learning models such as SVMs, MaxEnt models, Naive Bayes.

Bigram Count

Bigram count, also known as di-gram, i.e., n-gram with n = 2, applies to the sequence of two co-occurring or continuous words in a document. For this thesis, bigram count was developed using python's NLTK package to tokenize words n produce bigrams. The BigramCollocationFinder class was used to find bigrams in the data set.

Steps involved in creating bigram count scenario for HPC environment were:

1. Similar to word count, bigram count program was developed to read number of books assigned to each process based on its rank.
2. Each process tokenizes the words using Python's NLTK package and stores its list of tokens and BigramCollocationFinder class was used to find bigrams from documents.
3. Each process then counts the occurrence of bigram within its local list using Counter class from collections library.

4. Counter object was shared globally among all processes with “allreduce” operation on mpi4py library. The Allreduce operation is a collective operation where all participating processes are used to determine a global value.
5. The global Counter object contains the bigram and count as dictionary for all the documents.
6. This global Counter object was divided among processes based on its rank to write to a shared file.
7. Each process writes to shared file using “write_ordered” operation of mpi4py library. The “write_ordered” is the collective operation where all processes write to same file using shared file pointer.

```
<Code Snippet>
text = book.read()
lowers = text.lower()
no_punctuation=lowers.translate(None, string.punctuation)
tokens = nltk.word_tokenize(no_punctuation)
bigram_finder = BigramCollocationFinder.from_words(tokens)
for k,v in bigram_finder.ngram_fd.items():
tokens_temp.append(k)
list_of_tokens = list_of_tokens + tokens_temp
```

Figures 4.5 and 4.6 compare the average (for 3 runs) execution time of the bigram-count program for the small and the large dataset, respectively. Figures 4.7 and 4.8 show the parallel speedup and efficiency for the large dataset. The execution time improved significantly with increased number of processes, especially for the first 6 to 8 processes

after which the increased in the processes amounted to little or no improvement in the execution time.

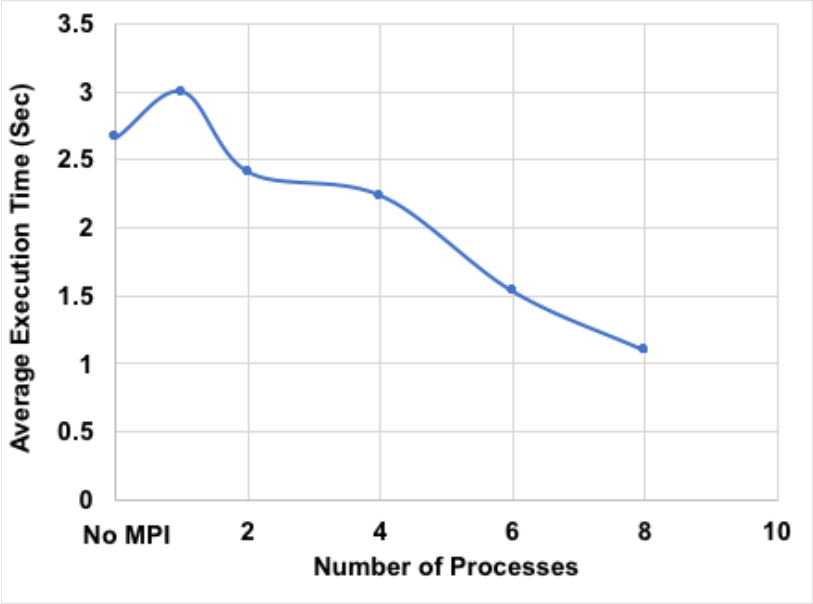

Figure 4.5 Average execution time of bigram-count program for small data set

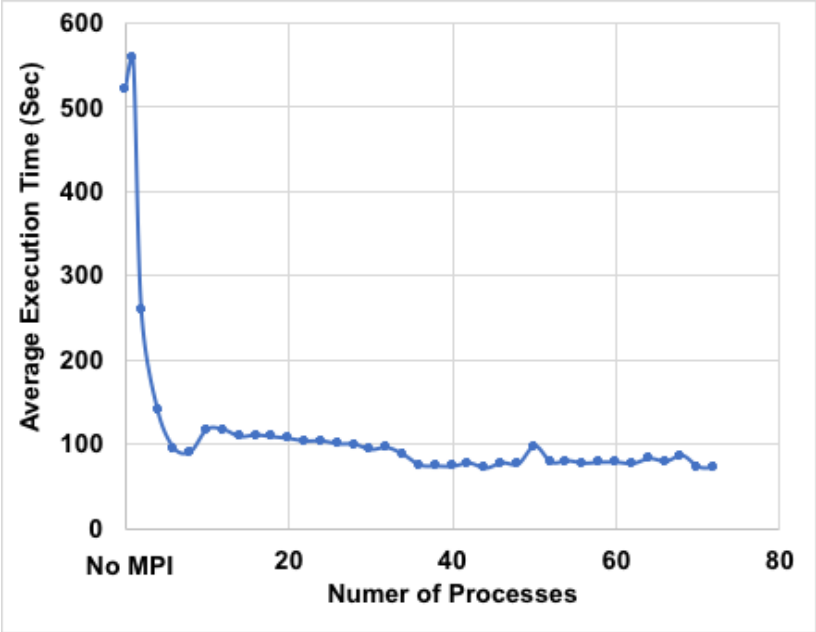

Figure 4.6 Average execution time of bigram-count program for large data set

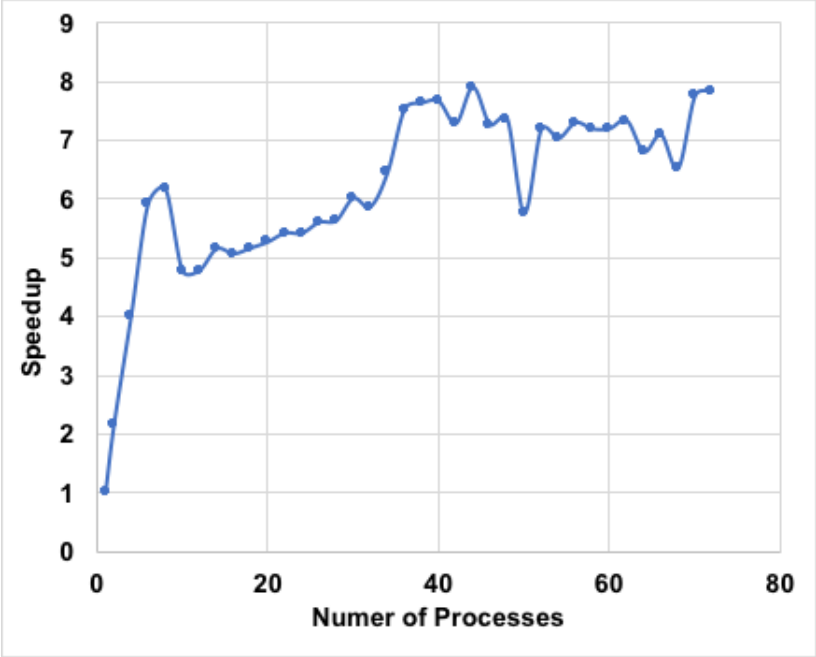

Figure 4.7 Parallel speedup of bigram-count program for large data set

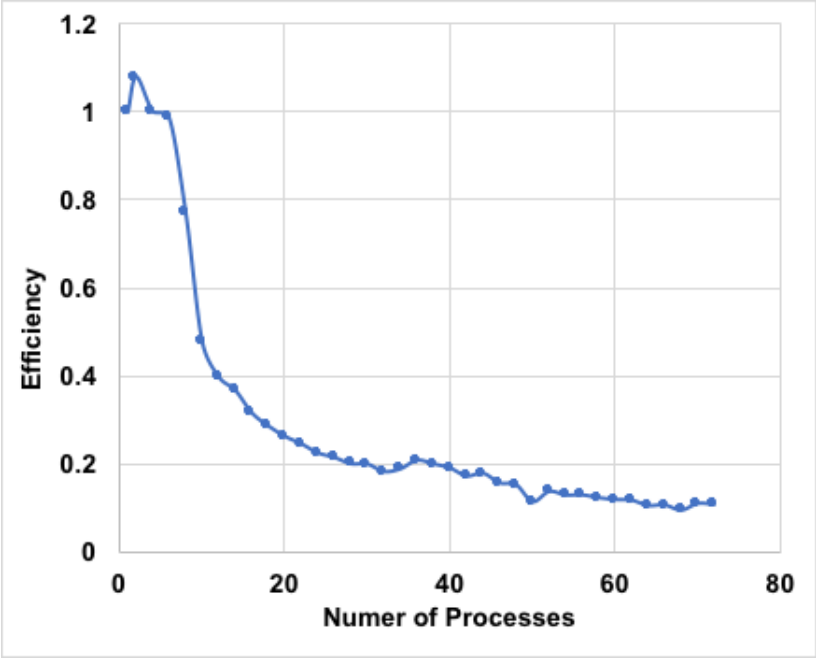

Figure 4.8 Parallel efficiency of bigram-count program for large dataset

Trigram Count

Trigram, which is n-gram with n = 3, applies to the sequence of three co-occurring or continuous words in a document. For this thesis, trigram count was developed using python's NLTK package to tokenize words n produce trigrams. The TrigramCollocationFinder class was used to find trigrams in the data set.

Steps involved in creating trigram count scenario for HPC environment were identical to the Bigram but uses three-word sequences instead of two-words.

```
<Code snippet>
text = book.read()
lowers = text.lower()
no_punctuation = lowers.translate(None, string.punctuation)
tokens = nltk.word_tokenize(no_punctuation)
trigram_finder = TrigramCollocationFinder.from_words(tokens)
for k,v in trigram_finder.ngram_fd.items():
tokens_temp.append(k)
list_of_tokens = list_of_tokens + tokens_temp
```

Figures 4.9 and 4.10 show the average (for 3 runs) execution time of the tri-gram-count program for the small and the large dataset, respectively. For comparison purpose, the execution time of the bigram and work count program is also shown in the figure. Figures 4.11 and 4.12 present the parallel speedup and efficiency of the trigram count program for the large dataset. Similar to the word-count and bigram-count programs, the execution time

improved greatly with the increased number of processes for the first 6 to 8 processes after which the increment in the processes resulted in little or no improvement in the execution time.

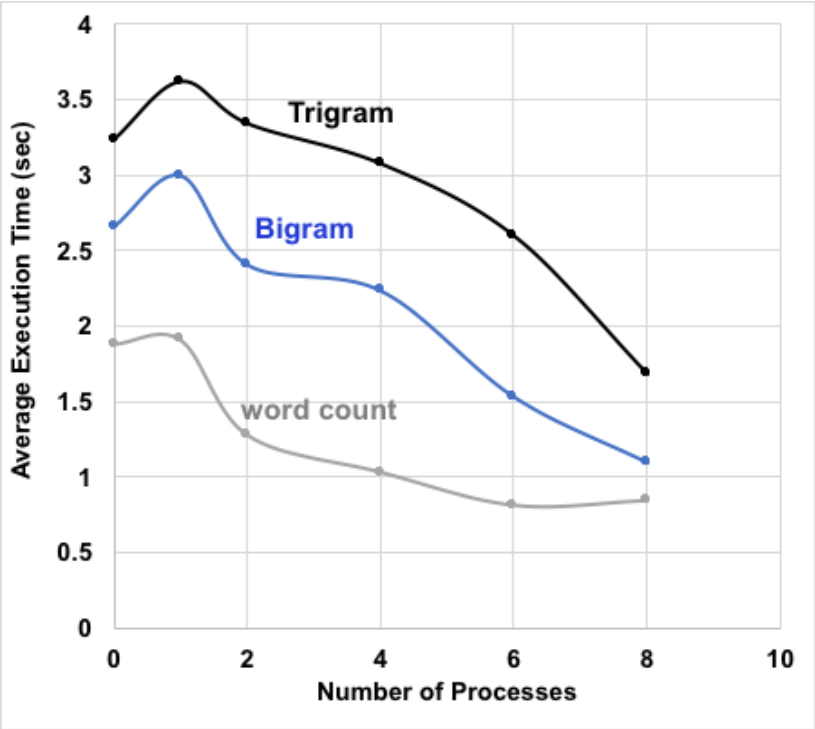

Figure 4.9 Average execution time of trigram count program for small data set. For comparison purpose the execution time of the bigram and word count program is replotted to show the effect of change in communication volume on execution time

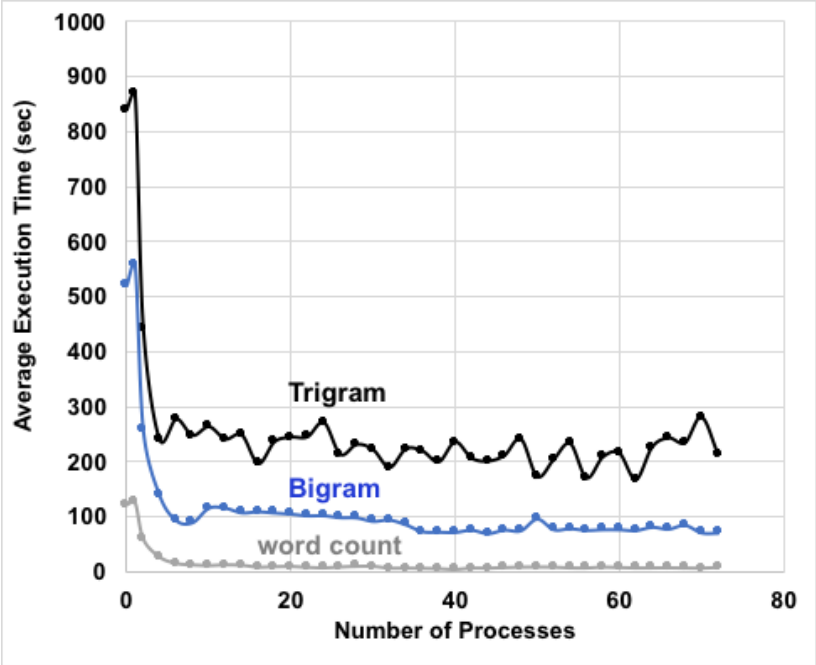

Figure 4.10 Average execution time of trigram count program for large data set. For comparison the execution time of the bigram and the word count program is also plotted to show the effect of change in communication volume on execution time

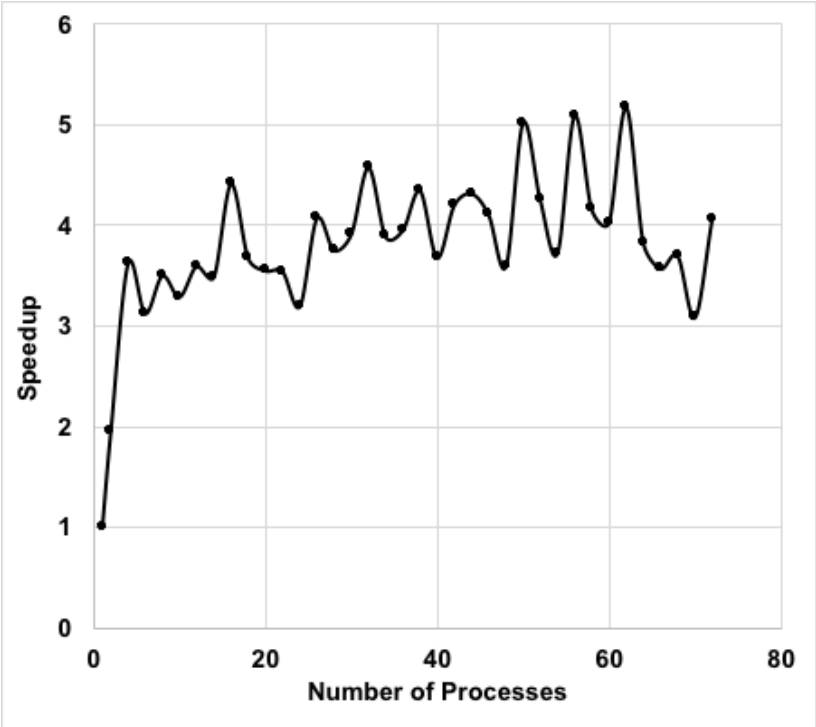

Figure 4.11 Parallel speedup for trigram-count program for large dataset

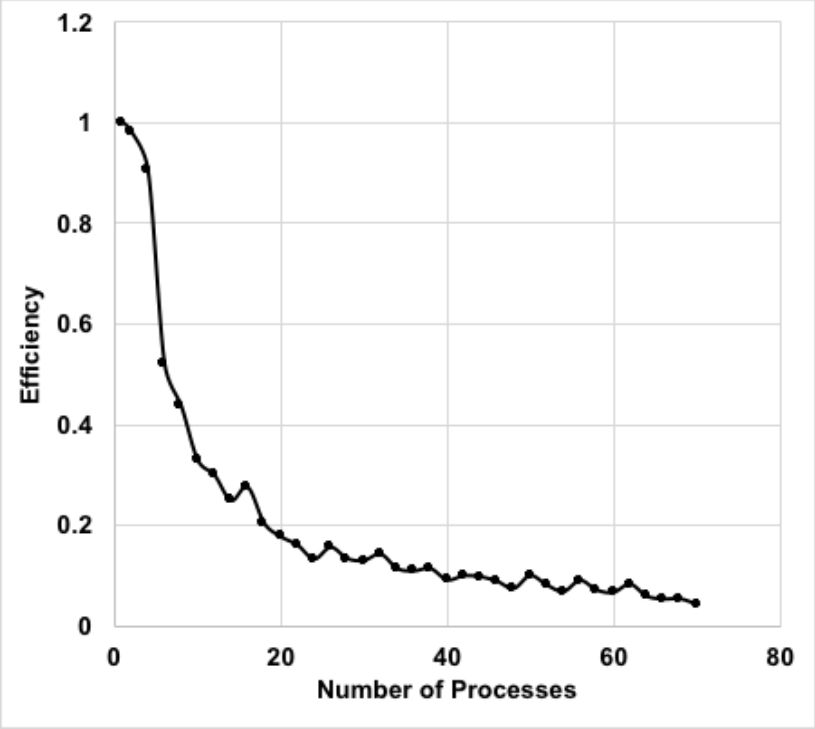

Figure 4.12 Parallel efficiency for trigram-count program for large dataset

4.2.2.3 Term Frequency-Inverse Document Frequency (TF-IDF) Calculation

TF-IDF is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus [63]. It is popular term weighting factor strategy used in information retrieval, text mining and text-based recommender systems.

Tf-idf is calculated as

Tf-idf = tf (term frequency) * idf (inverse document frequency)

Tf-idf can have many variants depending upon its normalization scheme or smoothing factor. For this thesis, tf-idf calculation is based on python's TfidfVectorizer from sklearn package with smooth_idf = false and norm = none.

That is term frequency = number of times a term (t) appears in a document (d).

And, Inverse document frequency idf (t),

$$idf(t) = \log \frac{1 + n_d}{1 + df(d,t)} + 1$$

Steps involved in creating tf-idf scenario for HPC environment were:

1. Similar to word count, bigram and trigram the tfidf program was developed to read number of books assigned to each process based on its rank.
2. Each process creates a term-document matrix for all the words read within the process, using Python's "textmining" package.
3. Each process then stores the frequency of term from term-document matrix into a triad using namedtuple (for document name and term) and dictionary in Python.
4. One dictionary variable "doc_freq" was used to store the number of documents that contain a particular term within a process.
5. This "doc_freq" dictionary variable was shared globally among all processes with "allreduce" operation on mpi4py library. The "allreduce" is the collective operation where all participating processes operate/process on data according to the operator provided to them.

   <snapshot below>

```
counterSumOp = MPI.Op.Create(addCounter, commute=True)
doc_freq = comm.allreduce(doc_freq, op=counterSumOp)
```

6. The tfidf was calculated by each process with global doc_freq variable using same formula from Python's TfidfVectorizer class of scikit learn package.

   <snapshot below>

```
# Calculating tfidf for each term in process
tfidf[Doc_term(key.Doc, key.term)] =
value * (math.log(total_Docs/doc_freq[key.term]) + 1)
```

7. Each process writes its tfidf for all documents and terms to shared file using "write_ordered" operation of mpi4py library. The "write_ordered" is the collective operation where all processes write to same file using shared file pointer. Shared file pointer specifies the location for writing data based on the rank and buffer size of each process participating in the communicator group.

```
From sklearn:
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(smooth_idf=False, norm=None)
response = vectorizer.fit_transform(books)
feature_names = vectorizer.get_feature_names()
```

The above code snippet shows the TfidfVectorizer class of scikit learn package that is used in sequential code and same variation of tfidf is used in parallel version of program.

```
<code snippet>
# Reducing dictionary to share with all processes
 doc_freq = comm.allreduce(doc_freq, op=counterSumOp)
 total_Docs = len(booknames)
 tfidf = {}
```

```
for key, value in Doc_term_count.iteritems():
# Calculating tfidf for each term in process
tfidf[Doc_term(key.Doc, key.term)] =
value * (math.log(total_Docs/doc_freq[key.term]) + 1)
```

Figures 4.13 and 4.14 compare the average (for 3 runs) execution time of the tf-idf program for the small and the large dataset, respectively. Figures 4.15 and 4.16 presents the parallel speedup and efficiency of the tf-idf program for the large dataset. Similar to the other NLP scenarios, the execution time improved significantly with increased number of processes. For the large dataset, a rapid improvement in the execution is noted for the first 6 to 8 processes after which the improvement slowed down.

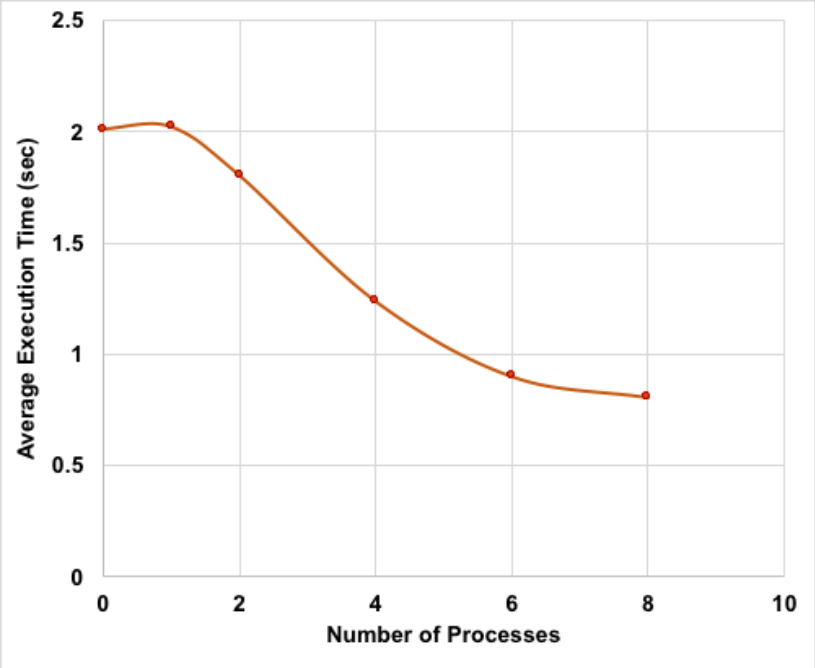

Figure 4.13 Average execution time of tf-idf program for small data set

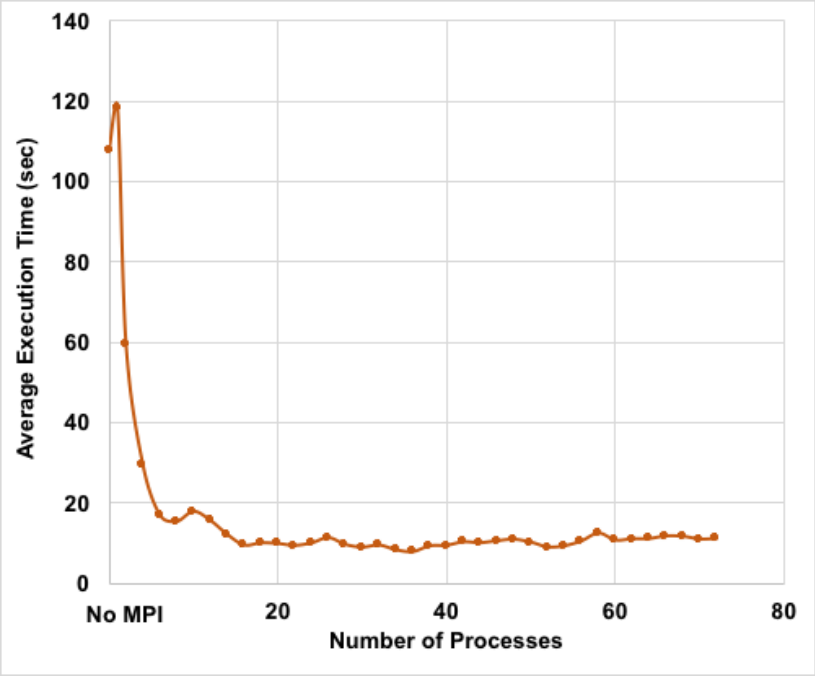

Figure 4.14 Average execution time of tf-idf program for large data set

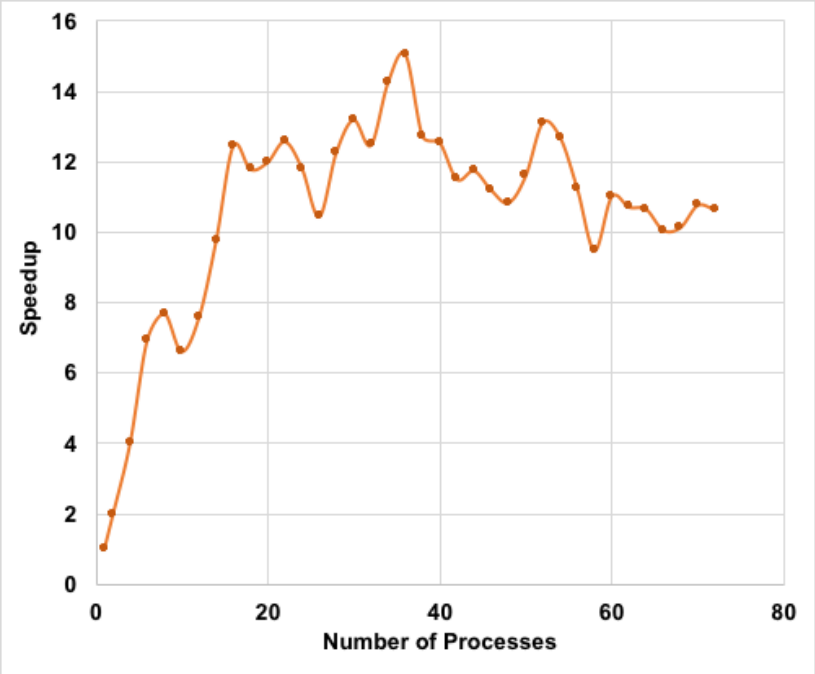

Figure 4.15 Parallel speedup of tf-idf program for large dataset

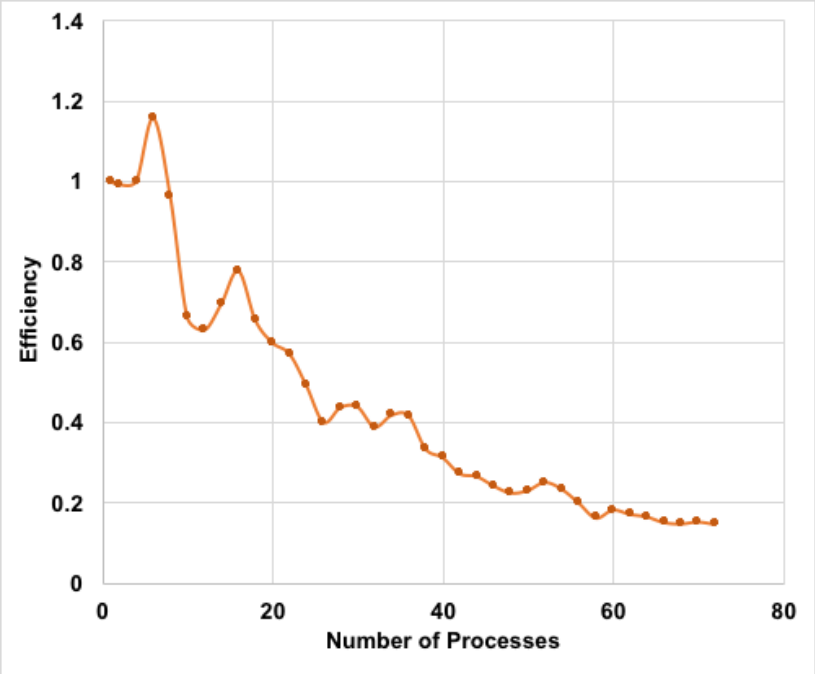

Figure 4.16 Parallel efficiency of tf-idf program for large dataset

Conclusion

The various NLP application scenarios like word count, bi-gram count, tri-gram count, and tf-idf calculation with mpi4py library for distributed computing were developed and executed. A significant improvement in the execution time with increased number of processes was observed under each scenario. The improvement was drastic for the first 6 to 8 processes after which the improvement slowed down or leveled off.

Chapter 5

Summary

The mpi4py (MPI for Python) is a library that provides bindings of the Message Passing Interface (MPI) standard for the Python programming language. It allows any Python program to exploit multiple processors. MPI is a standardized and portable message-passing standard designed by researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs [64].

Generally, applications that require high performance computing environment use C, C++ or Fortran programming languages which support MPI and OpenMP standards for communication among processes to support parallel and distributed computing. And, applications that operate on large data sets are written in programming languages that do not having bindings in the MPI specification. Yet, with increasing problem sizes, these applications also necessitate some form of parallel processing. The goal of this thesis was to evaluate the utilization of MPI with a non-traditional HPC programing language, namely Python, from the performance and functionality perspective.

The first part of the thesis evaluated the performance of a mpi4py library using multiple point-to-point micro-benchmarks, transferring Python objects, overlapping communication and computation cases using an InfiniBand and a Gigabit Ethernet network interconnect. The results showed that in many instances the performance obtained with the Python benchmarks is on par with their C-based counterparts.

The second part of this thesis presented Natural Language Processing (NLP) based application scenarios such as word-count, bigram, trigram count, and tfidf calculation on large dataset. All application scenarios showed a significant improvement in the execution time with the increasing number of processes.

References

1. https://en.wikipedia.org/wiki/Shared_memory, Retrieved on Oct 04, 2017.
2. https://en.wikipedia.org/wiki/Distributed_memory, Retrieved on Oct 10, 2017.
3. https://computing.llnl.gov/tutorials/parallel_comp/#Hybrid, Retrieved on Oct 14, 2017.
4. Ming Shen, Xiaozhong Guo, Pengqi Gao, Datao Yang, You Zhao, Zhonwei Fan, Jin Yu, and Yunfeng Ma, "Application of parallel computing for space debris close approach analysis", Proc. International Conference on Computer Science & Service System, p.n. 2213-2216, 11-13 Aug 2012.
5. Patrick McCormick, Richard Barrett, Bronis de Supinski, Evi Dube, Carter Edwards, Paul Henning, Steve Langer, and Allen McPherson, "Programming Models" Lawrence Livermore National Labaratory report No. LLNL-TR-474731, March 2011.
6. Dongyao Wu, Sherif Sakr, and Liming Zhu, "Big data programming models", Handbook of Big Data Technologies, Springer International Publishing, p.n. 31-63, 2017.
7. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard", Parallel Computing, Vol. 22, Issue 6, p.n. 789-828, Sept 1996.
8. https://www.cs.rutgers.edu/~venugopa/parallel_summer2012/openMP.html, Retrieved on Oct 14, 2017.

9. “OpenMP specification” http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
10. https://hw-lab.com/nvidia-announces-cudax86.html, Retrieved on Oct 10, 2017.
11. https://www.cs.rutgers.edu/~venugopa/parallel_summer2012/index.html, Retrieved on Oct 04, 2017.
12. http://www.brianstorti.com/the-actor-model/, Retrieved on Oct 04, 2017.
13. Tom White, “Hadoop: The Definitive guide (second edition)”, O’ReillyMedia, Inc.
14. David Dewitt, Jim Gray “Parallel database systems: The future of high performance database systems, Communications of the ACM, June 1992/Vol. 35, No. 6.
15. Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica “Apache Spark: A Unified Engine for Big Data Processing” Communications of the ACM, Vol. 59 No. 11, p.n. 56-65.
16. Aravind K. Joshi “Natural Language Processing” Science 13 Sep 1991: Vol. 253, Issue 5025, p.n. 1242-1249, 1991.
17. Frédéric Landragin, “Human-Machine Dialogue Design and Challenges” http://fred.landragin.free.fr/publi/13_WILEY.pdf, Retrieved on Feb 04, 2018.
18. https://www.theinquirer.net/inquirer/news/2124878/nvidia-cray-pgi-caps-launch-openacc-programming-standard-parallel-computing, The Inquirer, Feb 04, 2018
19. Jonathan Tompson and Kristofer Schlachter “An Introduction to the OpenCL Programming Model” NYU: Media Research Lab, Retrieved on Feb 04, 2018.

20. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica "Spark: Cluster Computing with Working Sets", University of California, Berkeley (2010).
21. A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M.J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics", VLDB Journal, 23(6) (2014).
22. A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, Hive, "A warehousing solution over a map-reduce framework", Proc. VLDB Endow, 2(2), 1626– 1629 (2009).
23. A. Lakshman, and P. Malik, "Cassandra: a decentralized structured storage system", ACM SIGOPS Oper. Syst. Rev. 44(2), 35–40 (2010).
24. M. Armbrust, R.S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley, X. Meng, T. Kaftan, M.J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: relational data processing in spark", in SIGMOD (2015), pp. 1383–1394.
25. William Clinger (June 1981), "Foundations of Actor Semantics", Mathematics Doctoral Dissertation, MIT, USA.
26. Typesafe. Akka (2016), http://akka.io/, Retrieved on Feb 04, 2018.
27. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, "Storm@ twitter", in Proceedings of the 2014 ACM SIGMOD international conference on Management of data (ACM, 2014), pp. 147–156.

28. Robert M. Metcalfe, and David R. Boggs, “Ethernet: distributed packet switching for local computer networks”, Communications of the ACM, Volume 19 Issue 7, July 1976.
29. Avinash Sodani, “Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor”, Hot Chips 27 Symposium (HCS), 2015 IEEE, 22-25 Aug 2015.
30. Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda, “High Performance RDMA-Based MPI Implementation over InfiniBand”, International Journal of Parallel Programming, June 2004, Volume 32, Issue 3, p.n. 167-198.
31. Collective Communication by Dr. Edger Gabriel in Parallel Computing class of fall, 2014.http://www2.cs.uh.edu/~gabriel/courses/cosc6374_f14/ParCo_17_NonBlockingCollectives.pdf Retrieved on 10 Oct, 2017.
32. https://www.forbes.com/sites/kalevleetaru/2017/05/22/googles-new-180-petaflop-deep-learning-cloud-and-the-future-of-academic-computing/#f805af161f0a. Retrieved on 10 Oct 2017.
33. http://www.worldwidewebsize.com/, Retrieved on 10 Oct 2017.
34. Steven Bird, Ewan Klein, and Edward Loper, “Natural language processing with Python”, O’Reilly Media Inc., First Edition, 2009.
35. https://en.wikipedia.org/wiki/Scikit-learn, Retrieved on 19 Oct 2017.
36. Ole Nielsen, Pypar Home page, 2002–2005, https://sourceforge.net/projects/pypar/files/pypar/pypar_1.9.3/, (2017).
37. Timothy Kaiser, Leesa Brieger, and Sarah Healy. 2006. MYMPI-MPI programming in Python, In PDPTA, Citeseer, p.n. 458–464.

38. SourceForge.net:MPI Python. http://sourceforge.net/projects/pympi, (2006).

39. L. Dalcin, R. Paz, and M. Storti, “MPI for Python”, Journal of Parallel and Distributed Computing, 65, 9 (2005), p.n. 1108–1115.

40. L. Dalcin, R. Paz, M. Storti, and J. D’Elia, “MPI for Python: performance improvements and MPI-2 extensions”, Journal of Parallel and Distributed Computing, 68, 5, p.n. 655–662, 2008.

41. Xing Cai, Hans Petter Langtangen, and Halvard Moe. 2005, “On the performance of the Python programming language for serial and parallel scientific computations”, Scientific Programming 13, 1, p.n. 31–56, 2005.

42. K Jarrod Millman and Michael Aivazis. 2011. Python for scientists and engineers. Computing in Science & Engineering 13, 2, p.n. 9–12, 2011.

43. Matti Bickel, Adrian Knoth, and Mladen Berekovic, “Evaluation of interpreted languages with open MPI”, In European MPI Users’ Group Meeting. Springer, 292–301, 2011.

44. L. Dalcin, R. Paz, and M. Storti. 2005, “MPI for Python”, Journal of Parallel and Distributed Computing, 65, 9, p.n. 1108–1115.

45. L. Dalcin, R. Paz, M. Storti, and J. D’Elia, “MPI for Python: performance improvements and MPI-2 extensions”, Journal of Parallel and Distributed Computing, 68, 5, p.n. 655–662, 2008.

46. Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S,

"Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation", In Proceedings, 11th European PVM/MPI Users' Group Meeting. Budapest, Hungary, p.n. 97–104.

47. Timo Schneider, Robert Gerstenberger, and Torsten Hoefler, "Application-oriented ping-pong benchmarking: how to assess the real communication overheads", Computing, Vol. 96, Issue 4, p.n. 279-292, 2014.
48. Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux, "The NumPy array: a structure for efficient numerical computation", Computing in Science & Engineering 13, 2 (2011), p.n. 22–30.
49. http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband, Retrieved on Dec 08, 2017.
50. Govinda G. Choudhary, "Natural language processing" in American Society for Information Science and Technology, 31 January 2005.
51. James F. Allen, "Natural language processing", Encyclopedia of Computer Science, John Wiley and Sons Ltd. Chichester, UK, ISBN: 0-470-86412-5.
52. "IBM WebSphere Translation Server for Multiplatforms", www.ibm.com, Archived from the original on 2017-11-08, Retrieved 2017-11-08.
53. "Watson Language Translator", www.ibm.com. 2016-11-28, Archived from the original on 2017-11-08, Retrieved 2017-11-08.
54. Language-translator-nodejs: Sample Node.js Application for the IBM Language Translation Service, Watson Developer Cloud, 2017-11-08, Retrieved 2017-11-08.
55. http://spam.abuse.net/overview/whatisspam.shtml, Retrieved on March 22nd 2018.

56. Summarization - Compressing Data into an Informative Representation by Varun Chandola and Vipin Kumar Department of Computer Science, University of Minnesota, MN, USA.
57. Text Summarization in Data Mining Colleen E. Crangle ConverSpeech LLC, 60 Kirby Place, Palo Alto, California 94301, USA, www.converspeech.com.
58. https://en.wikipedia.org/wiki/Question_answering retrieved on March 22nd 2018.
59. Agerri R. et al., "Big data for Natural Language Processing: A streaming approach", Knowledge Based Systems, Vol. 79, May 2015, p.n. 36-42.
60. http://nlp.johnsnowlabs.com/, Retrieved on March 30th 2018.
61. http://text-analytics101.rxnlp.com/2014/11/what-are-n-grams.html, Retrieved on March 30th 2018.
62. http://kavita-ganesan.com/micropinion-generation/#.Wr7BUtPwbMI, Retrieved on March 30th 2018.
63. Rajaraman, A.; Ullman, J.D. (2011), "Data Mining", Mining of Massive Datasets (PDF). pp. 1–17. doi:10.1017/CBO9781139058452.002. ISBN 978-1-139-05845-2.
64. https://en.wikipedia.org/wiki/Message_Passing_Interface Retrieved on April 3rd 2018.