Expedient Modal Decomposition of Massive Datasets Using High Performance Computing Clusters

Masters Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Sankeerth Vyapamakula Sreeramachandra,

Graduate Program in Computer Science and Engineering

The Ohio State University
2018

Master's Committee:

Dr. P. Sadayppan
Dr. Jack J McNamara

Abstract

High-fidelity observations of non-linear dynamical systems that are of practical interest lead to massive data sets which do not fit on a single computing node. Therefore, modal decomposition techniques must be able to exploit the capability of high-performance computing (HPC) facilities. Proper Orthogonal Decomposition and Sparse Coding are two of the commonly used modal decomposition techniques to obtain reduced order models. The goal of the research is to parallelize and implement these algorithms so that they can be used on high-performance computing clusters in order to expedite the process of modal decomposition from massive data sets. However, computation on various machines is associated with high memory usage and significant communication cost. Moreover, the overall computational cost is sensitive to the type of data set and various parameters of the algorithm. Therefore, several strategies are discussed and implementations are developed to address these constraints to perform expedient modal decomposition. Furthermore, a systematic study is performed over multiple data sets to assess the performance and scalability of the implementations.

Dedication

I would like to dedicate this work to my Mother, Father and my Sister who have always been with me and supported me on my decisions.

Acknowledgments

This thesis owes existence to help and inspiration from several people. First, I would like to express my sincere thanks to Dr. Jack McNamara to have shown faith in me and given me the opportunity to work with him. Second, I would like to express my deepest gratitude to Dr. Sadayappan for his constant guidance and technical insight into the problems that we faced. Following his advise, we were able to produce results using various approaches with better efficiency. Third, I would like to thank Dr. Rohit Deshmukh without whose research work on parallelizing algorithms, I would not have had the opportunity to implement it today. I owe most my effort to him who constantly supported me on all aspects and spent his valuable time sharing knowledge and opinions on regular basis. I am grateful for the support, motivation, patience and constant motivation from the above mentioned people without which this work would not have been accomplished by me.

A special thanks to Dr. Ben Grier to spend his valuable time running the test cases on Thunder for us, Shailesh for taking most of the readings to documenting

them to creating presentations and easing my work in various ways.

I would also like to take this opportunity to thank Dr. Abhijith Gogulapati and Dr. Nima Mansouri for attending all the meetings and helping me out with their suggestions. Thanks to Kirk, Emily, Kelsey, Nathan from the group as well who have been very welcoming and a great company in the lab.

My acknowledgment would be incomplete without the help and encouragement from my friends. A huge thank you to Prithvi, Vineeth, Anirudh, Venkat, Kalyan, Karan, Rohit and Pravar. Lastly, I would like to thank my friend Ilsim Shin who has been very close to me and constantly supported me in all my endeavors.

Finally, I want to thank my parents and sister. Without their constant support and encouragement I would not be the person I am today.

Vita

2012	B.E, Electrical and Electronics Engineering, B.M.S College of Engineering, Bangalore
2012 - 2015	Senior Software Engineer, Samsung Research and Development India, Bangalore
2015 - 2016	Software Developer, IBM Software Labs, Bangalore
2017	Graduate Research Associate, Department of Mechanical and Aerospace Engineering, The Ohio State University

Fields of Study

Major Field	Computer Science and Engineering

Table of Contents

List of Tables

List of Figures

List of Symbols

$\boldsymbol{A}$	= correlation matrix of snapshots
$a_{i,j}$	= entries of correlation matrix of snapshots
b_j^i	= the j^{th} element of the i^{th} eigenvector
f_{obj}	= objective function
L	= active (non-zero) number of modes in a snapshot matrix
m	= total number of snapshots in a snapshot matrix
n	= number of variables
n_{dim}	= 2 for two-dimensional flows, and 3 for three-dimensional flows.
n_{node}	= number of nodes
n_{node}^{POD}	= minimum number of nodes required for POD method
n_{node}^{SPC}	= minimum number of nodes required for sparse coding method
n_g	= number of grid points
$\boldsymbol{Q}$	= snapshot matrix
$\boldsymbol{q}$	= fluctuating component of the flow field
$\boldsymbol{S}$	= coefficient matrix
$\boldsymbol{U}$	= left eigenvectors
$\bar{\boldsymbol{U}}$	= mean velocity component
$\boldsymbol{V}$	= eigenvectors
β	= sparsity coefficient
λ	= eigenvalues
$\boldsymbol{\Lambda}$	= eigenvalue matrix

$\boldsymbol{\Phi}$	= matrix containing modes in its columns
$\boldsymbol{\Phi}_i$	= i^{th} mode
$\boldsymbol{\Phi}^{POD}$	= POD modal matrix, $\in \mathbb{R}^{n \times r}$
$\boldsymbol{\Phi}^{SPC}$	= sparse modal matrix, $\in \mathbb{R}^{n \times N}$
$\boldsymbol{p}$	= total number of processes used in computation
$\boldsymbol{p}_{row}$	= number of processes across the row in 1-D process grid
$\boldsymbol{p}_{col}$	= number of processes across the column in 1-D process grid
$\boldsymbol{p}_{max}$	= max number of processes that can be used in computation
$\boldsymbol{m}_p$	= number of snapshots per process
$\boldsymbol{n}_p$	= number of grid points per process
$\boldsymbol{p}_{rank}$	= MPI rank of a process
$\|\cdot\|$	= L2 norm, equal to the sum of squares of the entries in $(\cdot)$
$\|\cdot\|_0$	= L0 norm, equal to the number of non-zero entries in $(\cdot)$

Supscripts

(i)	= i^{th} iteration
$L0$	= i^{th} L0 penalty approach
POD	= matrices corresponding to the proper orthogonal decomposition
T	= Transpose of matrix

Subscripts

i, j, k	= indices used to represent rows and columns of a matrix

Chapter 1

Introduction

1.1 Problem Statement

The solution trajectories of complex physical systems are multi-dimensional, leading to massive observation data sets of statistical significance. Comprehending and using data sets generated from such complex systems is a non-trivial and computationally intensive task. Modal decomposition is an important step towards analysis, compression and reduced order modeling of such dynamical systems. Modal decomposition is a process of decomposing a large observation or data matrix into a small set of basis vectors and corresponding coefficients. Such massive data sets generated from physical systems generally do not fit on a single computer and require large computing power to carry out modal decomposition. Recent advancement in computing hardware and development of parallel algorithms has allowed taking a step further in making efficient and accurate predictions. Therefore, modal decomposition techniques must be able to operate on large data sets by

leveraging high-performance computing clusters. One of the widely used modal decomposition techniques is Proper Orthogonal Decomposition (POD), or Principal Components Analysis (PCA) [2] [8] [9]. The approach is based on identifying and ordering principal components in observed data. Another promising approach to obtain reduced order models is Sparse Coding [11] [12] which was found to perform better for canonical problems [1]. However, a drawback of the sparse coding approach is the significant increase in computational cost to decompose large data sets [1]. This issue limits the usage of the existing algorithm of sparse coding, to relatively simple, low-dimensional systems. Parallel implementation of these modal decomposition techniques are developed to leverage high-performance computing hardware and expedite the process of obtaining reduced order models. Several implementation strategies are proposed, assessed and discussed. Performance is characterized in terms of efficiency of execution, memory consumed and communication cost incurred. Scalability of the algorithms is assessed by varying the parameters of the algorithm.

1.2 Literature Review

Dimensionality reduction is a process converting a set of data having vast dimensions into data with lesser dimensions ensuring that it conveys similar information concisely. Proper Orthogonal Decomposition (POD) or Principal Component Analysis(PCA) is one of the commonly used algorithms for dimensionality reduc-

tion. Principal Component Analysis is widely applied in fields such as Neuroscience [15], Bioinformatics [16], E-Commerce [17], Data Mining [18], Image Processing [25] and many more. To address the increasing computational costs and memory requirements of scientific data analysis, a parallel and scalable solution is necessary. There have been several attempts at performing PCA in parallel [21–24] and many open source libraries and frameworks such as Hadoop [28], Spark [29], R Statistical Package [30, 31], Mahout-PCA [26], MLlib-PCA [27], MatlabMPI [32] that provide parallel implementation of PCA on large data sets. However, there are several limitations in using these open source implementations. In-memory capability can become a bottleneck with cost-efficient processing of large data sets as in-memory is quite expensive. Memory consumption is very high and it is not handled in a user-friendly way. Especially with Apache Spark which requires a huge amount of RAM to execute in-memory, thus cost is high. Portability of code on various HPC clusters becomes a challenge. These big data frameworks are usually associated with high startup overhead and latency for running several jobs on large data sets. Therefore, developing our own framework provides flexibility in usage to perform certain steps in the algorithm differently to improve the efficiency of execution for our problem.

One of the works provides a detailed analysis of the bottlenecks in scalability of PCA in distributed environment [33] and provides good insight on choosing the appropriate software library for executing PCA in parallel. It also stresses on high communication cost incurred for PCA on big data sets and cubic complexity

of SVD operation. The study proposes sPCA [34] that uses Probabilistic PCA to reduce communication complexity on matrices as large as $1.26B \times 71.5K$ (96 GB of non zeros) of Twitter data and $160M \times 128$ (82 GB) of image data apart from others. However, their approach is different from the standard PCA which we use and hence the output would be different. The study uses stochastic SVD that is better suited for sparse matrices.

Another recent work [35] provides a novel Distributed PCA algorithm and a good insight on its communication complexity with a well established formulation for the lower-bound cost. The study proposes that distributing data in a column-partition model, i.e. columns of the main matrix distributed, performs better in terms of communication.

Trade-off using external libraries such as Apache Spark to perform linear algebra operations over traditional C and MPI implementation on HPC platforms are discussed in [36]. It emphasizes efficient parallel I/O as one of the critical requirements for better performance of data analysis on a massive scale. PCA is computed on massive data sets with dimensions, $26,542,080 \times 81,600$ (16 TB), $6,349,676 \times 46,715$ (2.2 TB).

One of the more recent studies [14] performs expedient modal decomposition on large data sets using parallel implementation of POD and serial implementation of the sparse coding algorithm. The research presented in this thesis is an extension to the work done in the paper [14]. The study performs POD on massive data sets with dimensions, $500K \times 10K$ (42 GB), $18M \times 3000$ (162 GB) and $57M \times 810$ (370

GB). This work uses 2D data distribution among processes which impacts the I/O time severely. The study leverages HPC hardware to implement POD in parallel while sparse coding is still computed in serial.

1.3 Objectives

The overall objective of this research is to parallelize modal decomposition algorithms, perform a systematic characterization of various parallelization techniques and assess the performance on various data sets. Carrying out modal decomposition on massive data sets to obtain reduced order models is a memory intensive operation. Therefore, load balancing, through efficient distribution of data among nodes, is critical in order to ensure uniform memory usage and avoid overload. Several implementation strategies are carried out since decomposition of such massive data sets is a computationally expensive operation. Since these algorithms involve intensive I/O (input/output) operations, it is crucial to perform the file reads and writes in parallel to avoid idle state of processors and wastage of resources. Communication among processes to decompose large data sets involve high latencies and implementation must ensure to minimize the overhead to expedite the modal decomposition process. Therefore, it is important to develop a framework to carry out modal decomposition and address these challenges.

The specific objectives of the study are:

- Parallelize and implement the algorithms to perform expedient modal de-

composition of massive data sets on high-performance computing clusters.

- Demonstrate and assess the performance of parallelized algorithms on different data sets and hardware.

1.4 Key Contributions

- Developed a framework to carry out modal decomposition that is stable, robust and easy to maintain. The code performs efficient computation to obtain reduced order models with high accuracy.

- The code developed is compatible with various high-performance clusters. The code uses stable open source libraries to ensure portability of code on various hardware with little or no changes at all.

Chapter 2

Modal Decomposition Algorithms

This chapter focuses on the description of two algorithms used in carrying out modal decomposition to obtain reduced order models of discrete non-linear dynamical systems. Implementation of the algorithms are described by categorizing each one into three phases, namely, preprocessing, base and postprocessing.

2.1 Proper Orthogonal Decomposition

As mentioned previously, Proper Orthogonal Decomposition or Principal Component Analysis is a widely used technique in modal decomposition by identifying or ordering principal components in observed data.

Preprocessing phase begins with the construction of initial snapshot matrix $\boldsymbol{Q} = [\boldsymbol{q_1 q_2} \cdots \boldsymbol{q_i} \cdots \boldsymbol{q_m}]$ by stacking system observations $\boldsymbol{q_i}$ in columns.

Next, the fluctuating component $\boldsymbol{q}\left(\boldsymbol{x}, t\right)$ is computed by subtracting the initial

snapshot matrix $\boldsymbol{Q}$ from the mean $\bar{\boldsymbol{U}}(\mathbf{x})$ to center the data around origin.

$$\boldsymbol{q}\left(\boldsymbol{x}, t\right) = \boldsymbol{Q} - \bar{\boldsymbol{U}}(\mathbf{x}) \tag{2.1}$$

Since the data is collected from discrete observations (in space and time) of a non-linear dynamical system, quantities at each of the points in the fluctuating component are multiplied by the corresponding mesh volume $\bar{\boldsymbol{v}}$. With this, pre-processing phase is completed where the final snapshot matrix $\boldsymbol{Q}_F$ is:

$$\boldsymbol{Q}_F = \boldsymbol{q}\left(\boldsymbol{x}, t\right) \times \bar{\boldsymbol{v}} \tag{2.2}$$

Now, the algorithm proceeds to the base phase to extract a set of POD modes $\boldsymbol{\Phi}_i$ using the Method of snapshots [9] from the final snapshot matrix $\boldsymbol{Q}_F$ previously computed.

First, the correlation matrix is computed as:

$$\boldsymbol{A} = \boldsymbol{Q}_F^T \boldsymbol{Q}_F \tag{2.3}$$

Next, eigen-decomposition of the correlation matrix $\boldsymbol{A}$ is carried out using Singular Value Decomposition (SVD) [2, 9]:

$$\boldsymbol{A}\boldsymbol{V} = \boldsymbol{\Lambda}\boldsymbol{V} \tag{2.4}$$

where the entries of the correlation matrix $\boldsymbol{A}$ are $a_{i,j} = (\boldsymbol{q}_i, \boldsymbol{q}_j)$, and the rank of $\boldsymbol{A}$ is r. The eigenvalues corresponding to the eigen vectors $\boldsymbol{V}$ are arranged on the diagonal of $\boldsymbol{\Lambda}$. As many POD modes as the rank of $\boldsymbol{A}$ can be computed. Eigen-decomposition is performed on the correlation matrix $\boldsymbol{A}$ instead of the snapshot matrix $\boldsymbol{Q}$ since it is computationally very expensive to perform decomposition on a snapshot matrix that is tall($n_g >> m$) as the rank r can at max be m.

The set of r orthogonal POD modes $\boldsymbol{\Phi}_i$ are given by:

$$\boldsymbol{\Phi}_i = \frac{1}{\sqrt{\lambda_i}} \sum_{j=1}^{m} b_j^i \boldsymbol{q}_j \tag{2.5}$$

where λ_i are the eigenvalues of the correlation matrix $\boldsymbol{A}$ in descending order, $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_r \geq 0$, and b_j^i is the jth element of the ith eigen vector $\boldsymbol{V}_i$. The singular value corresponding to a POD mode is directly proportional to the energy contained in that mode [2] . Thus, the complete set of POD modes describes the snapshot matrix in entirety [2]. Consequentially, if the snapshot matrix is assumed to contain all the dimensions of the system, the complete set of POD modes will also contain all the information. Dimensionality reduction using the POD modes is obtained by truncating the POD set, such that the truncated set contains the desired proportion of the total energy.

The last step in the base phase is to compute POD coefficients $\boldsymbol{S}$ given by:

$$\boldsymbol{S}_i^j = (\boldsymbol{\Phi}_i, \boldsymbol{q}_j) \tag{2.6}$$

Finally, the algorithm proceeds to the postprocessing phase where RMS error is computed to measure the efficiency of decomposition and accuracy with which the snapshot matrix can be reconstructed:

$$\min_{\boldsymbol{\Phi},\boldsymbol{S}} \frac{1}{2} \|\boldsymbol{Q_F} - \boldsymbol{\Phi S}\|_F^2 \tag{2.7}$$

where $\|\cdot\|_F^2$ is the grid-independent square Frobenius norm of '$\cdot$'. Equation 2.7 provides the error in the representation of the snapshot matrix by the modes $\boldsymbol{\Phi}$. The process of computing the POD modes is summarized in Algorithm 1.

Algorithm 1: Proper Orthogonal Decomposition algorithm

Input : Snapshot matrix $\boldsymbol{Q}$ of size $n_g \times m$, number of pod modes to be compute (N)

Output: POD modes ($\boldsymbol{\Phi}$), POD coefficient matrix ($\boldsymbol{S}$)

1 Compute fluctuating component by subtracting mean from snapshot matrix: $\boldsymbol{q}(\boldsymbol{x},t) = \boldsymbol{Q} - \bar{\boldsymbol{U}}$

2 Compute final snapshot matrix by multiplying with mesh volume from grid: $\boldsymbol{Q}_F = \boldsymbol{q}(\boldsymbol{x},t) \times \bar{\boldsymbol{v}}$

3 Compute the correlation matrix $\boldsymbol{A}$: $\boldsymbol{A} = \boldsymbol{Q}_F^T \boldsymbol{Q}_F$

4 Carry out the eigen-decomposition of the correlation matrix: $\boldsymbol{AV} = \boldsymbol{\Lambda V}$

5 Compute the POD modes corresponding to the non-zero singular values (λ): $\boldsymbol{\Phi}_i = \frac{1}{\sqrt{\lambda_i}} \sum_{j=1}^{m} b_j^i \boldsymbol{q}_j$
By default, the POD modes are arranged in the descending order of singular values. Truncate the POD modes to a desired number.

6 Compute POD coefficients from POD modes and snapshot matrix: $\boldsymbol{S}_i^j = (\boldsymbol{\Phi}_i, \boldsymbol{q}_j)$

7 Compute RMS error to check accuracy with which snapshot matrix can be reconstructed: $\min_{\boldsymbol{\Phi},\boldsymbol{S}} \frac{1}{2} \|\boldsymbol{Q} - \boldsymbol{\Phi S}\|_F^2$

2.2 Sparse Coding

Sparse Coding is an alternate approach to POD for carrying out modal decomposition. This approach generates a finite dictionary of modes in which only a subset are active for a given time window. Sparse coding is associated with a high computational cost. To overcome this, the dimensionality of the problem is reduced by first factorizing the snapshot matrix into POD modes and POD coefficients [1].

$$\boldsymbol{Q} = \boldsymbol{\Phi}^{\boldsymbol{POD}} \boldsymbol{S}^{\boldsymbol{POD}} \tag{2.8}$$

where $\boldsymbol{S}^{\boldsymbol{POD}} \in \mathbb{R}^{r \times m}$ is the POD coefficient matrix, and r denotes the rank of $\boldsymbol{Q}$ and $\boldsymbol{\Phi}^{\boldsymbol{POD}}$. The rank r is less than the number of snapshots m if the snapshot matrix $\boldsymbol{Q}$ is rank-deficient. Here, all the POD modes are included to retain all the features of the snapshot matrix. Now, a set of sparse modes $\boldsymbol{X}$ is solved for using sparse coding.

Preprocessing phase begins with initializing the set of sparse modes $\boldsymbol{X}$ with $\boldsymbol{S}^{\boldsymbol{POD}}$ up to the input number of modes N that needs to be computed. This initialization is done by summing over two consecutive columns of $\boldsymbol{S}^{\boldsymbol{POD}}$. This ensures that $\boldsymbol{X}$ is not initialized randomly and same modes are not chosen to be active during the Batch-OMP process, else erroneous values are produced. The sparse modes $\boldsymbol{X}_i$ are then normalized such that $\|\boldsymbol{X}_i\| = 1$.

Next, the algorithm proceeds to the base phase where sparse coding approach [1, 7]

aims to solve:

$$\min_{\boldsymbol{S},\boldsymbol{\Phi}} \quad \|\boldsymbol{Q} - \boldsymbol{\Phi}\boldsymbol{S}\|_F^2, \quad \text{subject to} \quad \forall i, \|\boldsymbol{s}_i\|_0 \leq L \tag{2.9}$$

Here, the L0 norm $\|.\|_0$ counts the number of non-zero entries in $\boldsymbol{s}_i$. By enforcing $\|\boldsymbol{s}_i\|_0 \leq L$ in Eq. 2.9, the number of non-zero entries in $\boldsymbol{s}_i$ is constrained to be less than or equal to a constant L. Sparsity is defined as:

$$\text{Sparsity} = \frac{(N-L)\times m}{N \times m} \times 100\% = (1 - \frac{L}{N}) \times 100\% \tag{2.10}$$

where L is the active number of basis vectors or modes, out of N input number of sparse modes, in any given snapshot. Thus using Eq. 2.10, the sparsity and dimension is directly controlled by specifying L and N, respectively. Solving Eq. 2.9 is challenging since its solution is non-convex. Instead it is alternatively minimized following standard sparse coding procedures over matrices $\boldsymbol{S}$ and $\boldsymbol{\Phi}$:

$$\min_{\boldsymbol{s}_i} \quad \|\boldsymbol{q}_i - \boldsymbol{\Phi}\boldsymbol{s}_i\|_2^2, \quad \text{subject to} \quad \forall i, \|\boldsymbol{s}_i\|_0 \leq L \tag{2.11}$$

and

$$\min_{\boldsymbol{\Phi}} \quad \|\boldsymbol{Q} - \boldsymbol{\Phi}\boldsymbol{S}\|_F^2, \quad \text{subject to} \quad \forall i, \|\boldsymbol{\Phi}_i\| \leq 1 \tag{2.12}$$

Equation 2.11 determines a sparse representation of the each snapshot based on the current sparse modes. Greedy algorithm such as Batch-OMP [6] is used to learn $\boldsymbol{s}_i$. The Batch-OMP implementation uses Moore-Penrose pseudo-inverse [13]

which in turn uses Singular Value Decomposition(SVD) [7]. Equation 2.12 is an optimization over sparse modes across the whole snapshot matrix $\boldsymbol{Q}$ at once. It is a constrained least-squares problem that is solved by K-SVD algorithm [7]. The modes $\boldsymbol{X}$ and the corresponding coefficient matrix $\boldsymbol{S}$ are updated iteratively using K-SVD algorithm. Next step clears the inactive modes using certain predefined criteria. Thus, each convergence loop begins with solving for sparse coefficient matrix and then updating the modes iteratively and finally clearing the inactive modes. This process is finally stopped once the modes are converged, i.e.,

$$|f_{obj}(i) - f_{obj}(i-1)| < \epsilon \tag{2.13}$$

where

$$f_{obj}(i) = \left\| \boldsymbol{S^{POD}} - \boldsymbol{X}\boldsymbol{S} \right\|_F^2 \tag{2.14}$$

Here, i denotes the i^{th} iteration and ϵ is a pre-defined convergence criterion, set in this work as $\epsilon = 10^{-3}$. The resulting sparse modes $\boldsymbol{X}$ are projected onto the vector space of the snapshot matrix to obtain the final sparse modes:

$$\boldsymbol{\Phi} = \boldsymbol{\Phi}^{POD}\boldsymbol{X} \tag{2.15}$$

Finally, the algorithm proceeds to the postprocessing phase where RMS error is computed to test the accuracy of reconstruction using sparse modes $\boldsymbol{\Phi}$ previously

computed. First, corrected sparse coefficient matrix $\boldsymbol{S}_{corr}^{SPC}$ is computed as:

$$\boldsymbol{S}_{corr}^{SPC} = pinv(\boldsymbol{\Phi})\boldsymbol{Q} \tag{2.16}$$

where $pinv$ is Moore-Penrose pseudo-inverse [13] function that uses SVD internally. Next, reconstructed snapshot matrix $\boldsymbol{Q}_{REC}$ is computed as:

$$\boldsymbol{Q}_{REC} = \boldsymbol{\Phi}\boldsymbol{S}_{corr}^{SPC} \tag{2.17}$$

where $\boldsymbol{\Phi}$ is the sparse modes computed using Eq. 2.15 and $\boldsymbol{S}_{corr}^{SPC}$ is the corrected sparse coefficients using Eq. 2.16. Finally, RMS error is computed as:

$$\min \frac{1}{2} \left\| \boldsymbol{Q} - \boldsymbol{Q}_{REC} \right\|_F^2 \tag{2.18}$$

where $\boldsymbol{Q}_{REC}$ is the reconstructed snapshot matrix using Eq. 2.17

Algorithm 2: Greedy sparse coding algorithm using the L0 penalty (referred to as Sparse^{L0} algorithm)

Input : Snapshot matrix $\boldsymbol{Q}$ of size $n \times m$, number of sparse modes to be compute (N), number of active modes per snapshot (L), convergence criterion (ϵ_c)

Output: Sparse modes ($\boldsymbol{\Phi}$), sparse coefficient matrix ($\boldsymbol{S}$)

Decompose the snapshot matrix $\boldsymbol{Q}$ into the POD modes and POD coefficients:
$\boldsymbol{Q} = \boldsymbol{\Phi}^{\boldsymbol{POD}} \boldsymbol{S}^{\boldsymbol{POD}}$

Initialize convergence criterion (ϵ_c):
$\sum_{i=1}^{m} \sum_{j=1}^{N} \|\boldsymbol{S}_{ij}^{POD}\|^2$

Initialize the sparse modes $\boldsymbol{X}$ by summing up consecutive columns of $\boldsymbol{S}^{\boldsymbol{POD}}$

Normalize the computed sparse modes:
$$\boldsymbol{X}_i = \frac{\boldsymbol{X}_i}{\|\boldsymbol{X}_i\|_F^2}$$

Iteration number, $i = 1$

while $\epsilon \geq \epsilon_c$ **do**

  Solve for sparse coefficient matrix $\boldsymbol{S}^{SPC}$ using batch-OMP:
  $\min_{\boldsymbol{S}} \left\|\boldsymbol{S}^{\boldsymbol{POD}} - \boldsymbol{X}\boldsymbol{S}\right\|_F^2, \quad \text{subject to} \quad \forall i, \|\boldsymbol{s}_i\|_0 \leq L$

  Update the sparse modes $\boldsymbol{X}$ of the POD coefficient matrix $\boldsymbol{S}^{\boldsymbol{POD}}$ iteratively using K-SVD algorithm

  Clear inactive modes from dictionary by replacing with normalized value of $\boldsymbol{S}^{POD}$ by finding the pos as:
  $\boldsymbol{Er} = \sum_{k=1}^{m} \frac{1}{2} \left(\left\|\boldsymbol{S}_{\boldsymbol{k}}^{\boldsymbol{POD}} - \boldsymbol{X}\boldsymbol{S}_k\right\|_F^2 \right)$
  where pos = $\max(\boldsymbol{Er})$ and resetting the value as $\boldsymbol{Er}_{pos} = 0$

  Evaluate the objective function:
  $f_{obj}(i) = \left\|\boldsymbol{S}^{\boldsymbol{POD}} - \boldsymbol{X}\boldsymbol{S}\right\|_F^2$

  Check for convergence, $\epsilon = |f_{obj}(i) - f_{obj}(i-1)|$

  Update iteration number $i = i + 1$

end

Compute sparse modes for the snapshot matrix $\boldsymbol{Q}$:
$\boldsymbol{\Phi} = \boldsymbol{\Phi}^{\boldsymbol{POD}} \boldsymbol{X}$

Compute corrected sparse coefficient matrix:
$\boldsymbol{S}_{corr}^{SPC} = pinv(\boldsymbol{\Phi})\boldsymbol{Q}$

Compute reconstructed snapshot matrix:
$\boldsymbol{Q}_{REC} = \boldsymbol{\Phi}\boldsymbol{S}_{corr}^{SPC}$

Compute RMS error as:
$\min \frac{1}{2} \|\boldsymbol{Q} - \boldsymbol{Q}_{REC}\|_F^2$

Chapter 3

Implementation

This chapter focuses on the implementation of Proper Orthogonal Decomposition(POD) and Sparse Coding algorithms in parallel. To implement the algorithms in parallel, ScaLapack [4] library is used for linear algebra operations such as matrix multiplication, singular value decomposition(SVD) and matrix transpose. ScaLapack library internally uses Message Passing Interface(MPI) [3] for communication between processes. The processes used in computation are represented as a two-dimensional rectangular grid, or process grid [4]. The process grid has $\boldsymbol{p}_{row}$ process rows and $\boldsymbol{p}_{col}$ process columns, where the total number of processes, $\boldsymbol{p}$, involved in computation is equal to $\boldsymbol{p}_{row} \times \boldsymbol{p}_{col}$ and each process in the grid is referenced as $\boldsymbol{P}_{p_{row},p_{col}}$. If the process distribution is represented as an array, similar to a single row or column in the two-dimensional grid, it is called as one-dimensional process distribution. All global matrices must be distributed among process in one-dimensional(1D) or two-dimensional(2D) process grid fashion prior to the invocation of ScaLapack routines. To distribute the global snapshot matrix among

processes in 2D process grid, a single process (master process) needs to read the snapshots and distribute the data to other processes using ScaLapack communication routines. Since there can be several thousand snapshots and millions of data points in each snapshot, this kind of distribution is not scalable for our purpose as most of the processes are inactive or idle while master process reads the data from the snapshots. Moreover, a high communication cost is associated with distributing the global snapshot matrix with the master process communicating the data to every other process in the 2D process grid. In contrast, 1D process distribution allows each process to read the snapshots independent of other processes. This ensures that no process is inactive while reading the snapshots and zero communication cost is associated with the distribution of the snapshot matrix. Hence, for our parallel implementation, 1D process distribution is chosen over 2D process grid to distribute the data from global matrices.

3.1 Parallelizing POD

As mentioned earlier, parallel implementation of POD is necessary for decomposing massive data sets which cannot be fit on a single process or a node in a high performance computing cluster. There are three implementations of POD namely, $PODColCyclic$, $PODRowCyclic$ and $PODColCyclicTransposed$. Each implementation differs in data distribution of the global snapshot matrix. Furthermore, global matrices generated during the base phase follows the same data distribu-

tion. Therefore, only the preprocessing phase is described and its complexity and communication cost is tabulated corresponding to each implementation.

The first step in the preprocessing phase is the same for all implementations where the master process (rank 0 in MPI) reads the grid or mesh file and computes the mesh volume that will be used later.

3.1.1 PODColCyclic Implementation

In this implementation, processes are distributed across the columns of the process grid in a cyclic manner where data points in each column belong to a single process as shown in Fig. 3.1. The number of processes across the row and column of the process grid is given by:

$$\boldsymbol{p}_{col} = \boldsymbol{p} \text{ and } \boldsymbol{p}_{row} = 1$$

where, $\boldsymbol{p}$ is the total number of processes used in computation.

Each process initially identifies the number of snapshots it has to read based on a simple formula:

$$\begin{aligned} \boldsymbol{m}_p &= \boldsymbol{m}/\boldsymbol{p} \\ remainder &= \boldsymbol{m}\%\boldsymbol{p} \\ if(\boldsymbol{p}_{rank} &< remainder) \;\; \boldsymbol{m}_p += 1 \end{aligned} \tag{3.1}$$

where, $\boldsymbol{m}_p$ is the number of snapshots in each process, $\boldsymbol{m}$ is the total number of snapshots, $\boldsymbol{p}_{rank}$ is the rank of the process which ranges from 0 to $\boldsymbol{p}$ - 1.

For example, let the total number of snapshots, $\boldsymbol{m} = 5$ and the total number of processes, $\boldsymbol{p} = 3$. Initially, the snapshots would be distributed among 3 processes equally with 1 snapshot per process and remainder would be computed as 2. The remainder of 2 is then distributed among all the processes whose MPI rank is less than the remainder. There are 2 processes with rank $\boldsymbol{p}_{rank}$, 0 and 1 which are less than the remainder. Therefore the final distribution would be:

$$\boldsymbol{m}_0 = 2 \quad \boldsymbol{m}_1 = 2 \quad \boldsymbol{m}_2 = 1$$

where, $\boldsymbol{m}_0$, $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ are the number of snapshots in rank 0, 1 and 2 respectively.

The snapshots are read in a cyclic order, i.e. there is a difference of a number of processes, $\boldsymbol{p}$, between consecutive snapshots read by the same process as illustrated in Fig. 3.1:

P_0	P_1	P_2	P_0	P_1
Q_1^1	Q_2^1	Q_3^1	Q_4^1	Q_5^1
Q_1^2	Q_2^2	Q_3^2	Q_4^2	Q_5^2
Q_1^3	Q_2^3	Q_3^3	Q_4^3	Q_5^3
Q_1^4	Q_2^4	Q_3^4	Q_4^4	Q_5^4
Q_1^5	Q_2^5	Q_3^5	Q_4^5	Q_5^5
Q_1^6	Q_2^6	Q_3^6	Q_4^6	Q_5^6
Q_1^7	Q_2^7	Q_3^7	Q_4^7	Q_5^7
Q_1^8	Q_2^8	Q_3^8	Q_4^8	Q_5^8

Figure 3.1: PODColCyclic data distribution among processes.

where, Q_i^j is j^{th} data point in i^{th} snapshot.

Next, mean of the snapshot matrix is computed in following steps:

Table 3.1: Analysis of preprocessing phase in PODColCyclic implementation

Operation	Order of Complexity (O)	Communication cost	Memory per process
Compute $\bar{U}$	$(m * n + n)$	$t_s \log p + t_w n(p-1)$	n
$\boldsymbol{q}(\boldsymbol{x}, t) = \boldsymbol{Q} - \bar{\boldsymbol{U}}$	$(m * n)$	-	$m_p \times n$
$\boldsymbol{Q}_F = \boldsymbol{q}(\boldsymbol{x}, t) \times \bar{\boldsymbol{v}}$	$(m * n)$	$(t_s + nt_w) \log p$	$m_p \times n$

1. Sum of the data points in the local snapshot matrix is computed within each process as:

$$\bar{\boldsymbol{U}} = \sum_{i=1}^{m_p} \sum_{j=1}^{n} \boldsymbol{Q}_i^j$$

2. MPI Allreduce is performed to obtain the sum of all the data points along the row in the snapshot matrix within each process.

3. The sum computed must be divided by the total number of snapshots $\boldsymbol{m}$ to obtain the final mean $\bar{\boldsymbol{U}}$.

$$\bar{\boldsymbol{U}} = \forall i \, \bar{\boldsymbol{U}}_{\boldsymbol{i}}/m$$

Fluctuating component is then computed by subtracting the mean from the local snapshot matrix in each processor as shown in Eq. 2.1.

Mesh volume computed at the start by the master process is then distributed to all the processes using MPI Broadcast operation, $MPIBcast(\bar{\boldsymbol{v}})$.

Next, mesh volume at each process is multiplied by the fluctuating component at each process to obtain the final snapshot matrix as shown in Eq. 2.2.

Advantages

1. Reading of the snapshots is fairly trivial as each process would know exactly which snapshot to read and how many snapshots to read.

2. Writing POD modes and coefficients to files can be performed in parallel as each process would have all the necessary data.

Disadvantages

1. The maximum number of processes used in the preprocessing phase is equal to the total number of snapshots i.e. $\boldsymbol{p_{max} = m - 1}$, in which case each snapshot is read by 1 process.

2. The maximum number of data points in a snapshot is limited by the size of an integer, approximately 2.14 billion, since ScaLapack subroutines take integer as a parameter for number of rows and columns in the global snapshot matrix, else an integer overflow error is encountered.

Since the memory of a single snapshot used in $PODColCyclic$ is limited by approximately 8.5 GB (2.14 billion data points $\times$ 4 bytes), current implementation does not possess the ability to decompose the snapshot matrix of any given size. Hence, an alternate implementation was developed that is described next.

3.1.2 PODRowCyclic Implementation

In this implementation, processes are distributed across the rows of the process grid in a cyclic manner where data points in each row belong to a single process as shown in Fig. 3.1. Such a distribution allows decomposition of large snapshots with the number of data points greater than the size of an integer (approximately 2.14 billion). The number of processes across the row and column of the process grid is given by:

$$\boldsymbol{p}_{row} = \boldsymbol{p} \text{ and } \boldsymbol{p}_{col} = 1$$

where, $\boldsymbol{p}$ is the total number of processes used in the computation.

Each process initially identifies the number of data points in a snapshot it has to read based on a simple formula:

$$\begin{aligned} \boldsymbol{n}_p &= \boldsymbol{n}/\boldsymbol{p} \\ remainder &= \boldsymbol{n}\%\boldsymbol{p} \\ if(\boldsymbol{p}_{rank} < remainder) \;\; \boldsymbol{n}_p &+= 1 \end{aligned} \tag{3.2}$$

where, $\boldsymbol{n}_p$ is the number of data points from a single snapshot in each process, $\boldsymbol{n}$ is the total number of data points in a snapshot, $\boldsymbol{p}_{rank}$ is the rank of the process that ranges from 0 to $\boldsymbol{p}$ - 1.

For example, let the total number of snapshots, $\boldsymbol{m} = 5$ and the total number of processes, $\boldsymbol{p} = 3$. Let the total number of data points in each snapshot, $\boldsymbol{n} = 8$. Initially, the data points in each snapshot would be distributed among 3 processes

equally with 2 data points per process and remainder would be computed as 2. The remainder of 2 is then distributed among all the processes whose MPI rank is less than the remainder. There are two processes with rank $\boldsymbol{p}_{rank}$ 0 and 1 which are less than the remainder. Therefore the final distribution would be:

$$\boldsymbol{n}_0 = 3, \boldsymbol{n}_1 = 3, \boldsymbol{n}_2 = 2,$$

where, $\boldsymbol{n}_0$, $\boldsymbol{n}_1$ and $\boldsymbol{n}_2$ are the number of data points of a snapshot in MPI rank 0, 1, and 2 respectively.

The data points in a snapshot are read in a cyclic order, i.e. there is a difference of number of processes, $\boldsymbol{p}$, between two consecutive data points read by the same process. This is better illustrated in Fig. 3.2:

P_0	Q_1^1	Q_2^1	Q_3^1	Q_4^1	Q_5^1
P_1	Q_1^2	Q_2^2	Q_3^2	Q_4^2	Q_5^2
P_2	Q_1^3	Q_2^3	Q_3^3	Q_4^3	Q_5^3
P_0	Q_1^4	Q_2^4	Q_3^4	Q_4^4	Q_5^4
P_1	Q_1^5	Q_2^5	Q_3^5	Q_4^5	Q_5^5
P_2	Q_1^6	Q_2^6	Q_3^6	Q_4^6	Q_5^6
P_0	Q_1^7	Q_2^7	Q_3^7	Q_4^7	Q_5^7
P_1	Q_1^8	Q_2^8	Q_3^8	Q_4^8	Q_5^8

Figure 3.2: PODRowCyclic data distribution among processes.

where, Q_i^j is j^{th} data point in i^{th} snapshot.

Next, mean of the snapshot matrix is computed in following steps:

1. Since each process has a part of every snapshot, all corresponding data points of every snapshot can be summed up to obtain the local sum corresponding

Table 3.2: Analysis of preprocessing phase in PODRowCyclic implementation

Operation	Order of Complexity ($\boldsymbol{O}$)	Communication cost	Memory per process
Compute $\bar{\boldsymbol{U}}$	$(m * n + n)$	-	n
$\boldsymbol{q}(\boldsymbol{x}, t) = \boldsymbol{Q} - \bar{\boldsymbol{U}}$	$(m * n)$	-	$n_p \times m$
$\boldsymbol{Q}_F = \boldsymbol{q}(\boldsymbol{x}, t) \times \bar{\boldsymbol{v}}$	$(m * n)$	$(t_s + nt_w) \log p$	$n_p \times m$

to the data point.

$$\bar{\boldsymbol{U}} = \sum_{i=1}^{m} \sum_{j=1}^{n_p} \boldsymbol{Q}_i^j$$

2. The local sum computed must the be divided by the total number of snapshots $\boldsymbol{m}$ to obtain the final mean $\bar{\boldsymbol{U}}$.

$$\bar{\boldsymbol{U}} = \forall i \, \bar{\boldsymbol{U}}_{\boldsymbol{i}}/m$$

Fluctuating component is then computed by subtracting the mean from the local snapshot matrix in each process as shown in Eq. 2.1.

Mesh volume computed at the start by the master process is then distributed to all the processes using MPI Broadcast operation, $MPIBcast(\bar{\boldsymbol{v}})$.

Next, mesh volume at each process in multiplied by the fluctuating component at each process to obtain the final snapshot matrix as shown in Eq. 2.2.

Advantages

1. The maximum number of processes used, $\boldsymbol{p}_{max} = \boldsymbol{n}$, is equal to total number of data points in the snapshot. Thus, very large snapshots can be decom-

posed using ScaLapack routines by just increasing the number of processes in computation which addresses the limitation of the previous implementation.

Disadvantages

1. Poor caching is achieved while reading snapshots since the data points are not consecutive. This would result in a large number of cache misses and effectively increases I/O read time.

2. Writing POD modes and coefficients to output is difficult since each column of the global matrix is distributed among the processes. Thus, a $transpose$ of the global matrix is required to ensure an entire column is in a single process before writing it to a file.

3.1.3 PODColCyclicTransposed Implementation

To address the limitations of the maximum size of a snapshot that can be used in $PODColCyclic$, 3.1.3, and poor caching of data in $PODRowCyclic$ implementation, 3.1.2, $PODColCyclicTransposed$ implementation was developed. In this implementation, processes are distributed across the columns of the 1D process grid with distribution as:

$$\boldsymbol{p}_{col} = \boldsymbol{p} \text{ and } \boldsymbol{p}_{row} = 1$$

where, $\boldsymbol{p}$ is the total number of processes used in computation.

This is the same as the distribution used in $PODColCyclic$, 3.1.3. Therefore, the number of snapshots read by a process is identified by the Eq. 3.1.

For example, let the total number of snapshots, $\boldsymbol{m} = 3$ and the total number of processes, $\boldsymbol{p} = 3$. Initially, 1 snapshot would be distributed among 3 processes and remainder would be computed as 0. There is no snapshot left to be distributed further and the final distribution would be:

$$\boldsymbol{m}_0 = 1 \quad \boldsymbol{m}_1 = 1 \quad \boldsymbol{m}_2 = 1$$

where, $\boldsymbol{m}_0$, $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ are the number of snapshots in rank 0, 1 and 2 respectively.

The snapshots are read in cyclic order, i.e. there is a difference of number of processes, $\boldsymbol{p}$, between consecutive snapshots read by the same process as in Fig. 3.3:

P_0	P_1	P_2	P_3	P_4	P_5
Q_1^1	Q_2^1	Q_3^1			
Q_1^2	Q_2^2	Q_3^2			
Q_1^3	Q_2^3	Q_3^3			
Q_1^4	Q_2^4	Q_3^4			
Q_1^5	Q_2^5	Q_3^5			
Q_1^6	Q_2^6	Q_3^6			
Q_1^7	Q_2^7	Q_3^7			
Q_1^8	Q_2^8	Q_3^8			

Figure 3.3: PODColCyclicTransposed data distribution in preprocessing phase

where, Q_i^j is the j^{th} data point in i^{th} snapshot and the processes $\boldsymbol{p}_3$, $\boldsymbol{p}_4$ and $\boldsymbol{p}_5$ are inactive. Hence, if more processes than $\boldsymbol{m}$ are used in computation, then those processes are inactive until the snapshots are read.

The mean of the snapshot matrix is computed in the same way as computed in $PODColCyclic$, 3.1.3. Therefore, analysis of the preprocessing phase is same as the one described in $Table$ 3.1.

The preprocessing phase until here is similar to that of $PODColCyclic$ mentioned in 3.1.3. Next, $transpose(pstran)$ operation is performed on the global snapshot matrix to distribute the data to the inactive processes. This is illustrated in Fig. 3.4:

P_0	P_1	P_2	P_3	P_4	P_5	P_0	P_1
$Q_1{}^1$	$Q_1{}^2$	$Q_1{}^3$	$Q_1{}^4$	$Q_1{}^5$	$Q_1{}^6$	$Q_1{}^7$	$Q_1{}^8$
$Q_2{}^1$	$Q_2{}^2$	$Q_2{}^3$	$Q_2{}^4$	$Q_2{}^5$	$Q_2{}^6$	$Q_2{}^7$	$Q_2{}^8$
$Q_3{}^1$	$Q_3{}^2$	$Q_3{}^3$	$Q_3{}^4$	$Q_3{}^5$	$Q_3{}^6$	$Q_3{}^7$	$Q_3{}^8$

Figure 3.4: PODColCyclicTransposed data distribution during base phase

The size of the original snapshot matrix is $\boldsymbol{Q}_{8x3}$ and that of the transposed snapshot matrix is $\boldsymbol{Q}^T_{3x8}$. Table 3.3 shows the local distribution of the snapshot matrix before and after the transpose operation.

All the processes have a part of the global snapshot matrix after the $transpose$ operation is performed.

Advantages

1. Reading of the snapshots is fairly trivial as each process would exactly know which snapshot to read and how many snapshots to read.

Table 3.3: Analysis of preprocessing phase in PODColCyclic implementation

Process	Local rows in $\boldsymbol{Q}$	Local cols in $\boldsymbol{Q}$	Local rows in $\boldsymbol{Q}^T$	Local cols in $\boldsymbol{Q}^T$
$\boldsymbol{p}_0$	8	1	3	2
$\boldsymbol{p}_1$	8	1	3	2
$\boldsymbol{p}_2$	8	1	3	2
$\boldsymbol{p}_3$	-	-	3	1
$\boldsymbol{p}_4$	-	-	3	1
$\boldsymbol{p}_5$	-	-	3	1

Disadvantages

1. Writing POD modes and coefficients to output is difficult since each column of the global matrix is distributed among processes. Thus, a $transpose$ of the global matrix is required to ensure an entire column is in a single process before writing it to a file.

3.2 Parallelizing Sparse Coding

As discussed in the beginning of this chapter, data distribution of 1D process is simpler to implement over 2D process grid with no communication cost while reading the snapshots. Moreover, most of the sparse coding algorithm requires a complete column to be present in one process. A single coefficient file corresponding to a snapshot is read by one process. Therefore, processes are distributed across the columns of 1D process grid with distribution as:

$$\boldsymbol{p}_{col} = \boldsymbol{p} \text{ and } \boldsymbol{p}_{row} = 1$$

where, $\boldsymbol{p}$ is the total number of processes used in the computation.

Sparse modes are initialized using coefficients generated from running POD using one of the previously mentioned implementations. Since the number of POD coefficient files generated is the same as the number of snapshots, distribution of the coefficients across processes is done in the same manner as in Eq. 3.1.

For example, let the total number of snapshots, $\boldsymbol{m} = 5$ and the total number of processes, $\boldsymbol{p} = 3$. The final distribution of POD coefficients would be:

$$\boldsymbol{S}_0 = 2 \quad \boldsymbol{S}_1 = 2 \quad \boldsymbol{S}_2 = 1$$

where, $\boldsymbol{S}_0$, $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ are POD coefficients in rank 0, 1 and 2 respectively.

P_0	P_1	P_2	P_0	P_1
S_1^1	S_2^1	S_3^1	S_4^1	S_5^1
S_1^2	S_2^2	S_3^2	S_4^2	S_5^2
S_1^3	S_2^3	S_3^3	S_4^3	S_5^3
S_1^4	S_2^4	S_3^4	S_4^4	S_5^4
S_1^5	S_2^5	S_3^5	S_4^5	S_5^5

Figure 3.5: POD coefficients distributed among processes.

where, S_i^j is j^{th} data point in i^{th} coefficient.

As mentioned in Algo. 2, sparse modes $\boldsymbol{X}$ is initialized by summing over two consecutive columns of $\boldsymbol{S}^{POD}$. In addition to reading the POD coefficients according to Fig. 3.5, coefficients must also be read in the way as shown in Fig. 3.6 where the first process starts reading from the 2nd coefficient $\boldsymbol{S}_2$ and proceeds in the same

cyclic manner as before. The respective columns of the POD coefficients read in the two matrices are summed up as shown in 3.7 to initialize sparse modes.

P_0	P_1	P_2	P_0	P_1
S_2^1	S_3^1	S_4^1	S_5^1	S_1^1
S_2^2	S_3^2	S_4^2	S_5^2	S_1^2
S_2^3	S_3^3	S_4^3	S_5^3	S_1^3
S_2^4	S_3^4	S_4^4	S_5^4	S_1^4
S_2^5	S_3^5	S_4^5	S_5^5	S_1^5

Figure 3.6: POD coefficients distributed among processes.

P_0	P_1	P_2
$X_1^1 = S_1^1 + S_2^1$	$X_2^1 = S_2^1 + S_3^1$	$X_3^1 = S_3^1 + S_4^1$
$X_1^2 = S_1^2 + S_2^2$	$X_2^2 = S_2^2 + S_3^2$	$X_3^2 = S_3^2 + S_4^2$
$X_1^3 = S_1^3 + S_2^3$	$X_2^3 = S_2^3 + S_3^3$	$X_3^3 = S_3^3 + S_4^3$
$X_1^4 = S_1^4 + S_2^4$	$X_2^4 = S_2^4 + S_3^4$	$X_3^4 = S_3^4 + S_4^4$
$X_1^5 = S_1^5 + S_2^5$	$X_2^5 = S_2^5 + S_3^5$	$X_3^5 = S_3^5 + S_4^5$

Figure 3.7: Initializing sparse modes using POD coefficients.

Since Batch-OMP and KSVD are two major components in Sparse Coding, parallelization of these two algorithms are described with the pseudo code.

Parallelizing Batch OMP

Batch-OMP is executed in two loops as shown in Algo. 3. The outer loop iterates from 1 to the number of snapshots, $\boldsymbol{m}$. In each iteration, a matrix-vector multiplication is performed to compute y using a single POD coefficient. The vector x is a

column of the POD coefficient and it is already established from the data distribution in Fig. 3.5 that each column is present in a different process. The vector y, used in the inner loop, is independent of the value computed in the previous iteration. Since computation of y in each iteration can be done independently, outer loop can be parallelized. This is achieved by replacing the outer for loop that iterates from 1 to the number of snapshots, $\boldsymbol{m}$, with the loop that iterates from 1 to the number of snapshots per process, $\boldsymbol{m}_p$. Matrix-vector multiplication is replaced by matrix-matrix multiplication of sparse modes $\boldsymbol{X}$ and POD coefficients $\boldsymbol{S}^{POD}$ such that y needed for each iteration is present in its respective process and is independent of the y in other processes. The parallelized version of Batch-OMP is shown in Algo. 4.

Thus, each process will execute the outer for loop in as many iterations as the number of snapshots in that process by distributing work among the processes.

As with the outer for loop, inner for loop cannot be parallelized in its current form. To achieve parallelization, processes must run the loop independent of other processes. However, the "Update y" step in the inner for loop requires data points from $\boldsymbol{G}$ that are present in different processes. Moreover, the data points required in each iteration are different and it is associated with several calls to MPI subroutines to communicate the data. Hence, it would slow down the process and an alternate solution employed is to maintain copies of the global matrix $\boldsymbol{G}$ in each process to facilitate independent execution. This solution is used, however it is not scalable and feasible to maintain large matrices of $\boldsymbol{G}$ in every process.

Algorithm 3: Batch OMP algorithm

Input : Sparse modes $\boldsymbol{X}$, POD coefficients $\boldsymbol{S}^{POD}$, Number of active modes L
Output: Sparse coefficient matrix ($\boldsymbol{S}^{SPC}$)

Init: $\boldsymbol{G} = \boldsymbol{X}^T\boldsymbol{X}$, $index$ = $zeros(L)$

for i = 1...m ***do***
  $x = \boldsymbol{S}_i^{POD}$
  $y^{'} = \boldsymbol{X}^T x$
  $y = y^{'}$
  for j = 1...L ***do***
    Get the position of max value in y
    $pos = \boldsymbol{max}(\boldsymbol{abs}(y))$
    Store the value of pos in $index$
    $index(j) = pos$
    Update y
    $y = \boldsymbol{G}(:index(1:j) * \boldsymbol{pinv}(\boldsymbol{G}(index(1:j) : index(1:j))$
    $y = y * y^{'}(index(1:j)) - y^{'}$
    Initialize values in y to 0 such that same active modes are not chosen
    $y(index(1:j)) = 0$
  end
  $a = pinv(X(:, index(1:L))) * x$
  $\boldsymbol{S}^{SPC}(index(1:j), i) = a$
end

Algorithm 4: Parallelized Batch OMP algorithm

Input : Sparse modes $\boldsymbol{X}$, POD coefficients $\boldsymbol{S}^{POD}$, Number of active modes L
Output: Sparse coefficient matrix ($\boldsymbol{S}^{SPC}$)

Init: $\boldsymbol{G} = \boldsymbol{X}^T\boldsymbol{X}$, $index$ = $zeros(L)$

$\boldsymbol{Y} = \boldsymbol{X}^T\boldsymbol{S}^{POD}$
for i = 1...m_p ***do***
  $y = \boldsymbol{Y}_i$
  for j = 1...L ***do***
    $Continue$ ...
  end
  $a = pinv(X(:, index(1:L))) * x$
  $\boldsymbol{S}^{SPC}(index(1:j), i) = a$
end

Parallelizing KSVD

Unlike Batch-OMP in Algo. 3, KSVD is entirely executed in parallel. In contrast with breaking down the for loop in Batch-OMP, all the processes involved in the computation enter the loop at the same time and each process iterates until the user provided input number of modes.

Algorithm 5: Parallelized KSVD algorithm

Input : Sparse modes $\boldsymbol{X}$, POD coefficients $\boldsymbol{S}^{POD}$
Output: Sparse coefficient matrix ($\boldsymbol{S}^{SPC}$), Sparse modes $\boldsymbol{X}$

1 ***for*** i = 1...k ***do***
2 Find data indices whose values are $non-zero$ along row i in $\boldsymbol{S}^{SPC}$
$non-zero-indices = \boldsymbol{find}(\boldsymbol{S}^{SPC}(i,:))$
3 If there are no $non-zero$ values, replace the non active mode with column of $\boldsymbol{S}^{POD}$
4 $\boldsymbol{S}^{'} = \boldsymbol{S}^{SPC}(:, non-zero-indices)$
5 $\boldsymbol{E} = \boldsymbol{S}^{POD}(:, non-zero-indices) - \boldsymbol{X}\boldsymbol{S}^{SPC}$
6 $updated-modes, singular-value, \beta = function : eigen-power(\boldsymbol{E})$
7 $\boldsymbol{S}^{SPC}(i, non-zero-indices) = singular-value * \beta$
8 $\boldsymbol{X}_i = updated-modes$
end

Since, $non-zero-indices$ is obtained from $\boldsymbol{S}^{SPC}$ for i^{th} row in each iteration, this statement is executed by the process that has the data corresponding to i^{th} row.

Next, a temporary coefficient matrix, $\boldsymbol{S}^{'}$, is constructed using only the columns corresponding to the $non-zero-indices$. To ensure all the processes are involved in computation in this step, a transposed form of the coefficient matrix, $\boldsymbol{S}^{SPC}$, is used. The error, $\boldsymbol{E}$, is computed as shown in Algo. 5 and is used as an input to the $eigen-power$ function, which returns the $updated-modes$ upon completion.

The resulting $updated - modes$ replaces the i^{th} column of the sparse modes matrix. Therefore, only the process that has the i^{th} column of the sparse modes matrix performs this operation.

Thus, the for loop is executed as many times as the number of user-defined modes, k. Hence, complexity of the KSVD algorithm is directly dependent on k.

Chapter 4

Experimental Results and Analysis

This chapter illustrates the results obtained for Proper Orthogonal Decomposition and Sparse Coding on various data sets. "Total memory consumed" and "Total time taken" are the two major metrics of performance analyzed in the results.

4.1 Proper Orthogonal Decomposition

Performance of all three implementations discussed previously i.e. $PODColCyclic$, $PODRowCyclic$ and $PODColCyclicTransposed$ is assessed by varying parameters such as:

- Number of snapshots
- Number of data points in each snapshot
- Number of POD modes
- Number of computational nodes

4.1.1 Complexity analysis and Memory requirement for POD

Before the results are illustrated, complexity analysis and memory allocated explicitly by the program are tabulated:

Table 4.1: Complexity analysis and Memory allocated for POD

Operation	Order of Complexity ($\boldsymbol{O}$)	Memory for all processes
Snapshot matrix, $\boldsymbol{Q}$	-	$n \times m$
Truncated grid, $\bar{\boldsymbol{v}}$	-	$p \times n$
Mean, $\bar{\boldsymbol{U}}$	$(mn + n)$	$p \times n$
$\boldsymbol{q}(\boldsymbol{x}, t) = \boldsymbol{Q} - \bar{\boldsymbol{U}}$	(mn)	-
$\boldsymbol{Q}_F = \boldsymbol{q}(\boldsymbol{x}, t) \times \bar{\boldsymbol{v}}$	(mn)	-
$\boldsymbol{A} = \boldsymbol{Q}_F^T \boldsymbol{Q}_F$	$(nm^2 + nm)$	m^2
$\boldsymbol{AV} = \boldsymbol{\Lambda V}$	(m^3)	$2m^2 + p \times m$
$\boldsymbol{\Phi} = \boldsymbol{QV}$	(nmk)	$n \times k$
$\boldsymbol{S} = \boldsymbol{\Phi}^T \boldsymbol{Q}$	$(knm + nk)$	$k \times m$
$\boldsymbol{E} = \boldsymbol{\Phi S}$	(nkm)	$n \times m$

where, m is the number of snapshots, n is the number of points in a single snapshot, p is the total number of processes and k is the user-defined number of POD modes.

The total computational complexity of POD algorithm is given by the sum of all the terms in the 2nd column:

$$Complexity = \boldsymbol{O}(m^3 + nm^2 + 3knm + 3nm + nk + n) \tag{4.1}$$

Equation 4.1 suggests that complexity of POD is more sensitive to the number

of snapshots. Thus, a quadratic increase in time can be expected with increase in m.

Similarly, adding up all the values in the 3rd column gives us the total memory allocated for the data used in POD, which is given by:

$$Memory = (3m^2 + 2(n \times m) + 2(p \times n) + k(n + m) + p \times m) \tag{4.2}$$

Equation 4.2 suggests that memory allocated for the data used in the algorithm varies in the order of m^2. Moreover, the number of processes, p, present in the equation indicates that memory is expected to increase with the increase in number of processes. The number of modes, k, can go up to m if 100 percent modes are chosen for computation in which case memory required will increase further. Although all these parameters including the size of each snapshot n influence the memory requirement in a way, memory is more sensitive to the total number of snapshots, m, which is similar to the observation in the complexity equation as well.

4.1.2 Strong Scaling

In strong scaling experiment, computational resources are varied by keeping the data constant. The questions posed in such an experiment are:

- How does the performance vary by keeping the amount of work constant, but by increasing the computational resources?

- Is it possible to see a reduction in the total time taken for the algorithm as the number of processes is increased? If so, is it possible to always expect faster execution of the algorithm by just increasing the computing resources?

This experiment is conducted as per the data listed in Table 4.2:

Table 4.2: Data used for strong scaling experiment

Number of data points	Number of snapshots	Number of modes
100M	100	40
1M	5000	2000
1K	10000	4000

Three data sets are used to conduct the strong scaling experiment. The data set that has a snapshot matrix of dimensions $100\ million\ points \times 100\ snapshots$ is considered as a tall snapshot matrix since $n >> m$. Similarly, the data set with dimensions $1\ million\ points \times 5000\ snapshots$ is considered as a moderately tall snapshot matrix with $n > m$ and the one with the dimensions $1\ thousand\ points \times 10000\ snapshots$ is considered as a fat snapshot matrix with $m >> n$. These data sets were chosen for experimental purpose as they are comparable in dimensions to the data sets used in practice.

For each of the above data set, the number of snapshots and modes is kept constant. However, results are obtained by doubling the number of nodes from 2 to 16. Each node uses 28 processes for computation. Thus the total number of processes, $\boldsymbol{p}$, is the number of nodes $\times$ 28.

4.1.2.1 Issues with I/O Time

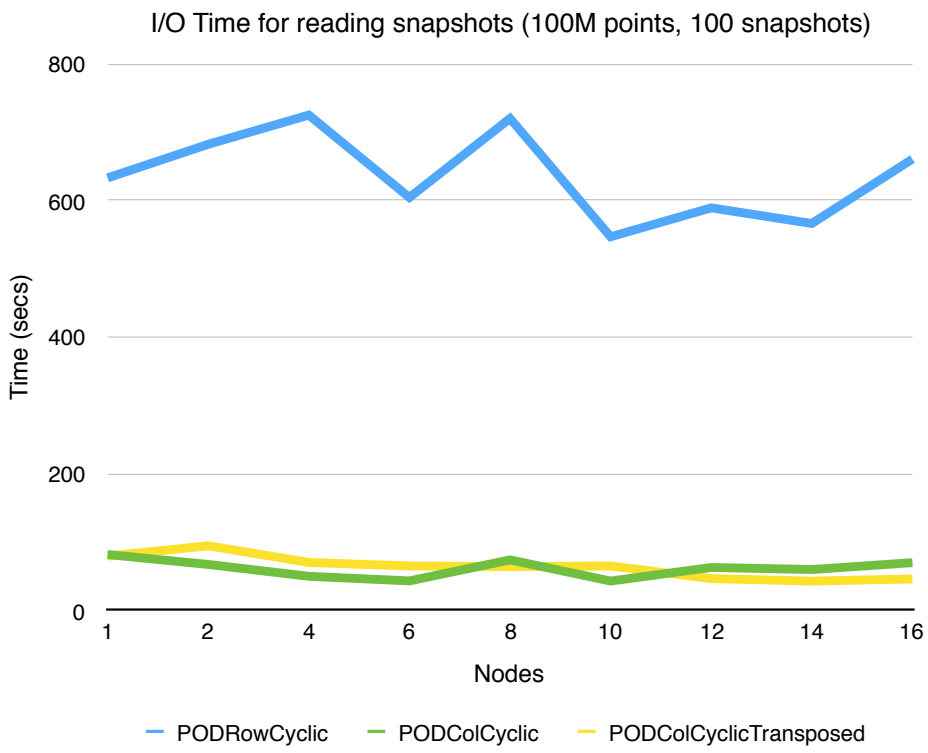

Figure 4.1: Time taken for reading snapshots (100M points, 100 snapshots)

It is important to consider the implication of I/O time to read snapshots for various implementations before delving into the analysis of the results. It is already discussed in the implementation that $PODRowCyclic$ will end up spending a significant amount of time in reading snapshots due to poor caching. The difference in the amount of time taken by $PODRowCyclic$ to read the snapshots when compared to $PODColCyclic$ and $PODColCyclicTransposed$ is seen in Fig. 4.1. As the number of processes increases, time to read the snapshots must decrease since fewer data points are read by each process. However, a constant decrease in time is not seen for any of the implementations. Thus, it can be expected to see

slight inconsistencies in the results that involve I/O time for reading the snapshots. Therefore, a tolerance of 5 to 10 percent difference for total time taken between consecutive execution of the same data set is acceptable.

4.1.2.2 Comparison of Computation, MPI and I/O Time

Since, algebraic operations such as Matrix Multiplication, Matrix Transpose and Singular Value Decomposition(SVD) are performed in parallel using a large number of processes, the time required for communication of data among processes is a major factor in the consideration of total time taken. Since, Message Passsing Interface(MPI) is used implicitly by ScaLapack and explicitly by the application for communicating data among processes, rest of the results that refer to $MPI\ Time$ is the overall time spent for communication in the execution of the program. By tuning the performance parameters such as the number of snapshots, the number of modes and the number of nodes it is seen that $MPI\ Time$ varies differently for different implementations. The graphs, 4.2, 4.3, 4.4 illustrate the comparison of $I/O\ Time$, $Computation\ Time$ and $MPI\ Time$ by varying computational resources.

MPI time increases with increase in the number of nodes. Although the data used for computation is constant, it is distributed among further number of processes with an increase in number of nodes. Thus, collective communication time increases to perform the same number of operations. Since computational complexity of the algorithms in a parallel environment is represented as a function of the number of processes, complexity not only depends on the size of data but also

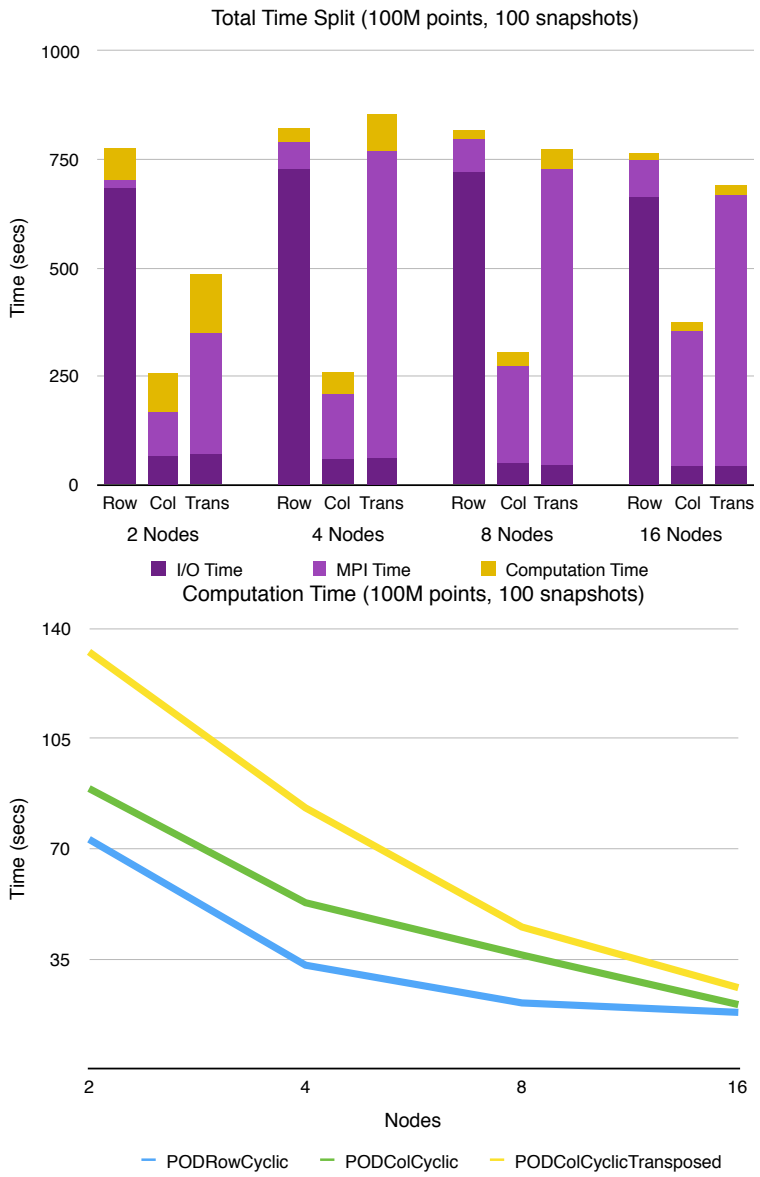

Figure 4.2: (a) Computation, MPI and I/O Time Comparison (b) Computation Time for (100M points, 100 snapshots)

on the number of processes used in computation.

Computation time decreases with an increase in number of nodes. This is due to the fact that with a fewer number of nodes, computational load falls on fewer processes and, as a result work done per process is high. However, as the number

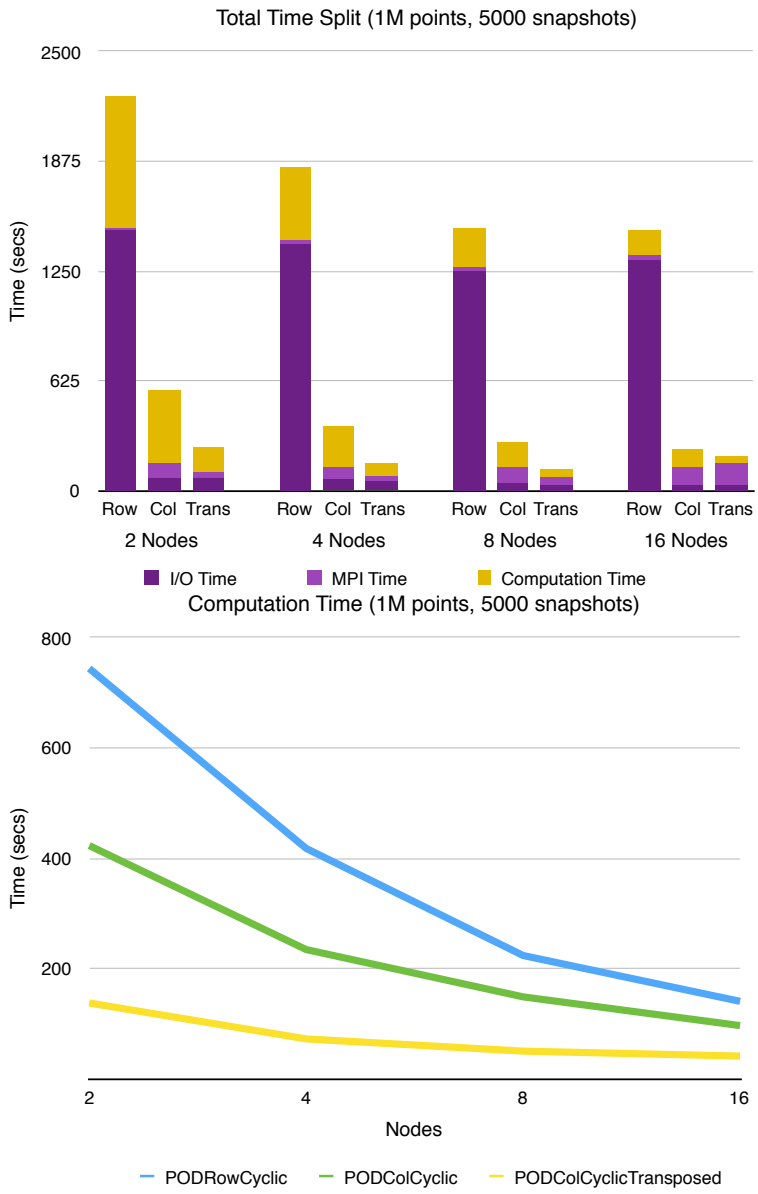

Figure 4.3: (a) Computation, MPI and I/O Time Comparison (b) Computation Time for (1M points, 5000 snapshots)

of nodes increases, work is distributed among more processes, reducing the load on each process significantly and enables processes to perform better. The results indicate the same and uphold the fact that better parallelism is achieved by increasing the computational resources, however, at the cost of increasing total time

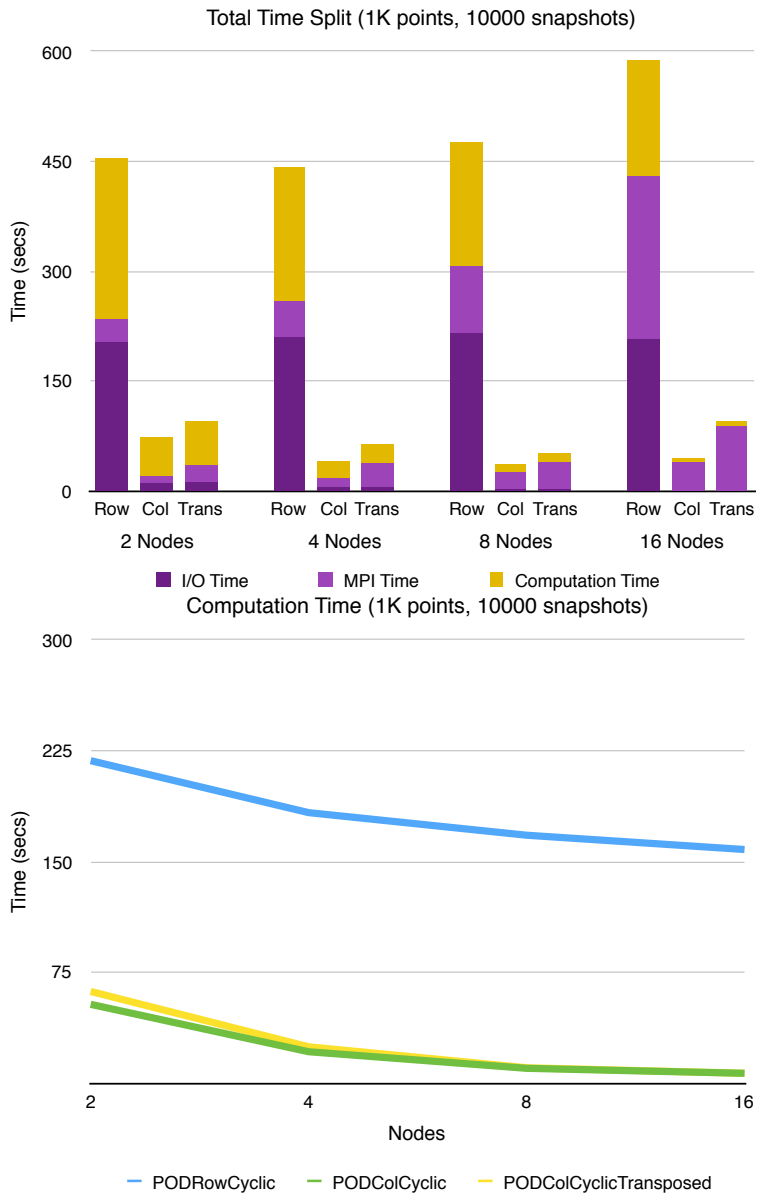

Figure 4.4: (a) Computation, MPI and I/O Time Comparison (b) Computation Time for (1K points, 10000 snapshots)

due to increase in time required for communication.

Depending on the type of the data set and implementation used, there is always an optimum number of nodes on which the performance is better. It is important to note that increasing the computational resources further would only allow MPI

time to dominate the Total time required for completion of the algorithm.

4.1.2.3 Data Transfer

This section illustrates the amount of data transfer that takes place with different implementations for different snapshot matrices with the increase in computational resources.

Prior to demonstrating the trend with respect to Data Transfer, it is important to visualize how matrix multiplication works in parallel with $PODColCyclic$ and $PODRowCyclic$ implementations.

Matrix Multiplication in PODColCyclic Implementation

For the purpose of demonstration, matrix multiplication performed during the computation of POD modes is used. In Fig. 4.5, Q_i^j is j^{th} data point in i^{th} snapshot and V_i^j is j^{th} data point in i^{th} singular vector. Φ_i^j is the j^{th} data point in i^{th} POD mode. Φ_1^1 and Φ_2^2 must be computed as:

$$\Phi_1^1 = Q_1^1 V_1^1 + Q_2^1 V_1^2 + Q_3^1 V_1^3 + Q_4^1 V_1^4 + Q_5^1 V_1^5$$

$$\Phi_2^2 = Q_1^2 V_2^1 + Q_2^2 V_2^2 + Q_3^2 V_2^3 + Q_4^2 V_2^4 + Q_5^2 V_2^5$$

To compute Φ_1^1 at P_0, entire column of V_1 is required and similarly to compute Φ_1^1, entire column of V_2 is required. With respect to $PODColCyclic$ implementation, entire column of the global matrix is present in a single process. Thus, there is no data transfer associated with the global matrix, V. However, entire row of Q is required from Q_1^1 to Q_5^1. This is distributed among the processes along the

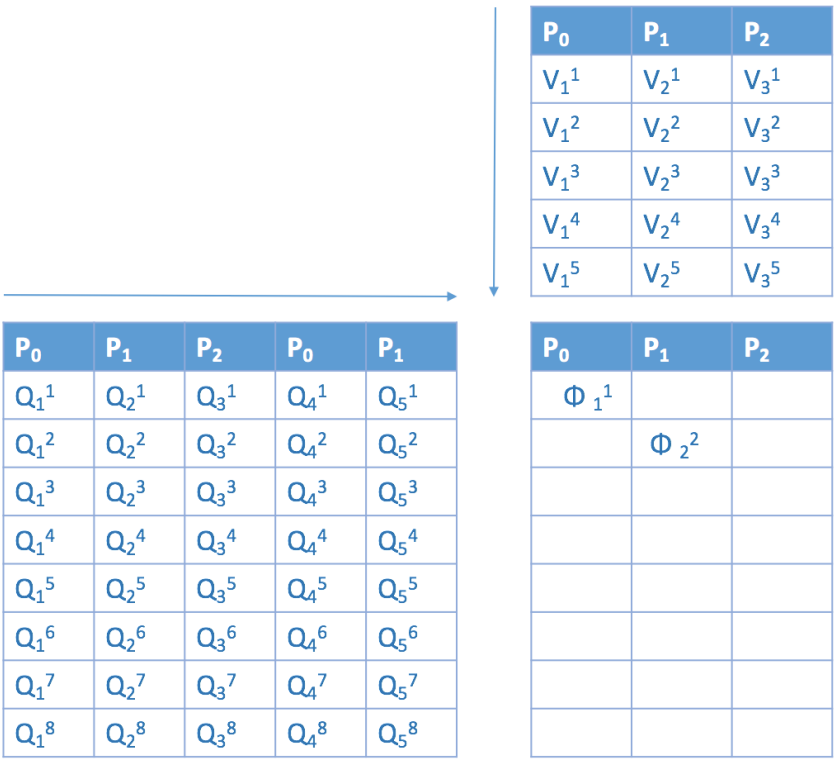

Figure 4.5: PODColCyclic Matrix Multiplication

row as seen from Fig. 4.5. Thus the first row of Q has to be communicated to P_0. Since, Q_1^1 and Q_4^1 are already present in P_0, only the rest of the values in the row have to be sent to P_0. In a similar way, to compute Φ_2^2 at P_1, entire column of V_2 is required which is already present in P_1. However, it requires values from 2^{nd} row of Q which are not present in P_1. This extends to all other processes as well. As a result, by the end of matrix multiplication, entire Q matrix would be distributed to P_0, P_1 and so on. Although it is not always feasible for each process to hold the entire Q matrix, a temporary buffer is created in every process and necessary data is transferred to it as and when required.

Matrix Multiplication in PODRowCyclic Implementation

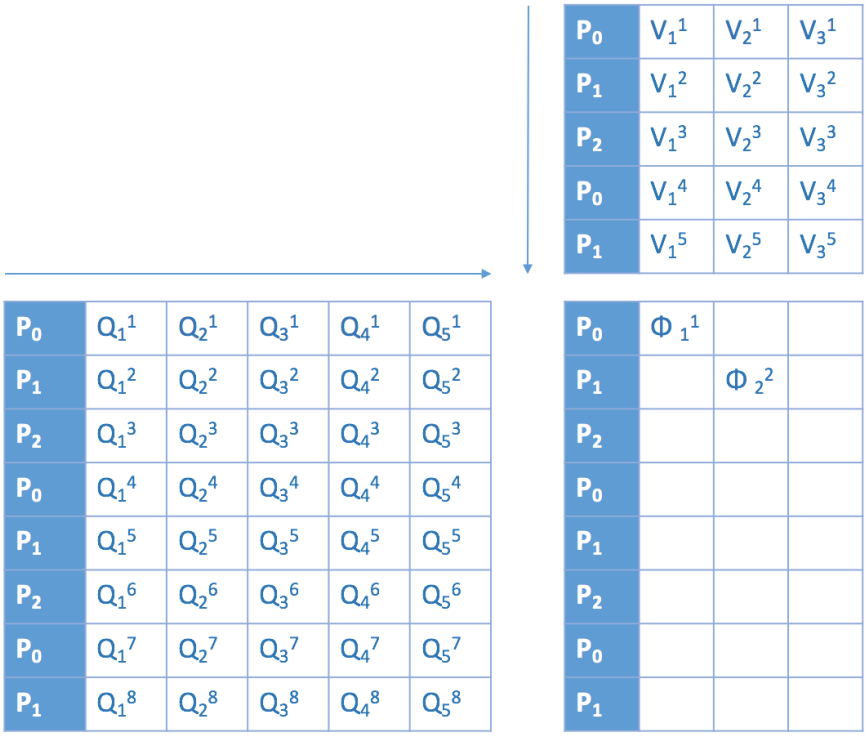

Figure 4.6: PODRowCyclic Matrix Multiplication

Matrix multiplication with $PODRowCyclic$ is computed similarly to that of $PODColCyclic$ implementation. However, it differs in the way data is transferred. To compute Φ_1^1, first row of Q and first column of V is needed. Previously with $PODColCyclic$, first column of V was present in P_0, but with $PODRowCyclic$ first row of Q is present in P_0. Thus to compute Φ_1^1, first column of V is needed that is not already present in P_0. This extends to all other processes as well and as a result, by the end of matrix multiplication, entire V matrix would be transferred to P_0, P_1 and so on as and when it is required.

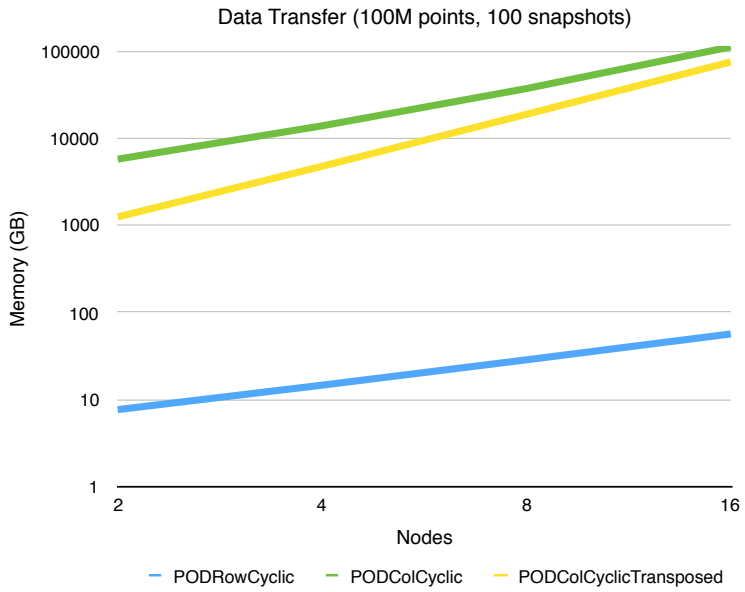

Figure 4.7: (a) Computation, MPI and I/O Time Comparison (b) Computation Time for (1M points, 5000 snapshots)

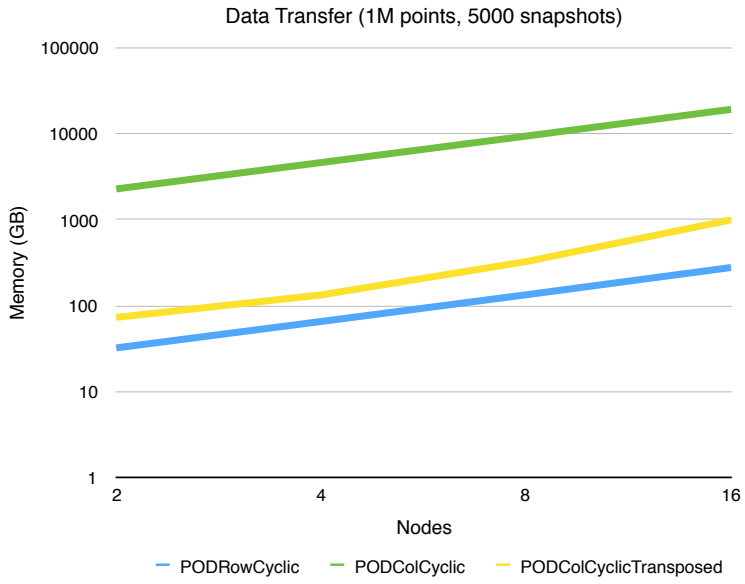

Figure 4.8: (a) Computation, MPI and I/O Time Comparison (b) Computation Time for (1M points, 5000 snapshots)

In the sections 4.1.2.3 and 4.1.2.3, it was established that entire matrix present on the left-hand side of the matrix multiplication equation is transferred in case of

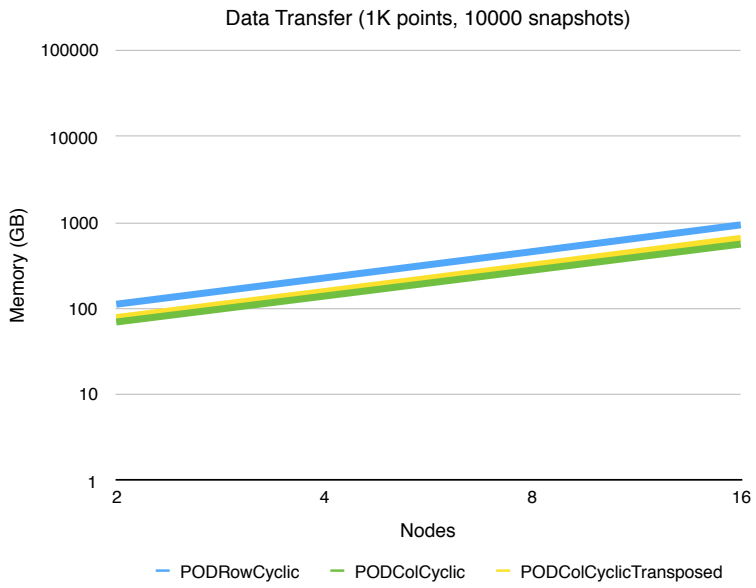

Figure 4.9: (a) Computation, MPI and I/O Time Comparison (b) Computation Time for (1M points, 5000 snapshots)

$PODColCyclic$ implementation and entire matrix present on the right-hand side of the matrix multiplication equation is transferred in the case of $PODRowCyclic$ implementation. Thus, high data transfers are observed for $PODColCyclic$ implementation with tall snapshot matrices as seen in Fig. 4.7 and $PODRowCyclic$ implementation with fat snapshot matrices as seen in Fig. 4.9. In addition to this, if $PODColCyclic$ uses only as many processes as the number of snapshots to read snapshots, rest of the processes do not have any data present in them. Thus $PODColCyclic$ is associated with high data transfer to ensure rest of the processes are also involved in the computation.

Although data transfers are in the order of thousands of Giga Bytes(GB), total memory consumed is still a small fraction of that number. This is because processes allocate temporary buffers needed for intermediate communication. More-

over, MPI Time is proportional to the data transfer. Therefore, high data transfers indicate that MPI time takes a significant proportion of the total time required for computation.

Another important observation is that amount of data transfer increases with increase in computational resources. This is because increase in computational resources require more processes to communicate with each other to perform linear algebraic computations that are communication intensive.

4.1.2.4 Total Memory Consumed

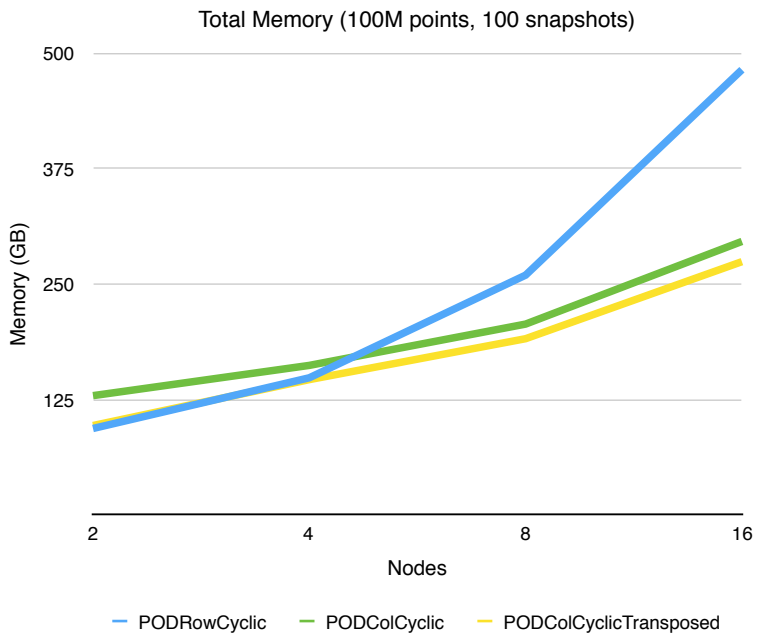

Figure 4.10: Total Memory (100M points and 100 snapshots)

It has already been established from the previous section of Data Transfer, 4.1.2.3, that increase in computational resources increases the number of processes used in computation and thus temporary memory allocated for computation by these pro-

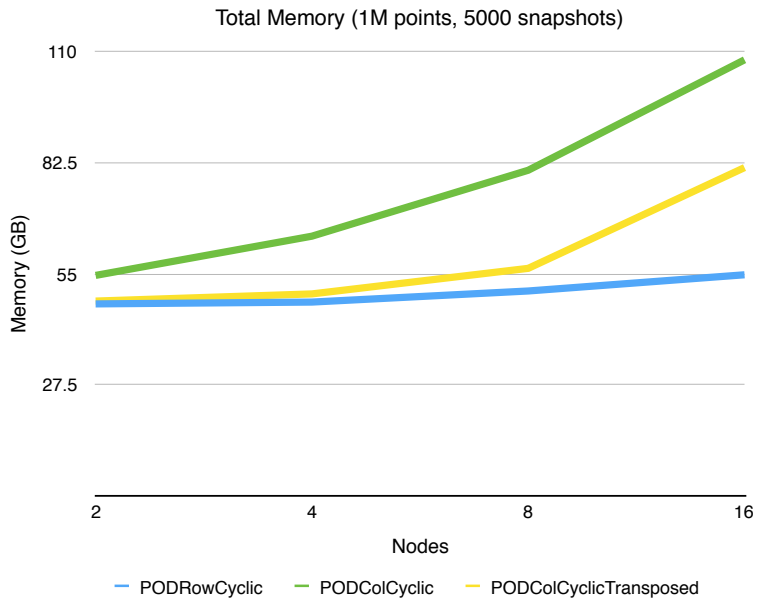

Figure 4.11: Total Memory (1M points and 5000 snapshots)

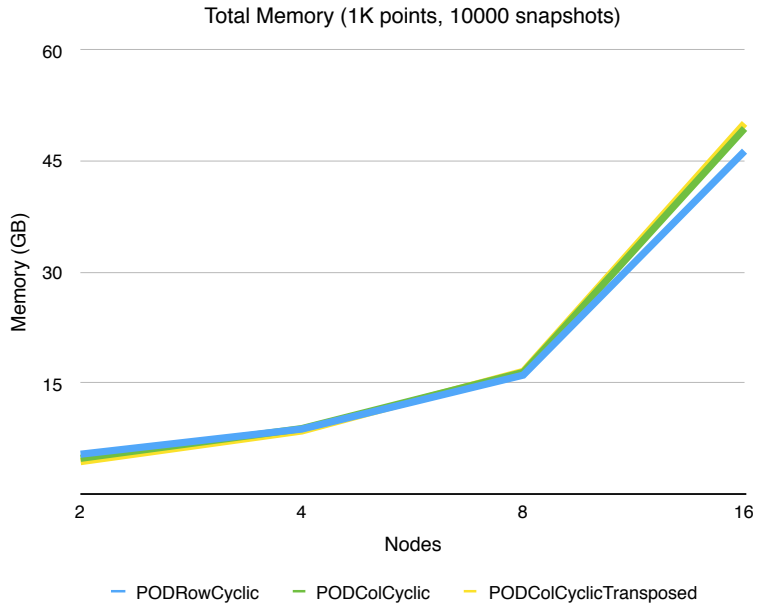

Figure 4.12: Total Memory (1K points and 10000 snapshots)

cesses would add up to the total memory. This is the reason for the linear increase in memory as observed in the graphs, 4.10, 4.11 and 4.12.

The disparity in memory consumed with different implementations is due to

the difference in temporary buffers allocated by ScaLapack for matrix multiplication operations.

4.1.2.5 Inference on Strong Scaling

Increase in computational resources increases MPI Time and thus it is beneficial to perform POD on a fewer number of nodes to ensure that all the data is accommodated and around 40 to 50 percent of the memory is left for the allocation of temporary buffers by ScaLapack.

Better parallelism is achieved by using more computational resources but it quickly gets overwhelmed by the communication cost.

$PODColCyclic$ appears to be the best among implementations with better execution time for various snapshot sizes and marginally more memory is consumed. However, it is still well within the user available limit on a node and the difference in memory usage is not high.

4.1.3 Weak Scaling

In weak scaling experiment, computational resources and data are varied proportionally. The questions posed in such an experiment are:

- Do we see a constant ratio of work done per process as we double the data and the computational resources? Do the implementations scale well?
- How does the memory vary by proportionally varying the parameters such

as the number of snapshots or the number of modes. Does memory increase at all? If yes, do we know which parameter is memory more sensitive to?

With weak scaling, two experiments are performed, namely, changing the number of snapshots and the number of modes in proportion to the number of nodes.

4.1.4 Varying Snapshots

This experiment is conducted as per the data listed in Table 4.3:

Table 4.3: Data used for weak scaling - Varying number of snapshots

Number of data points	Number of modes	(Nodes, Snapshots)	(Nodes, Snapshots)	(Nodes, Snapshots)	(Nodes, Snapshots)
100M	40	(2, 50)	(4, 100)	(8, 200)	(16, 400)
1M	2000	(2, 5000)	(4, 10000)	(8, 20000)	(16, 40000)
1K	4000	(2, 10000)	(4, 20000)	(8, 40000)	(16, 80000)

4.1.4.1 Comparison of Computation, MPI and I/O Time

It is to be noted that as the number of snapshots increases, computational complexity increases as shown in Eq. 4.1.

In Fig. 4.13, increase in MPI time is not proportional to increase in snapshots for $PODColCyclicTransposed$. It can be seen that MPI Time is very high at more than 2 hours for as low as 400 snapshots. Since it is required to decompose in

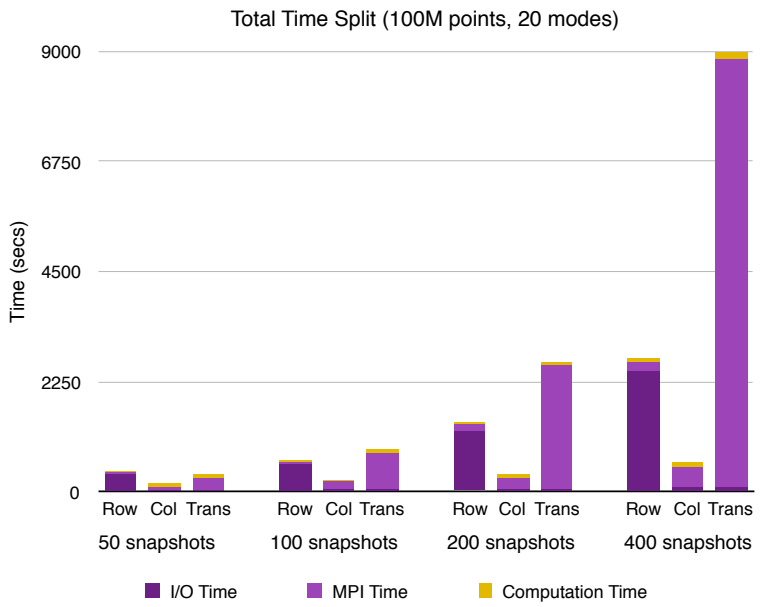

Figure 4.13: Computation, MPI, I/O Time Comparison (100M points, 20 modes)

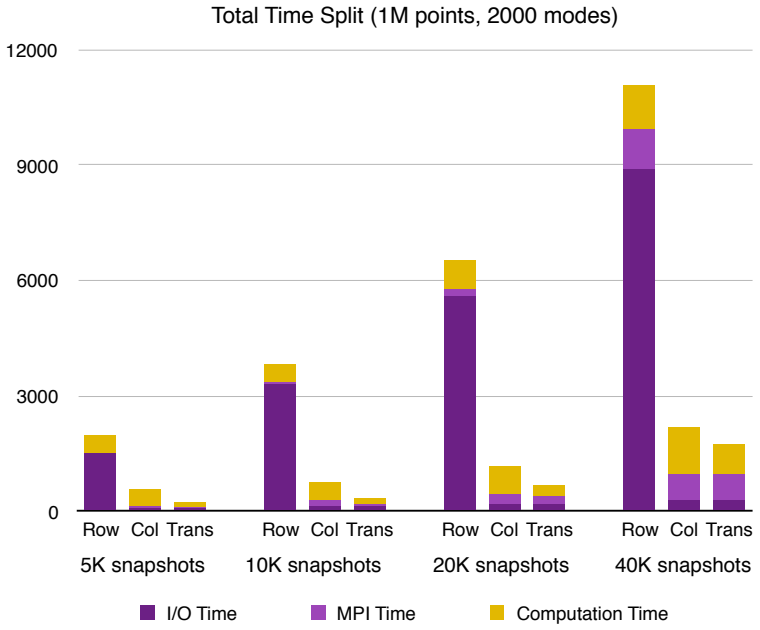

Figure 4.14: Computation, MPI, I/O Time Comparison (1M points, 2000 modes)

the order of thousands of snapshots, $PODColCyclicTransposed$ is not suitable for computation of tall snapshot matrices and does not scale well.

There is a huge increase in I/O time for reading snapshots with $PODRowCyclic$

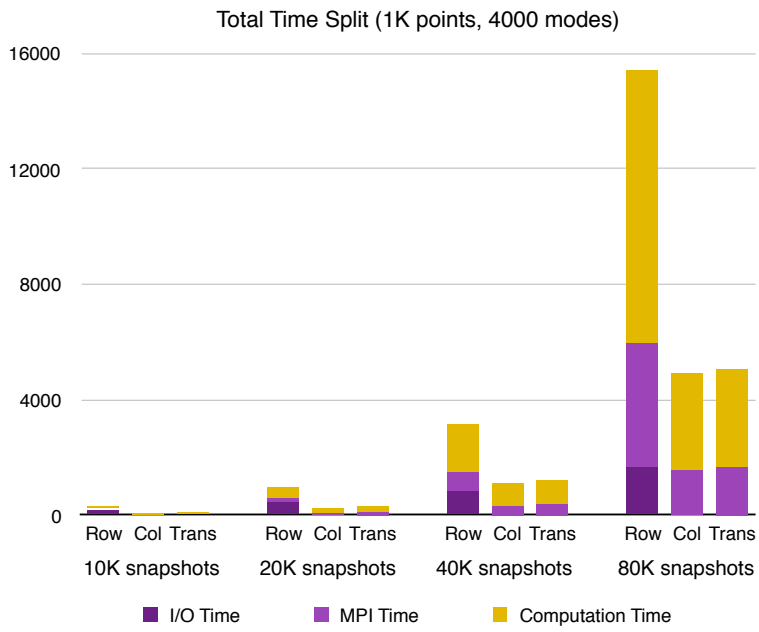

Figure 4.15: Computation, MPI, I/O Time Comparison (1K points, 4000 modes)

as seen in Fig. 4.14 and Fig. 4.15. This results in $PODRowCyclic$ taking 4 to 5 times more time than the other two implementations. Thus, $PODRowCyclic$ is highly I/O intensive and does not scale well with the increase in number of snapshots because each process reads a part of every snapshot which would result in a heavy load on I/O coupled with poor caching ability.

A substantial increase in computation time as seen in Fig. 4.15 for 80K snapshots is due to the time taken for SVD operation as shown in Fig. 4.16.

$PODColCyclicTransposed$ has an advantage over $PODColCyclic$ implementation for the data set (1M points, 2000 modes), as seen in Fig. 4.14, because it takes less time for computing reconstructed snapshot matrix. This operation is required only when reconstruction error for POD is to be computed to verify the accuracy with which decomposition is performed. However, the difference is not too signif-

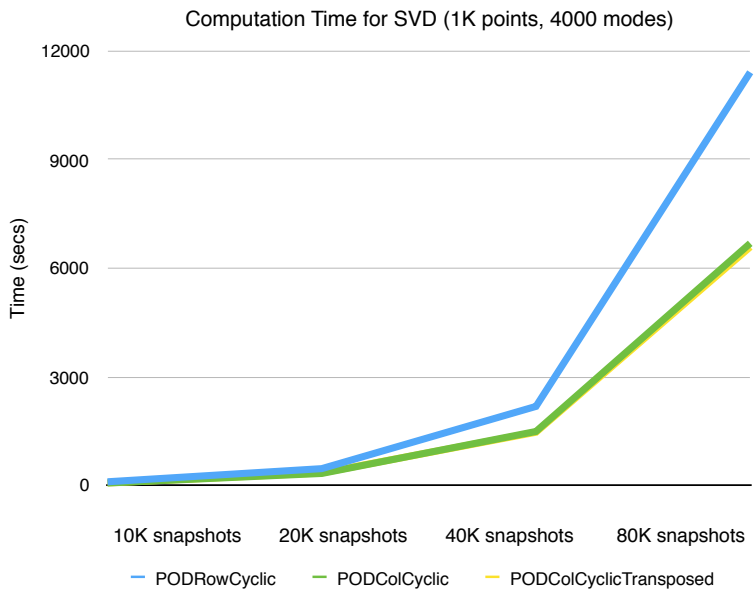

Figure 4.16: SVD Time (1K points, 4000 modes)

icant to infer that $PODColCyclicTransposed$ performs better than $PODColCyclic$.

4.1.4.2 Total Memory Consumed

Total memory consumed is similar irrespective of the type of data set or implementation used. Thus, each implementation scales similarly with respect to memory and there is no particular advantage of using one implementation over another.

Memory consumed is more sensitive to the number of snapshots, m, because memory allocated is proportional to m^2 as stated in Eq. 4.2. Thus, increase in memory for fat snapshot matrices is observed as seen in Fig. 4.19.

Memory allocated using Eq. 4.2 is plotted as the dotted line. It is observed that memory allocated by the program is comparable to memory consumed for tall snapshot matrices, however, margin of error in prediction increases with the

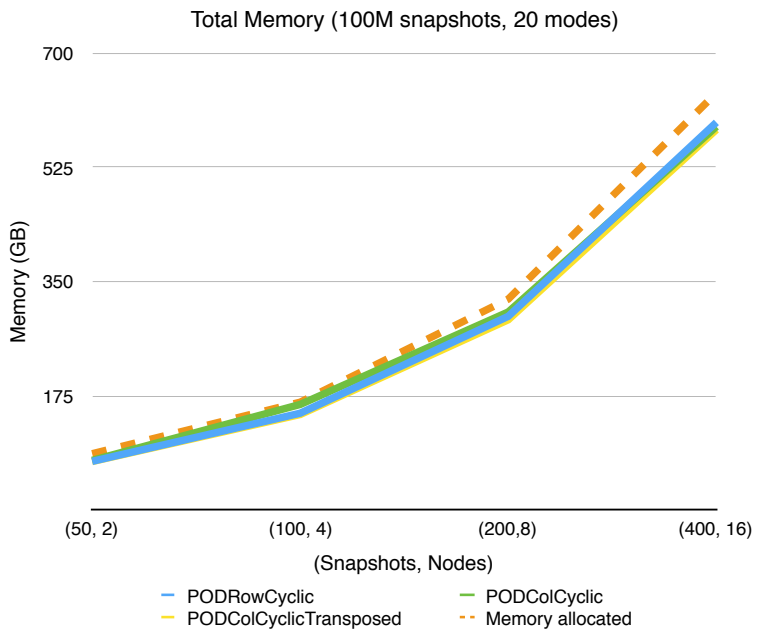

Figure 4.17: Total Memory Consumed (100M points, 20 modes)

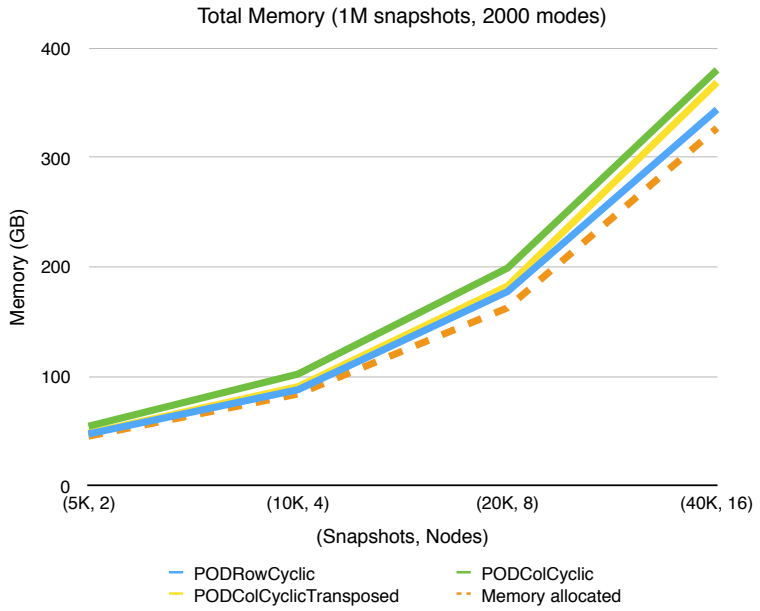

Figure 4.18: Total Memory Consumed (1M points, 2000 modes)

increase in number of snapshots. Thus, Eq. 4.2 provides a way to predict the memory required to choose the number of computational resources needed to execute POD in parallel.

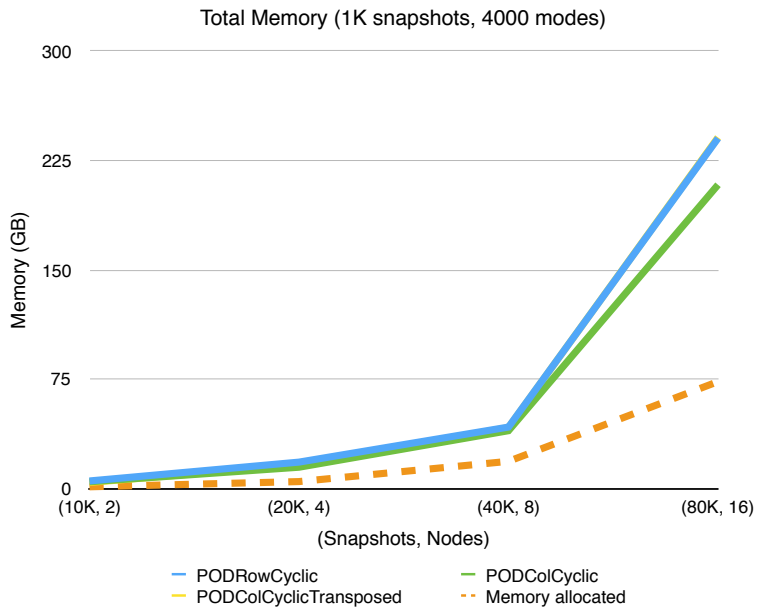

Figure 4.19: Total Memory Consumed (1K points, 4000 modes)

4.1.5 Varying Modes

This experiment is conducted as per the data listed in Table 4.4:

Table 4.4: Data used for weak scaling - Varying number of modes

Number of data points	Number of snapshots	(Nodes, POD modes)	(Nodes, POD modes)	(Nodes, POD modes)	(Nodes, POD modes)
100M	100	(2, 10)	(4, 20)	(8, 40)	(16, 80)
1M	5000	(2, 500)	(4, 1000)	(8, 2000)	(16, 4000)
1K	10000	(2, 1000)	(4, 2000)	(8, 4000)	(16, 8000)

4.1.5.1 Comparison of Computation, MPI and I/O Time

As the number of modes increases, computational complexity increases as shown in Eq. 4.1. However, the effect of increase in the number of modes, k, is quite less

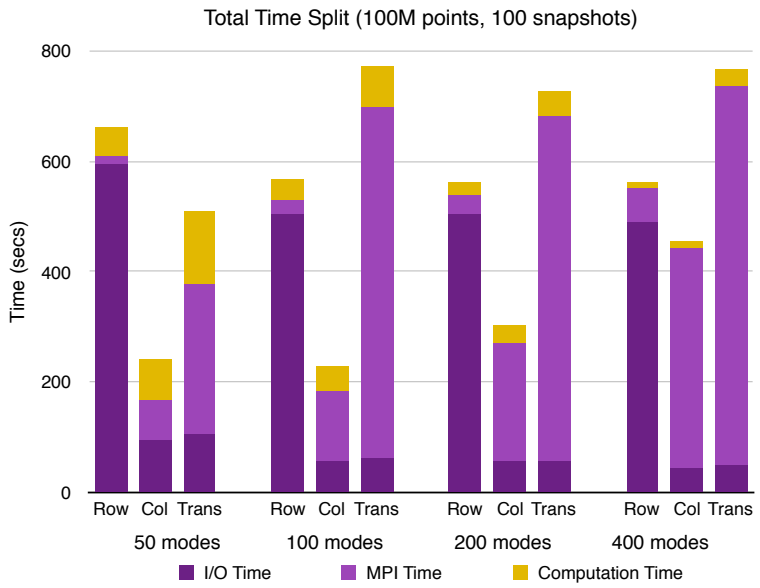

Figure 4.20: Computation, MPI, I/O Time Comparison (100M points, 100 snapshots)

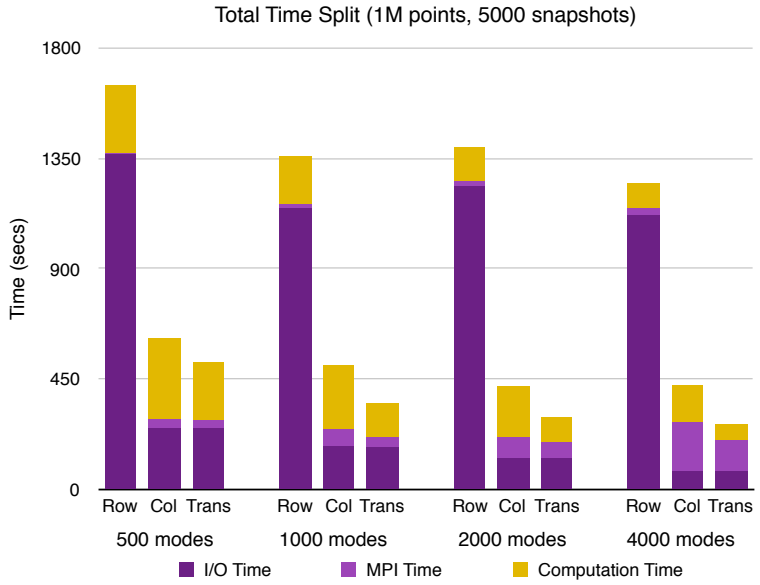

Figure 4.21: Computation, MPI, I/O Time Comparison (1M points, 5000 snapshots)

as compared to the increase in snapshot size, n, or the number of snapshots, m. Thus, increase in work does not substantiate a proportional increase in nodes. As a result, computation time reduces quickly, however, MPI time reaches a minimum,

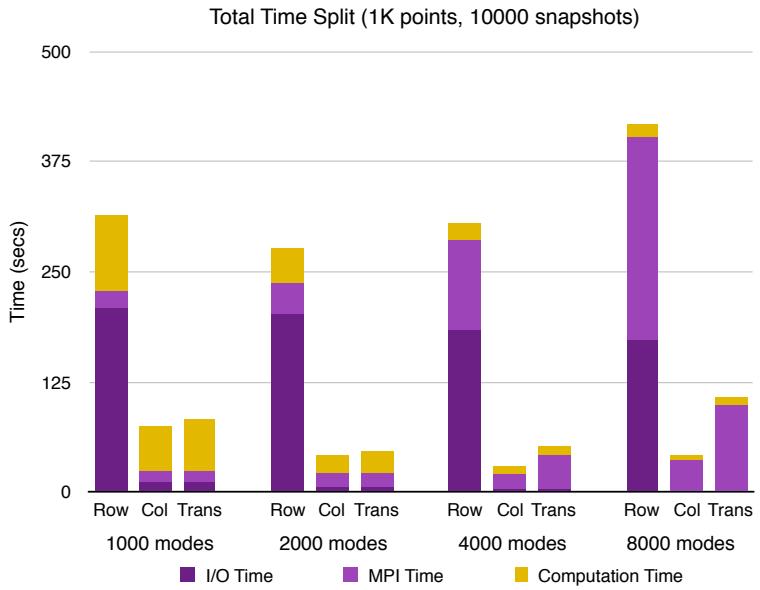

Figure 4.22: Computation, MPI, I/O Time Comparison (1K points, 10000 snapshots)

post which it starts increasing to an extent of dominating the overall time as shown in Fig. 4.20, 4.21 and 4.22.

4.1.5.2 Total Memory Consumed

Total memory consumed follows a similar pattern as compared to varying the number of snapshots in Sec. 4.1.4.2. Since memory, is more sensitive to the number of snapshots, m, than the number of modes, k, lesser increase in memory was observed. However, as with the previous case, there is no benefit of one implementation over another when it comes to consumption of memory.

4.1.6 Inference on Weak Scaling

Results for $PODColCyclic$ suggest that the implementation is scalable with respect to doubling the number of snapshots. Total time taken for cases using the largest number of snapshots with each data set varies from under 30 minutes for 100M points and 400 snapshots to just over an hour for 1K points and 80000 snapshots. There is no spike observed with respect to the total time taken or memory consumed for $PODColCyclic$ implementation. Thus, it is possible to conclude from these results that larger data sets can be decomposed within a reasonable amount of time and also consume memory well within the user available limits of each node.

A spike in MPI time for $PODColCyclicTransposed$ implementation for 100M points and 400 snapshots case suggests that $transpose$ of a global matrix is a communication heavy operation and consumes most of the time. Similarly, I/O time for reading snapshots in the case for $PODRowCyclic$ increases substantially with the increase in the number of snapshots. Thus, these implementations would not necessarily scale well with respect to tall and fat snapshot matrices respectively.

4.1.7 Experiments with Fixed Computational Resources

Three experiments are performed by varying shape of the snapshot matrix, number of snapshots and number of pod modes by keeping computational resources fixed at 8 nodes. The questions posed in such an experiment are:

- How does the performance change by varying just one of the parameters and keeping the others fixed? This is important to eliminate the non-linearity with respect to change of multiple parameters simultaneously.

- By keeping the total memory of the snapshot matrix constant and by varying the shape of the matrix, how does the performance change?

- Is the performance dependent on the total number of points used in the snapshot matrix or on the dimensions of the matrix, namely, size of each snapshot and the number of snapshots?

4.1.8 Change of shape of snapshot matrix

This experiment is conducted as per the data listed in Table 4.5:

Table 4.5: Data used for change of shape of snapshot matrix

Number of data points	Number of snapshots	Number of modes
1M	50	20
100K	500	200
10K	5000	2000
1K	50000	20000

4.1.8.1 Comparison of Computation, MPI and I/O Time

Increase in the total time and computational time is seen in Fig. 4.23 by reducing n by a factor of 10 and increasing m and k by a factor of 10 and keeping the total

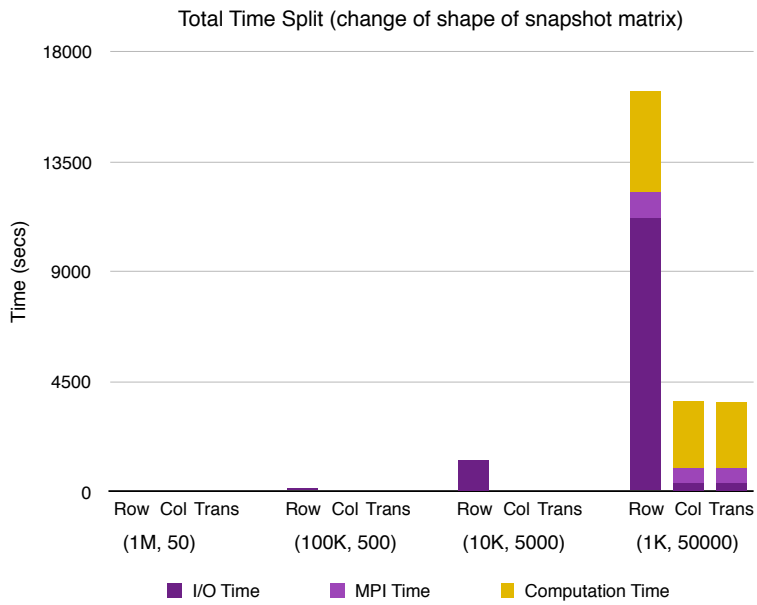

Figure 4.23: Computation, MPI, I/O Time Comparison (Change of shape of snapshot matrix)

number of data points to be constant. A steep increase in the total time for decomposing the data set (1K points, 50000 snapshots) is due to the time taken for SVD operation as shown in Fig. 4.24. This is because complexity of SVD is affected only by m since SVD is performed on co-variance matrix of size $m \times m$ as mentioned in Algo. 1. Therefore, an overall increase in computational complexity can be attributed to change in m as per Eq. 4.1.

SVD operation performed using ScaLapack is computationally expensive as seen in Fig.4.24. Thus, it becomes a major factor in computational time while decomposing large number of snapshots. Since $m >> n$ for a fat snapshot matrix, by computing the co-variance matrix as $\boldsymbol{QQ}^T$ reduces the size of matrix to $n \times n$ instead of $m \times m$. Thus, it would be possible to reduce the computational complexity and memory required by a large extent for a fat snapshot matrix.

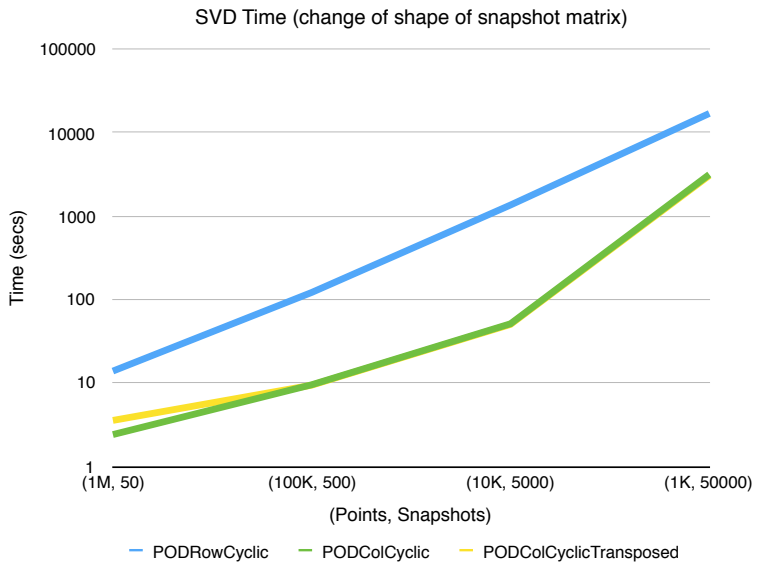

Figure 4.24: Time for SVD (Change of shape of snapshot matrix)

4.1.8.2 Total Memory Consumed

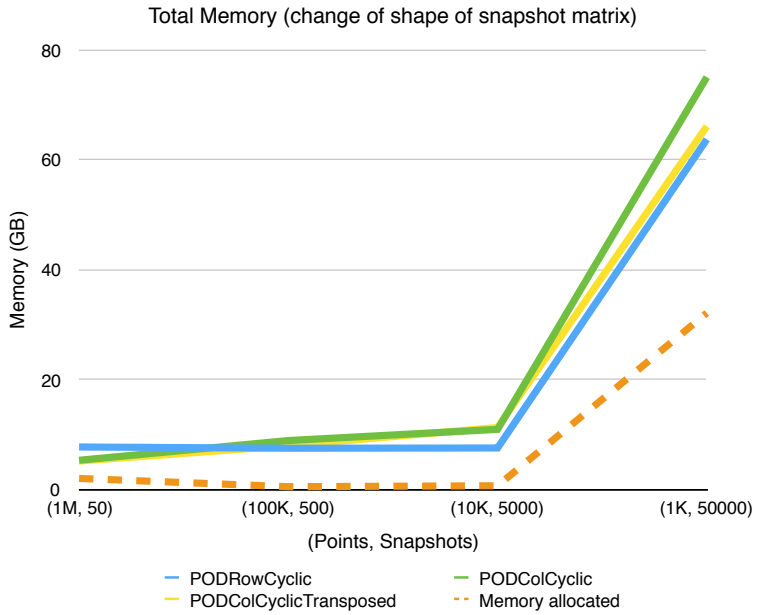

Figure 4.25: Total Memory (Change of shape of snapshot matrix)

As memory is more sensitive to change in m than n as per Eq. 4.2, a steep increase in total memory is observed as the number of snapshots is increased.

4.1.9 Varying Snapshots

This experiment is conducted as per the data listed in Table 4.6:

Table 4.6: Data used for varying number of snapshots

Number of data points	Number of modes	Number of snapshots	Number of snapshots	Number of snapshots	Number of snapshots
100M	40	50	100	200	400
1M	2000	5000	10000	20000	40000
1K	4000	10000	20000	40000	80000

4.1.9.1 Comparison of Computation, MPI and I/O Time

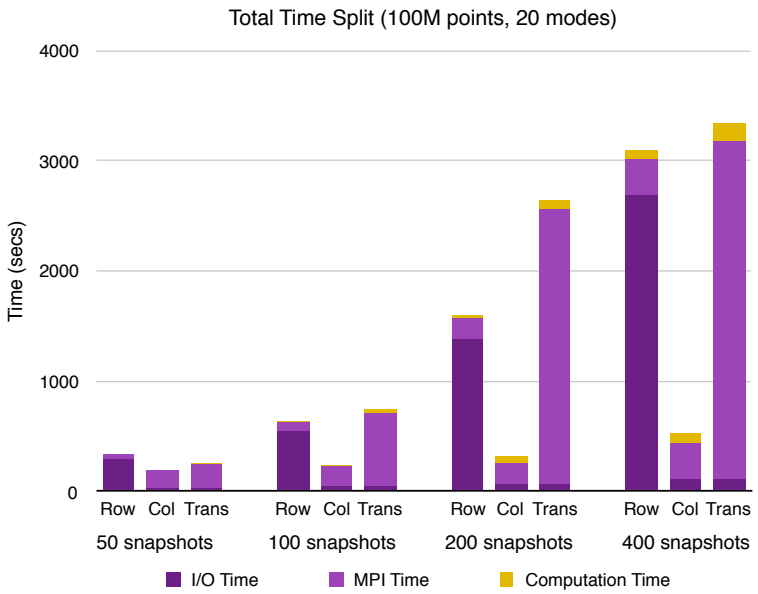

Figure 4.26: Computation, MPI, I/O Time Comparison (100M points, 100 snapshots)

Although, the scale for each of Figures in 4.26, 4.27, 4.28 differ from that in

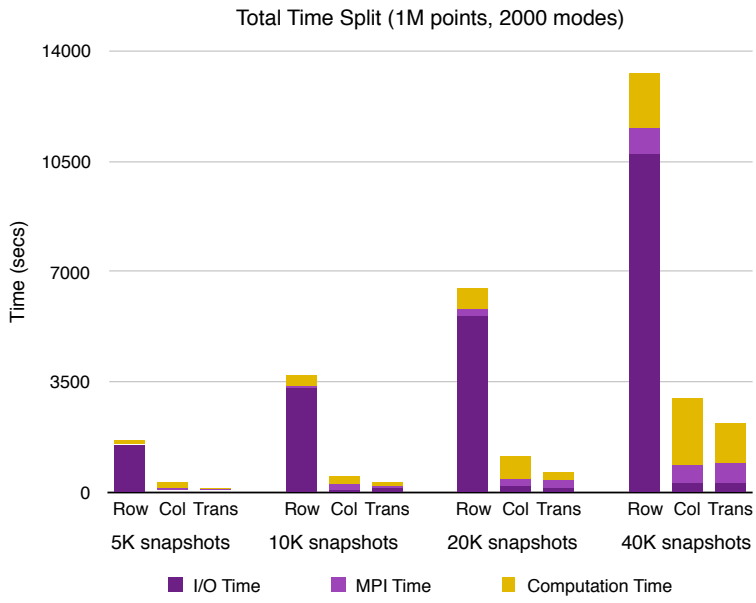

Figure 4.27: Computation, MPI, I/O Time Comparison (1M points, 5000 snapshots)

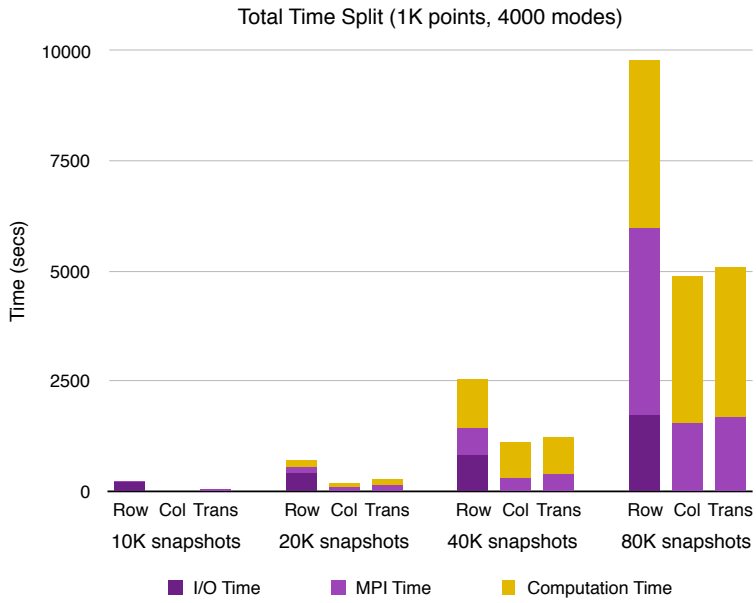

Figure 4.28: Computation, MPI, I/O Time Comparison (1K points, 10000 snapshots)

Sec. 4.1.4.1, pattern observed is still the same. Total time for each implementation almost doubles by doubling the number of snapshots. This suggests that decomposing snapshots using current implementations is scalable, however, time taken

for each implementation varies depending on the type of data set. Increase in MPI time is high for the first data set using $PODColCyclicTransposed$ and so is the I/O time for the second data set using $PODRowCyclic$ implementation. Thus, the total time for completion would exceed by many times when compared to $PODColCyclic$ implementation, which would make $PODColCyclic$ a preferred choice for all the data sets.

4.1.9.2 Total Memory

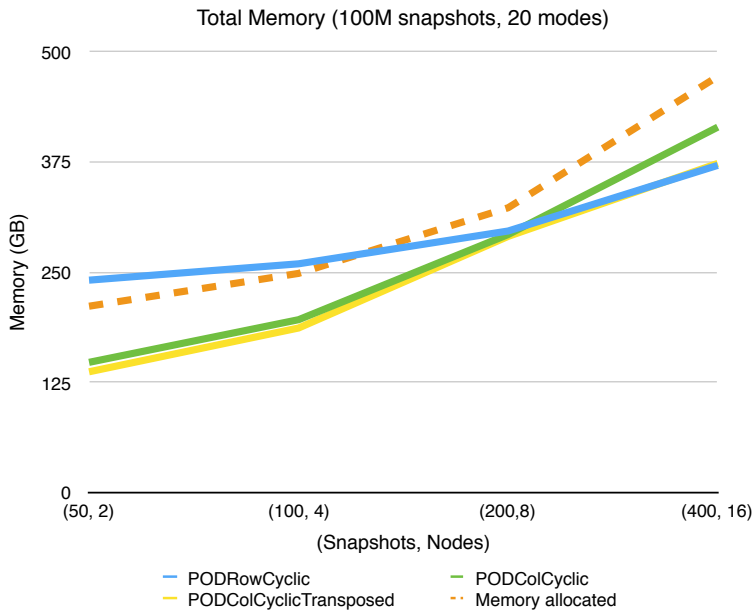

Figure 4.29: Total Memory (100M points, 100 snapshots)

As with previous results illustrated for Total Memory by varying the number of snapshots in Weak Scaling, 4.1.4.2, here too there is not much difference between implementations for various data sets. Thus, there is no advantage of choosing one implementation over another. Moreover, the dotted line plotted using Eq. 4.2

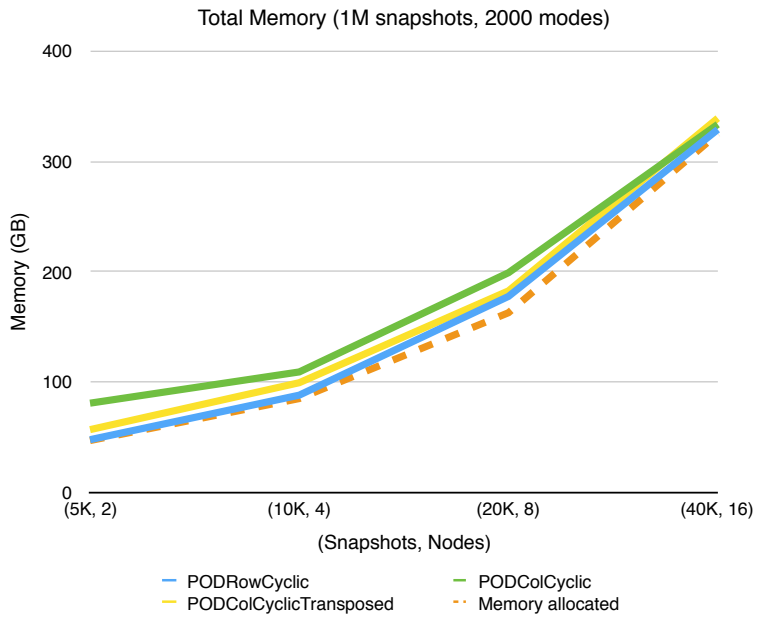

Figure 4.30: Total Memory (1M points, 5000 snapshots)

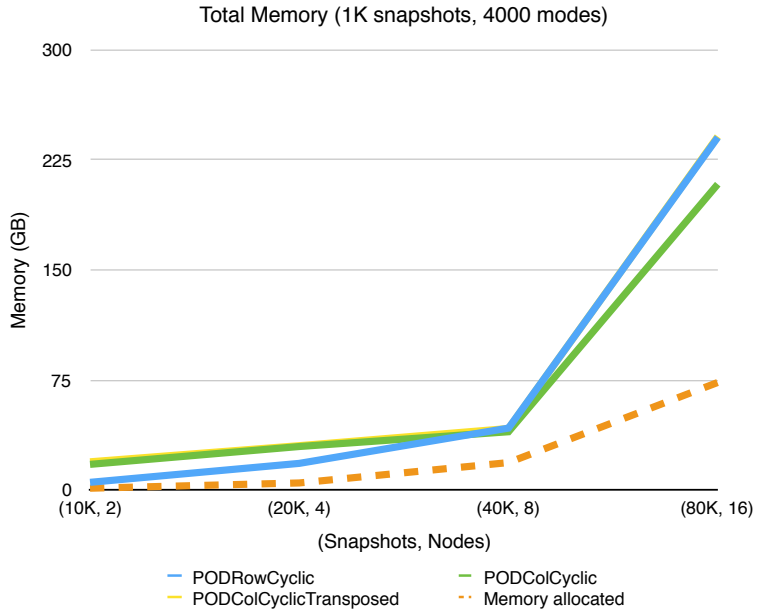

Figure 4.31: Total Memory (1K points, 10000 snapshots)

provides a good estimate of memory that would be consumed.

4.1.10 Varying Modes

This experiment is conducted as per the data listed in Table 4.7:

Table 4.7: Data used for varying number of modes

Number of data points	Number of snapshots	Number of modes	Number of modes	Number of modes	Number of modes
100M	100	10	20	40	80
1M	5000	500	1000	2000	4000
1K	10000	1000	2000	4000	8000

With this experiment, we see a similar trend with respect to Total Time and Total Memory Consumed. However, an increase in memory or time is not so substantial since, the number of modes k has least effect on $Complexity$ in Eq. 4.1 or $Memory$ in Eq. 4.2 functions. Thus, implementation that is best for least number of modes would still perform the best for highest number of modes which turns out to be $PODColCyclic$ implementation as described in the Sec. 4.1.5.

4.1.11 Inference on Experiments with Fixed Computational Resources

With the benefit of I/O time for reading snapshots and avoiding the overhead on performing $transpose$ operation, $PODColCyclic$ has turned out to be the best performing implementation. Moreover, $PODColCyclic$ also scales well by varying any of the parameters chosen to test the performance thus far.

Since the total memory consumed is same for all implementations regardless of the parameters varied, there is no gain achieved in memory consumption with any implementation.

The ability of $PODColCyclic$ to complete the execution in a reasonable amount of time suggests that it is the most preferred implementation to perform modal decomposition using POD.

4.2 Sparse Coding

There are three implementations that are currently available to compute Sparse Coding. Two of them are serial implementations in which computation of KSVD is done using either dense or sparse matrix. Better performance can be achieved with sparse implementation by using matrix multiplication routines specially designed for sparse matrices if the sparsity of the coefficient matrix, $\boldsymbol{S}^{SPC}$, is quite low. In addition to this, there is also a parallel implementation. However, results are not illustrated for each one of them since the algorithm is currently not scalable. Parameters that affect the performance are:

- Number of snapshots
- Number of sparse modes
- sparsity

This experiment is conducted by varying the number of snapshots as 100, 200,

400 and 800. The number of modes are varied as 10, 20 and 40. For each one of these combinations, results are obtained by varying sparsity as 0.25, 0.50 and 0.75.

4.2.1 Comparison of Time for Batch-OMP and KSVD

As described earlier in Algo. 2, Batch-OMP and KSVD are two major algorithms executed under the convergence function for generating sparse modes and these two result in almost entire time taken for every iteration. Since, the number of iterations required for convergence of sparse coding algorithm can vary and each iteration takes almost the same amount of time, results illustrated consider only a single iteration of the sparse coding algorithm. Also, the scale used for measuring time is different for each of the results illustrated.

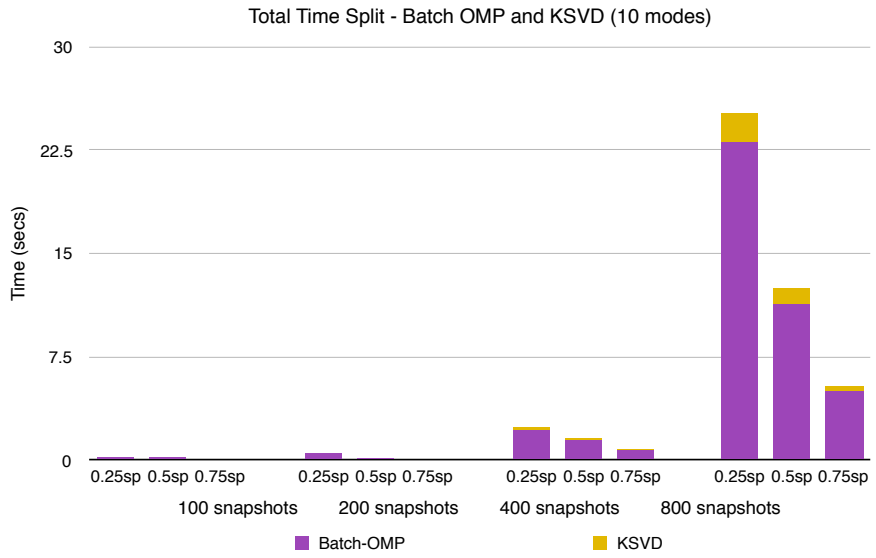

Figure 4.32: Batch-OMP and KSVD Time Comparison (10 modes)

Time taken for Batch-OMP forms a significant portion as compared to KSVD for two reasons:

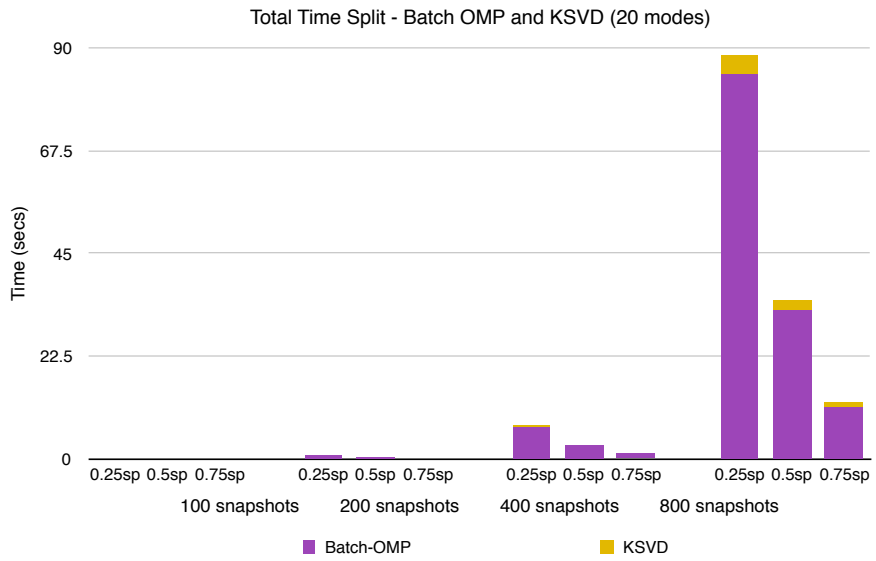

Figure 4.33: Batch-OMP and KSVD Time Comparison (20 modes)

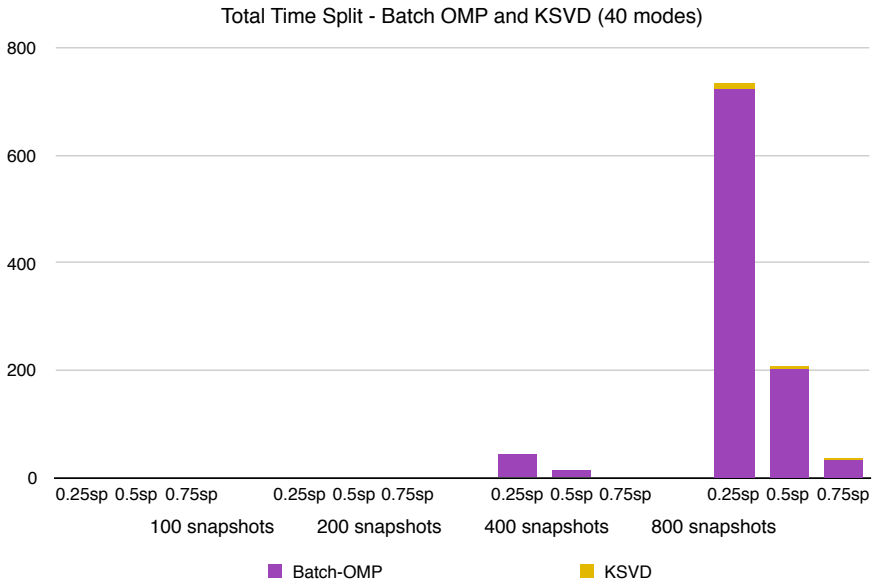

Figure 4.34: Batch-OMP and KSVD Time Comparison (40 modes)

1. Inner loop of Batch-OMP is not parallel as described in Algo. 2

2. $pseudo - inverse$ function is called for every iteration in the inner loop of Batch-OMP which in turn calls SVD. However, it is already established in the previous Sec. 4.1 that time taken for SVD increases exponentially, resulting

in exponential increase in time taken per iteration.

Increase in the number of snapshots, m, increases the time for Batch-OMP because its outer loop runs until the number of snapshots.

Inner loop of Batch-OMP iterates until the number of active modes which is computed as:

$$active\ modes = (1 - \boldsymbol{ceil}(sparsity)) * sparsemodes$$

Thus, decrease in sparsity increases the number of $active\ modes$ used in computation and hence the inner loop of Batch-OMP runs for more iterations in turn increasing the time.

As the number of modes increases, time taken for Batch-OMP increases too, because global matrix, $\boldsymbol{G}$, in Algo. 2 is of size $k \times k$. As the size of matrix $\boldsymbol{G}$ increases, time taken for SVD increases and hence the difference in time between different modes.

Therefore, Batch-OMP is affected by all the parameters. Complexity increases with increase in the number of snapshots or modes and decrease in sparsity. Also, the effect of time taken for KSVD decreases with increase in time taken for Batch-OMP. Thus, using a sparse or a dense matrix for $\boldsymbol{S}^{SPC}$ would not benefit in any way as the fraction of time required for KSVD is quite small.

Hence, current implementation of Sparse Coding is computationally very intensive and does not scale well. Since, the inner loop of Batch-OMP is not parallelized, there is no benefit gained from parallel implementation as well.

4.2.2 Total Memory

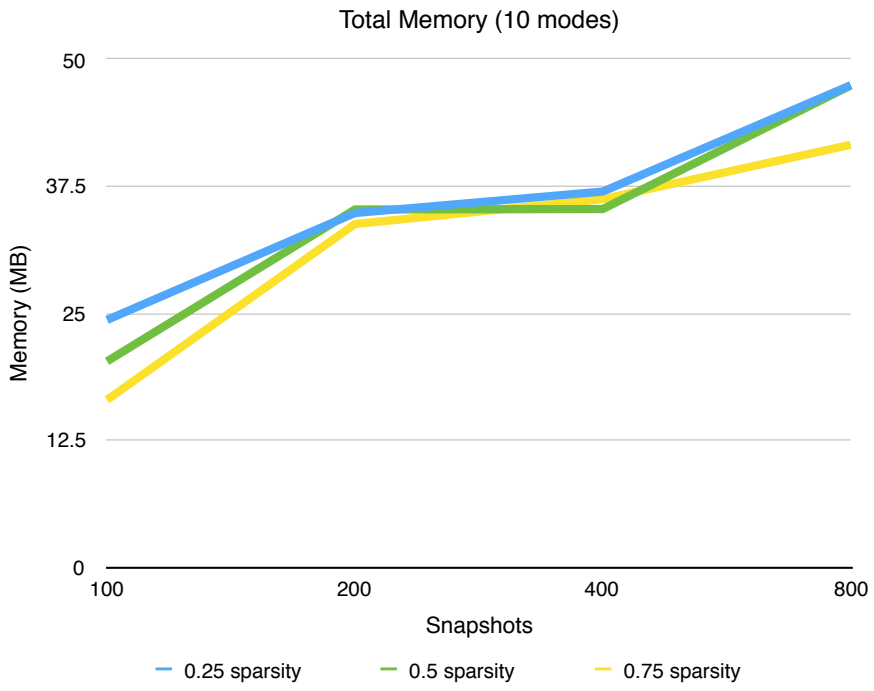

Figure 4.35: Total Memory Consumed (10 modes)

Unlike POD, memory consumed is quite less in case of sparse coding. As seen from Figures, 4.35, 4.36 and 4.37, only a few MB is consumed even for 800 snapshots. Unless, sparse coding is computed for as high as 100K snapshots, it is preferred to use serial execution than parallel since there is no gain in parallel implementation currently with exponential time required for Batch-OMP algorithm. Since the two matrices that impact the use of memory, namely, POD coefficients and sparse modes, are dependent on the number of snapshots and number of modes, increasing one of them increases the total memory consumed.

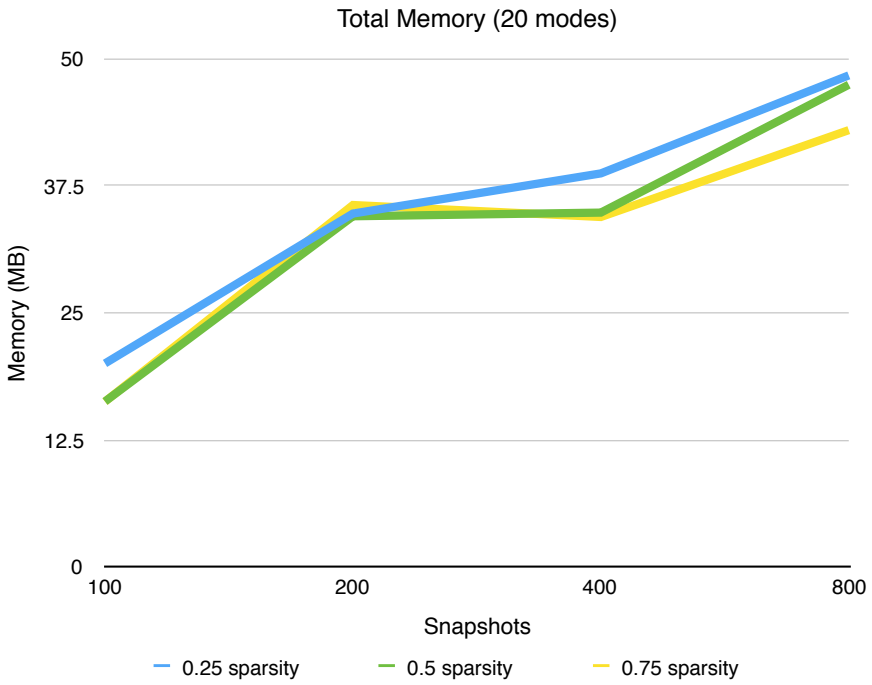

Figure 4.36: Total Memory Consumed (20 modes)

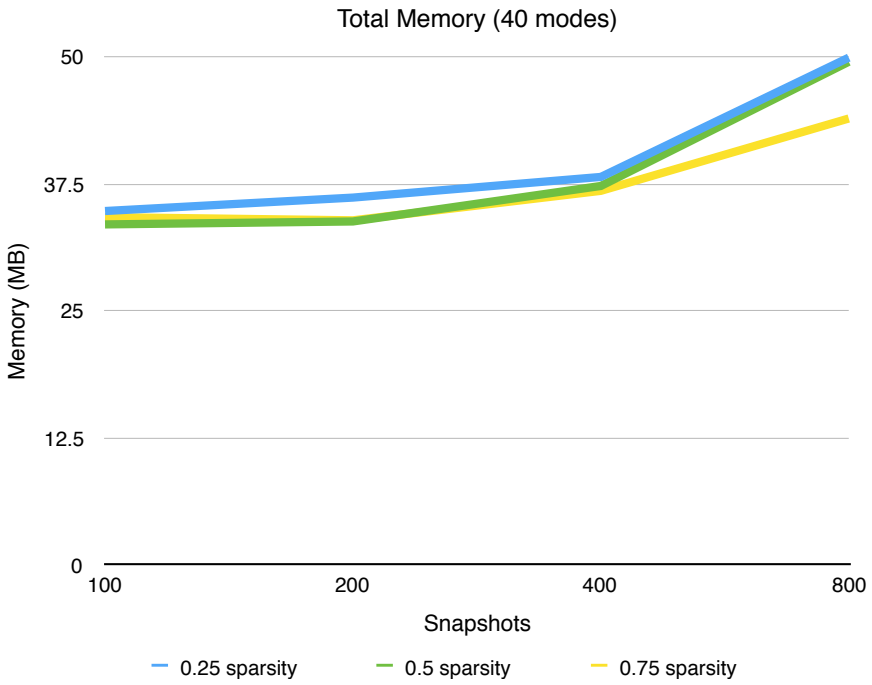

Figure 4.37: Total Memory Consumed (40 modes)

Chapter 5

Conclusions

A systematic characterization of various parallel implementations of Proper Orthogonal Decomposition(POD) was done and based on the inferences from various experiments performed, $PODColCyclic$ implementation shows good promise in terms of memory consumed and time taken to decompose large data sets. However, $PODColCyclic$ can be used only when the number of data points in a single snapshot is less than the size of an interger. In that case, $PODRowCyclic$ implementation becomes the preferred choice compared to $PODColCyclicTransposed$ since $transpose$ operation is quite expensive for tall snapshot matrices. Hence, $PODColCyclic$ is a preferred choice for all other cases when it comes to achieving scalability. Issues related to $PODRowCyclic$ and $PODColCyclicTransposed$ are emphasized with its usage limited to only certain types of data sets.

Another important factor to consider is the computational resources that must be utilized. Since decomposition using POD is communication intensive, using more resources than required to decompose the data sets could prove counter pro-

ductive. From various experiments, it is established that POD can be executed once the total memory allocated in the code occupies half the user available memory in nodes leaving the rest for computational purposes for ScaLapack.

Sparse Coding on the other hand is seen a computationally intensive than being memory intensive. Current state of the algorithm, most importantly Batch-OMP, does not allow the algorithm to scale well since time taken for computation of each iteration increases exponentially with increase in snapshots.

Thus, for performing modal decomposition of massive snapshots, POD is currently a viable option leaving room for improvement of the sparse coding algorithm.

Chapter 6

Recommendations for Future Work

6.1 Proper Orthogonal Decomposition

Since, POD consists of several steps that include linear algebraic operations such as Matrix Multiplications, Matrix Transpose and SVD (Singular value decomposition), each step being sensitive to performance parameters in a different way, it is highly difficult to find the overall benefits of a particular implementation. It was observed that $PODRowCyclic$ implementation would perform better for computing the reconstruction error, however, $PODColCyclic$ implementation for computation of POD modes. Moreover, this trend will not hold true for any type of data set, as the computations for a particular step might outperform for one of the implementations but at the same time perform worse for a different sized data set. Given that there are multiple implementations, it is important to evaluate each implementation not only as a whole as done in the current research but also for each step of the algorithm by executing strong and weak scaling experiments.

It is beneficial to come up with a complexity function that selects the best implementation, given a certain set of parameters. This could be integrated with the code and the best performing implementation is executed without the need for the user to decide. In addition to this, having the code to output optimum number of processes to be executed with would be very productive since it is already observed from the strong scaling experiments that increasing computing resources keeping the data fixed would degrade the performance after an optimum level is reached owing to increase in MPI time.

6.2 Sparse Coding

Currently, $pseudo - inverse(pinv)$ is a major roadblock to the scalability of sparse coding algorithm. This is due to the fact that it internally uses SVD which does not scale well with the increase in number of snapshots. Thus, it is imperative to come up with an alternative approach of performing $pseudo-inverse$ to run sparse coding on larger data sets.

Currently, Batch-OMP takes the majority of time in executing the algorithm as it has not been fully parallelized. By allowing each process to independently run a part of this algorithm without any inter-dependencies would make it scalable by distributing work among the processes.

Bibliography

[1] Rohit, D., Model Order Reduction of Incompressible Turbulent Flows, Ph.D. Thesis, The Ohio State University, Columbus, Ohio, 2017

[2] Holmes, P., and Lumley, J. L., and Berkooz, G., and Rowley, C.W., Turbulence, coherent structures, dynamical systems and symmetry, *Cambridge university press*, New York, 2nd ed., 2012

[3] Message Passing Interface (MPI), *http://mpi-forum.org/*

[4] Blackford, L. S., Choi, J., Cleary, A., DAzeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C., *ScaLAPACK Users Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[5] Cormen, T. H., *Introduction to algorithms,* MIT press, 2009.

[6] Rubinstein, R., Zibulevsky, M., and Elad, M., *Efficient implementation of the K-SVD algorithm using batch orthogonal match- ing pursuit,* CS Technion, Vol. 40, No. 8, 2008, pp. 115.

[7] Aharon, M., Elad, M., and Bruckstein, A., *K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Repre- sentation,* Signal Processing, IEEE Transactions on, Vol. 54, No. 11, 2006, pp. 43114322.

[8] Lucia, D. J., and Beran, P. S., and Silva, W. A., Reduced-order modeling: new approaches for computational physics *Progress in Aerospace Sciences*, Vol 40, No. 1, 2004, pp. 51-117

[9] Sirovich, L., Turbulence and the dynamics of coherent structures. I-Coherent structures. II-Symmetries and transformations. III-Dynamics and scaling, *Quarterly of applied mathematics*, Vol. 45, 1987, pp. 561-571

[10] Liang, Z., and Deshmukh, R., and McNamara, J. J. , and Wytock, M., and Kolter, J. Z., Expedient and Parallelizable Sparse Coding Algorithm for Large Datasets, *AIAA Paper 2016-0463*, Structures, Structural Dynamics, and Materials Conference, 2016

[11] Olshausen, B. and Field, D., *Emergence of simple-cell receptive field prop- erties by learning a sparse code for natural images,* Nature, Vol. 381, No. 6583, 1996, pp. 607609.

[12] Kreutz-Delgado, K., Murray, J. F., Rao, B. D., Engan, K., Lee, T., and Sejnowski, T. J., *Dictionary learning algorithms for sparse representation,* Neural computation, Vol. 15, No. 2, 2003, pp. 349396.

[13] Golan J.S., *MoorePenrose Pseudoinverses. In: The Linear Algebra a Beginning Graduate Student Ought to Know*, Springer, Dordrecht, 2012.

[14] Liang, Z., Deshmukh, R., McNamara, J. J. , Wytock, M., and Kolter, J. Z., Expedient and Parallelizable Sparse Coding Algorithm for Large Datasets, *AIAA Paper 2016-0463,* 2016.

[15] K. J. Friston, C. D. Frith, P. F. Liddle, R. S. J. Frackowiak, Functional Connectivity: The Principal-Component Analysis of Large (PET) Data Sets, *MRC Cyclotron Unit*, Hammersmith Hospital, London, U.K, 1993.

[16] Xiuwen Zheng, David Levine, Jess Shen, Stephanie M. Gogarten, Cathy Laurie, Bruce S. Wei A high-performance computing toolset for relatedness and principal component analysis of SNP data, *Genetics and population analysis*, Department of Biostatistics, University of Washington, Seattle, WA 98195-7232, USA, 2012.

[17] Badrul Sarwar, George Karypis, Joseph Konstan, John Riedl, Analysis of Recommendation Algorithms for E-Commerce, *GroupLens Research Group / Army HPC Research Center*, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 5545, 2000.

[18] Hairong Qi, Tsei-Wei Wang, J. Douglas, Birdwell, Global Principal Component Analysis for Dimensionality Reduction in Distributed Data Mining, University of Tennessee, Knoxville, TN, 37996, USA, 2004.

[19] R. Choy and A. Edelman, *Parallel Matlab: Doing it right*, http://www-math.mit.edu/ edelman/homepage/papers/pmatlab.pdf, Nov 2003.

[20] A. Trefethen, V. Menon, MultiMATLAB: integrating MATLAB with high-performance parallel computing, *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, Pages: 1 - 18, ISBN:0-89791-985-8, San Jose, CA, 1997.

[21] Hillol Kargupta, Weiyun Huang, Krishnamoorthy Sivakumar, Erik Johnson, Distributed Clustering Using Collective Principal Component Analysis, *Knowledge and Information Systems*, Springer-Verlag, 2001.

[22] Hillol Kargupta, Weiyun Huang, Krishnamoorthy Sivakumar, Byung-Hoon Park, and Shuren Wang, Collective Principal Component Analysis from Distributed, Heterogeneous Data, *School of Electrical Engineering and Computer Science*, Washington State University, Pullman, WA 99164-2752, USA, 2000.

[23] Ordonez, C., Mohanam, N., Garcia-Alvarado, C., Distrib Parallel Databases, 32: 377., https://doi.org/10.1007/s10619-013-7134-6, Houston, TX 77204, USA, 2014.

[24] Liu, W., Zhang, H., Tao, D. et al., Multimed Tools Appl (2016) 75: 1481, https://doi.org/10.1007/s11042-014-2004-4

[25] J. M. Porta, J. J. Verbeek, and B. J. Krse., Active appearance-based robot localization using stereo vision., *Autonomous Robots*, 18(1), Springer Verlag, 2005

[26] Mahout machine learning library: http://mahout.apache.org/.

[27] MLlib machine learning library: https://spark.apache.org/mllib/.

[28] J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in Proc. 6th Conf.Symp. on *Operating Systems Design and Implementation*, 2004, pp. 1010.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster Computing with Working Sets, in Proc. 2nd USENIX Conf. on *Hot Topics in Cloud Computing*, 2010.

[30] R Development Core Team. An Introduction to R. June 2004, http://www.r-project.org

[31] R Development Core Team. Writing R Extensions. June 2004, http://cran.r-project.org/manuals.html

[32] Jeremy Kepner, Stan Ahalt, MatlabMPI, *Short communication*, 2004

[33] Tarek Elgamal and Mohamed Hefeeda, Analysis of PCA Algorithms in Distributed Environments, *Qatar Computing Research Institute*, Qatar Foundation, Doha, Qatar, 20 April 2015.

[34] Tarek Elgamal, Maysam Yabandeh, Ashraf Aboulnaga, Waleed Mustafa, Mohamed Hefeeda, sPCA: Scalable Principal Component Analysis for Big Data on Distributed Platforms, *ACM SIGMOD International Conference on Management of Data,* Qatar Computing Research Institute, Twitter, NTG Clarity, 2015, pp. 79-91.

[35] Christos Boutsidis, David P. Woodruff, Communication-optimal Distributed Principal Component Analysis in the Column-partition Model, Yahoo Labs, IBM Research, 25 April 2015.

[36] Alex Gittens, Aditya Devarakonda, Evan Racah, Michael Ringenburg, Lisa Gerhardt, Matrix Factorizations at Scale: a Comparison of Scientific Data Analytics in Spark and C+MPI Using Three Case Studies *2016 IEEE International Conference on Big Data (Big Data)*, 2016, IEEE 204