MPI Parallel Computing on Eigensystems of Small Signal Stability Analysis for Large Interconnected Power Grids

Yu Ming Jiao

M. Eng.
Electrical & Computer Engineering

McGill University
Montreal, Quebec, Canada
Aug. 20, 2010

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of master of engineering

Dedication

This document is dedicated to the graduate students of the McGill University.

Acknowledgement

I would like first and foremost to express my thankfulness and deepest gratitude to my supervisor, Prof. Boon-Teck, Ooi, for his contribution and instructions on this project. Special thanks will then go to Dr. Hadi M. Banakar, previous consultant in power engineering group of McGill, who has provided necessary instructions for the early stage of this project. Special thanks will then definitely go to Mr. Francois Guretin, HPC analyst from RQCHP, who has provided me instructions for running jobs on Mammouth Series II cluster and improved the efficiency of my MPI parallel codes. Special thanks will also go to Dr. Suzane Talon, assistant of director of RQCHP and her wonderful team, who have provided kindly introductory seminar for parallel computing in summer 2009. Special thanks go to the technical staff in CLUMEQ super-computer center for their assistance of running jobs on Krylov cluster. I would also appreciate all the instructors in my undergraduate and graduate study in McGill, from whom I have obtained necessary theoretical knowledge to accomplish this project, especially Prof. Xiao-Wen Chang, Prof. Francisco D. Galiana, Prof. Dennis Giannacopoulos, Prof. Geza Joos, Prof. Steve McFee, Prof. Hannah Michalska and Prof. Boon-Teck, Ooi for my graduate courses study. I would thank all the graduate students in power engineering group and it was a pleasure to be studying together with them. Especially, I would like to express my thankfulness to Mr. Etienne Veilleux for editing the abstract in French of this thesis. And last but not least, special thanks will go to my beloved wife who has supported me all the time for study and research in McGill during the past four years.

Table of Contents

List of Tables

List of Figures

List of Acronyms

MPI – Message Passing Interface

PC – Personal Computer

ROM – Rank-One Modification

PMU – Phase Measurement Unit

WAM – Wide-area Measurement

WAC – Wide-area Control

SCDA – Supervisory Control and Data Acquisition

B & B – Break and Bind

CLUMEQ - Consortium Laval, Université du Québec, McGill and Eastern Quebec

RQCHP –Reseaux Quebecois Calcul de Haute Performance

DC – Divide and Conquer

HPC – High Performance Computing

HPTC – High Performance Technical Computing

FLOP – Float point operations

PBS – Portable Batch System

I/O – Input and Output

RTM – Rank-Two Modification

Abstract

Eigenanalysis is widely used in power system stability study. With PC technologies available today, it takes long time to compute the entire eigensystems of large interconnected power grids. Since power transmission lines are connected & disconnected and line loads keep changing frequently, tracking eigensystems in real-time requires parallel computation. Recently, a parallel eigensystem computation method, the Break and Bind (B & B) method, has been proposed by Dr. H. M. Banakar in McGill University. This method is viewing connection of two isolated sub-networks as being equivalent to a rank-one modification (ROM) of the stiffness matrix and considering the two sub-networks as a single entity. Research of this thesis consists of implementing the B & B method based on Message Passing Interface (MPI) parallel programming in #C. The developed MPI codes were executed on super-computers - Krylov cluster of CLUMEQ and Mammouth Series II cluster of RQCHP. The testing results have demonstrated that the eigensystem of a power system composed of around 4,000 generators can be updated within two seconds.

Résumé

L’analyse des valeurs propres est largement utilisée dans les études de stabilité des réseaux électriques. En utilisant les ordinateurs personnels disponibles aujourd'hui, le calcul de la totalité des valeurs propres de plusieurs grands réseaux électriques interconnectés requiert beaucoup de temps. Étant donné que les lignes de transport d'électricité sont connectées et déconnectées et que les charges ne cessent de varier, le suivi des valeurs propres en temps réel nécessite des calculs en parallèles. Récemment, une méthode de calcul en parallèle des valeurs propres, la Break et Bind (B & B), a été proposée par le Dr. H. M. Banakar à l'Université McGill. Cette méthode voit la connexion de deux sous-réseaux isolés comme étant équivalent à une modification de rang un de la matrice de raideur et considère les deux sous-réseaux comme une entité entière. La recherche de cette thèse consiste à implanter la méthode B & B avec une programmation parallèle en #C basé sur l’interface Message Passing Interface (MPI). Le code de programmation développé en MPI a été exécuté avec des superordinateurs - Krylov de CLUMEQ et Mammouth série II de RQCHP. Les résultats des tests ont démontrés que les valeurs propres d'un système composé d'environ 4,000 alternateurs peuvent être calculées à l’intérieur de deux secondes.

Chapter 1 Introduction

1.1 Background

Modern power systems are steadily growing as load demand increases. For dynamic systems to be stable, they must have equilibrium operating states and must return to their equilibrium operating states after disturbances. Therefore power system stability analysis is important. Frequent sources of disturbances in power systems are faults in the transmission lines. Study on these comes under "Transient Stability Analysis" [1], which is dealt with by numerically integrating the simplified equations modeling the entire power system network. "Transient Stability Analysis" programs are usually run for short durations, long enough for the fault transients to be damped so that the system will be considered to be stable. However, if the "Transient Stability Analysis" programs are allowed to continue simulating, disturbances may emerge from the equilibrium states and grow exponentially with time. This happens when the power system is not stable from small signal stability point of view. The exponential growth from small perturbations may never return to the equilibrium steady-state.

Small signal stability analysis proceeds in two stages:

1. Solving for the equilibrium state vector $\underline{x_0}$ from the dynamic equations $\dot{\underline{x}} = \underline{f}(\underline{x},\underline{u})$ which model the power system.
2. Applying small signal perturbation $\underline{x} = \underline{x_0} + \underline{\Delta x}$ and small signal linearization to derive a set of equations $\underline{\Delta \dot{x}} = [A]\underline{\Delta x}$, where $[A] = \left.\frac{\partial \underline{f}(\underline{x},\underline{u})}{\partial \underline{x}}\right|_{\underline{x_0}}$

This thesis focuses on computing the eigensystem of $[A]$. Because a power system can have thousands of generators, N, the dimension of $[A]$, is very large. Computing the entire eigensystem of [A] is time-consuming and usually costs around O (N^3) [2]. Furthermore, since the parameters in $[A]$ keep changing as the

load changes and lines are connected & disconnected in the power system on hourly or daily basis, the main purpose of this study is to meet the on-line update requirement. Two obstacles have to be overcome in this task:

1. Computation of eigensystems takes long time.
2. Large scale problems require large memory storage for the computation

Because of recent implementations of Phase Measurement Units (PMU) [3] and Wide-Area Measurement (WAMs) [4], there is intention of putting the power system under Wide-Area Control (WAC). The integrity of the Wide-Area system requires that it is stable from the small signal stability viewpoint to begin with. It means that the real parts of all the eigenvalues of [A] must lie in the left side of the complex s-plane.

The Wide-Area Control requires on-line updating and monitoring the power system dynamics. The goal is achieved by obtaining eigensystems of the state matrix. According to supervisory control and data acquisition (SCDA) [5] measurement cycles, on-line updates must be completed within seconds. Such challenging requirements can be met by super-computers, which can update the eigensystems in a parallel way. In addition super-computers usually have abundant memories.

To date, parallel computing has not made significant impact in power system studies. Many researchers in this field have strived to develop efficient parallel algorithms to compute selected or partial eigenvalues [6, 7and 8] and improve QR algorithm for eigenvector computation [9] for the purpose of small signal analysis. Recently, Dr. Hadi Mohamed Banakar addressed parallel processing of the entire eigensystems by proposing what he calls the Break and Bind (B & B) method [10]. He uses the results of the original work by Bunch, Nielsen, and Sorensen [11] to update the eigensystem of a symmetric matrix that is subjected to ROM. Bunch, Nielsen and Sorensen were concerned about computing eigensystems of symmetric tri-diagonal matrices, encountered in the last stage of computing eigensystems for symmetric matrices. The B & B method is superior to other methods in the following ways:

- The entire eigensystem of non-symmetric matrix [A] could be computed directly from those before ROMs.
- The proposed mathematical models can be implemented efficiently in a parallel processing environment.

This thesis focuses on the development of efficient parallel computing algorithms which implement the B & B method. The codes were developed based on MPI in #C and executed on 1) Krylov cluster of super-computer center CLUMEQ – [Consortium Laval, Université du Québec, McGill and Eastern Quebec] and 2) Mommounth Series II cluster of super-computer center RQCHP – [Reseaux Quebecois Calcul de Haute Performance].

Up to this point in time, Dr. Banakar has only demonstrated the capability of the B & B method in a small system using a PC. For the power system community to take notice, it is necessary to demonstrate that an update for a system of size $N=4,000$ can be accomplished within the measurement cycles of a few seconds. The research results showed that this demanding objective has been achieved.

The work required to succeed consisted of:

- Mastering protocols of MPI and MPI routines
- Being proficient in MPI parallel programming in #C based on numerical methods
- Having ability to deal with large scale data both on PC and super-computers
- Possessing certain mathematical derivation & analytical skills

1.2Current Techniques for Computing Eigensystems

Eigenproblems are frequently encountered in engineering field. For example, in power engineering, oscillation modes following perturbations in power systems are characterized by the eigenvalues of the [A] matrix. Therefore it is important to have efficient computer algorithms to compute eigensystems for large matrices in a relatively short period. To date, numerous algorithms for computing eigensystems are available, among which the most famous one is QR algorithm [John G.F. Francis, 1961 and Vera Nikolaevna Kublanovskaya, 1961] and it is also one of the top 10 algorithms in the 20^{th} century. Based on the author's knowledge, commercial software, MATLAB, employs this algorithm to compute

eigensystems of all kinds of matrices including non-symmetric ones. Besides QR, there are also some other algorithms for computing eigensystems of matrices with special structures, e.g. symmetric and Hermitian matrices, among which the most famous ones are the Jacobi method and the Lanczos method [2]. However, all the algorithms above are called iterative methods due to the following facts:

- When the degree *n* of characteristic polynomials of the matrix is greater than 4, there are no closed form formulae for the roots.
- Regarding eigenvector computation, even if the eigenvalues are known, there are no closed form solutions either. Usually the commonly used numerical methods to compute eigenvectors are power method or inverse iterative methods [2].

For iterative methods, the cost of computation is very expensive. Roughly speaking, the cost for QR algorithm to compute the entire eigensystems is around $25N^3$ and if only the eigenvalues are to be computed, the cost is around $10N^3$ [2]. As mentioned earlier, this thesis proposed an efficient parallel algorithm which can update the eigensystems of large sparse symmetric matrices subject to ROMs within a few seconds based on the B & B method [10]. The findings will definitely have a bright future in power engineering applications because the real-time updated eigensystems inform operators the status of small signal stability of the system.

1.3 Introduction to HPC and Canadian HPC Consortia

High Performance Computing (HPC) is to deal with large scale problems via super-computers or computer clusters. It usually refers to parallel computing. High-Performance Technical Computing (HPTC) is being applied to biotechnology, medicine, aerospace, nanotechnology, environmental research, engineering etc. Usually a super-computer is composed of hundreds or even thousands of processors. For example, Mammouth series II, which is one of the super-computers used in this research, has 616 Intel Xeon quad-cores. When solving large scale problems on a single-processor machine or PC, there are two major obstacles that one will encounter very often:

- There is no enough memory space to store the data and variables during the computation.
- Computation takes long time.

For instance, when one wants to compute the eigensystems of a symmetric matrix $A \in R^{5000X5000}$ or even larger by **eig** function of MATLAB using a PC, usually the error message "Out of memory" will occur. On the other hand, during the research, the time for computing eigensystems based on MATLAB built-in function **eig** of the 4062-node system (see Chapter 5) was recorded, i.e. $A \in R^{4062X4062}$. The total computing time based on MATLAB **eig** is around 336 seconds. With parallel computing, the total updating time could be reduced to 1.48 seconds with 72 processors implementation. The detailed discussions and analyses are found in 5.1.2. The two problems discussed above can easily be tackled with super-computers since a supercomputer or computer cluster usually has much larger memory storage either for a shared-memory or a distributed-memory machine. Furthermore the total computation tasks can be divided into relatively smaller ones to be implemented simultaneously on different processors. A natural question regarding supercomputer is that how these processors will communicate with each other during the computation? Communication among the processors is facilitated by predefined protocols.

Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) are two commonly used protocols. Basically, these two standards are collections of routines or libraries that could be implemented in #C, C++ or FORTRAN. Based on the author's knowledge, MPI is suitable for clusters with distributed memory, while on the other hand, OpenMP is for shared memory machines. The parallel algorithms developed in this thesis are based on MPI in #C. Detailed descriptions of MPI routines are available in many parallel computing books e.g. [12, 13 and 14] and in website resources, e.g. [15]. A discussion sharing the author's insight is presented in Chapter 4.

As supercomputers or computer clusters are expensive, they are affordable by consortia of users. Compute Canada is leading the creation of a powerful HPC national platform for research [Compute Canada]. It has 7 consortia in total

shown in Fig.1.1. The MPI codes developed in this thesis were executed on Krylov cluster of CLUMEQ and Mammouth Series II cluster of RQCHP. Detailed system parameters regarding these two super-computers are listed in Appendix A.

- **ACEnet - Atlantic Computational Excellence Network**
- **CLUMEQ – Consortium Laval, University of Quebec, McGill and Eastern Quebec**
- **RQCHP – Reseaux Quebecois Calcul de Haute Performance**
- **HPCVL – High Performance Computing Virtual Laboratory**
- **SciNet**
- **SHARCNET – Shared Hierarchical Academic Research Computing Network**
- **WestGrid – Western Canada Research Grid**

Fig. 1.1 Canadian Consortia – Compute Canada

1.4 Thesis Contribution

This thesis proposed a few computing algorithms based on MPI in #C for updating eigensystems of large sparse symmetric matrices subject to ROMs. The following tasks have been fulfilled during the course of research.

1. Developed a technique to create raw data of eigensystems with distinct eigenvalues of large sparse symmetric matrices up to *4,419X 4,419*. It is based on MATLAB scripting codes. Note that the eigenvalues must remain distinct before and after ROMs in order for the proposed method to work.
2. Developed a few efficient MPI parallel algorithms in #C to update eigensystems of the stiffness matrix $[K]$ based on proposed mathematical models [10]. The updating process can be completed within two seconds for a 4062-node system subject to one ROM.
3. Developed a function routine which uses two communicators to compute simultaneously.
4. Developed efficient methods to compute the largest eigenvalue based on [11].
5. Conducted detailed timing analysis for the designed algorithms.
6. Performed preliminary error analysis.

In addition, this thesis is written to serve as a guide to parallel programming based on MPI in C. A few algorithms which employ frequently encountered MPI routines are illustrated and explained.

1.5 Thesis Outline

Chapter 2 summarizes power systems background for the proposed mathematical models - B & B method and its applications in small signal stability analysis.

Chapter 3 Details theoretical foundation of the proposed approach and related numerical analysis. It draws attention that parallel computing is a great opportunity to obtain quick answers.

Chapter 4 gives an overview of parallel computing, detailed MPI routines implementations and designed MPI parallel algorithms.

Chapter 5 presents illustrative examples, summarizes the results and conducts detailed timing analysis & preliminary error analysis.

Chapter 6 summarizes the thesis, main conclusions and possible future research areas.

Chapter 2 Break and Bind Method

This chapter details the mathematical models used in small signal stability analysis of power systems in preparation for the research on how the analysis can be numerically implemented via parallel computing. As Dr. H. M. Banakar's B & B method has not been published yet, it is necessary to present a summary of the portions relevant to the objectives of the thesis. The similar mathematical derivation and examples of applications can also be found in [10].

2.1 Power System Background

A. Power Network Model

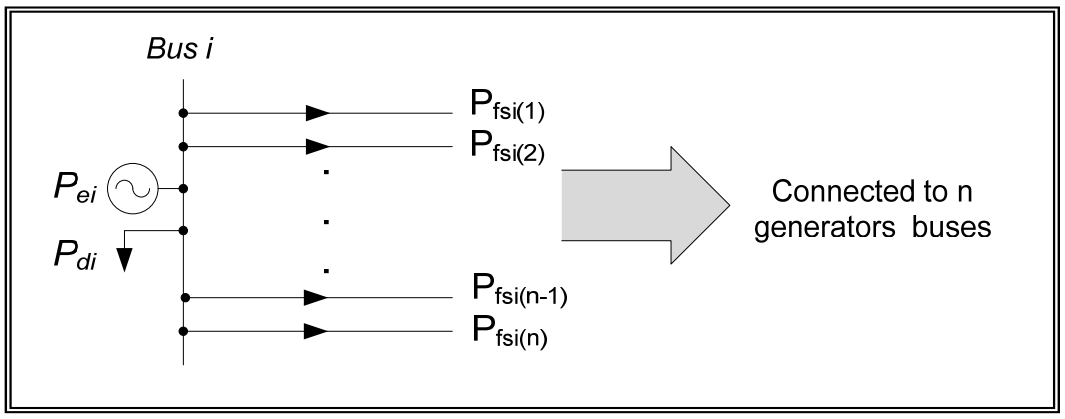

Fig. 2.1 Power network schematic - *bus i*

For a given power system, suppose it is composed of N_L transmission lines connecting N_B buses. The real power balance equation at each PV bus can be expressed as,

$$p_{e_i} - p_{d_i} = \sum_{k \in S_i} p_{f_k} \qquad i = 1,..., N_B . \tag{2.1.1}$$

where p_{e_i} - electrical power delivered by generator i; and p_{d_i} - load demand at bus i; S_i – Indices vector contains all the bus indices connected to bus i. Fig 2.1

shows a simple schematic of network model for bus i to which n generator buses are connected and the buses indices are contained in the vector S_i.

The line power flow, p_{f_k}, is defined by,

$$p_{f_k} = V_i V_j [B_k \sin\theta_k + G_k \cos\theta_k] - \frac{1}{2} G_k (V_i^2 + V_j^2) \qquad (i,j) \in k \tag{2.1.2}$$

If line k is lossless, then (2.1.2) simplifies

$$p_{f_k} = V_i V_j B_k \sin\theta_k \qquad (i,j) \in k \tag{2.1.3}$$

where B_k - susceptance of line k; G_k - conductance of line k; V_i - voltage magnitude at bus i; i and j are respectively the sending and receiving end buses indices of line k; $\theta_k = \delta_i - \delta_j$ is the angle difference between these two buses. Furthermore, (2.1.1) can be expressed in matrix form as,

$$\underline{P}_e - \underline{P}_d = [\Im]\underline{P}_f \tag{2.1.4}$$

where $[\Im]$ is the incidence matrix and satisfies,

$$[\Im]^T \underline{\ell} = \underline{0} \tag{2.1.5}$$

where $\underline{\ell}$ is defined as an $N_B \times 1$ vector of 1's.

Since voltage controls respond to system disturbances relatively fast, one can assume that bus voltage magnitudes remain constant during system oscillations, i.e. the principle of perturbation analysis of non-linear models applies here.

B. Generator Dynamic Model

In the study of power system oscillations, system dynamics is defined by, in terms of the rotor dynamics of on-line generators,

$$\frac{2H_i}{\omega_s}\frac{d\omega_i}{dt} = P_{m_i} - P_{e_i} \qquad i = 1,2,\cdots,N_G \tag{2.1.6}$$

where N_G - number of generators; ω_s - system synchronous speed; H_i - normalized inertia of the i^{th} generator.

C. Power Flow Equations

The power system small signal model requires linearizing the system of non-linear relations around a steady-state operating point. The steady-state operating point is based on

$$\frac{d\omega_i}{dt} = 0 \qquad i = 1,2,\cdots,N_G \tag{2.1.7}$$

and it is obtained by solving the nonlinear algebraic equations

$$P_{m_i} - P_{e_i} = 0.0 \qquad i = 1,...,N_G \tag{2.1.8}$$

The thesis assumes that they will be solved by load flow solvers. In the solutions the steady-state operating angles are identified by a superscript '*0*', e.g. θ_j^0. In small perturbation linearization, for angle θ_k=δ_i - δ_j, one has $\Delta\theta_k = \Delta\delta_i - \Delta\delta_j$. The perturbation angles can be put in matrix form as

$$\Delta\underline{\theta} = [\Im]^T \Delta\underline{\delta} \tag{2.1.9}$$

Taking the perturbation of (2.1.3) with respect to θ_k, one has

$$\Delta p_{f_k} = \alpha_k \Delta\theta_k \tag{2.1.10}$$

where,

$$\alpha_k = V_i V_j B_k \cos\theta_k^{\,0} \tag{2.1.11}$$

Using (2.1.9), (2.1.10) can be expressed in matrix form by,

$$\Delta\underline{P}_f = [D(\underline{\alpha})]\Delta\underline{\theta} = [D(\underline{\alpha})][\Im]^T \Delta\underline{\delta} \tag{2.1.12}$$

Here the notation $[D(\underline{\alpha})]$ indicates a diagonal matrix whose main diagonal elements are defined by entries of the array $\underline{\alpha}$ in (2.1.11). Now, inserting (2.1.12) into the incremental form of (2.1.4), i.e. $\underline{\Delta P}_e = [\Im]\underline{\Delta P}_f$ one has,

$$\Delta\underline{P}_e = [K]\Delta\underline{\delta} \tag{2.1.13}$$

For a fixed $\underline{P}_d$ one has $\Delta\underline{P}_d = 0$. The matrix $[K]$ in (2.1.13) given by,

$$[K] = [\Im][D(\alpha)][\Im]^T \tag{2.1.14}$$

which is often called the stiffness matrix.

One notices that matrix $[K]$ in (2.1.14) is real and symmetric.

Applying small perturbation technique to (2.1.6) and representing the result in the matrix form, one obtains,

$$[D(\underline{h})]\ \Delta\underline{\dot{\omega}} = \Delta\underline{P}_m - \Delta\underline{P}_e \tag{2.1.15}$$

The entries of $\underline{h}$ in the diagonal matrix $[D(\underline{h})]$ in (2.1.15) are $h_i=2H_i/\omega_s$. In this thesis it is assumed that $\Delta\underline{P_m} = \underline{0}$.

D. System Dynamic Models

Substituting (2.1.13) into (2.1.15) one has,

$$[D(\underline{h})]\Delta\dot{\underline{\omega}} = -[K]\Delta\underline{\delta} \tag{2.1.16}$$

In the absence of controls and governors, the system dynamics is defined by (2.1.16), where the angle-speed relationship is,

$$\Delta\dot{\underline{\delta}} = \Delta\underline{\omega} \tag{2.1.17}$$

(2.1.16) and (2.1.17) can be put into the standard form $\dot{\underline{x}} = [A]\underline{x}$ with $\underline{x}^T = [\Delta\underline{\omega}^T, \Delta\underline{\delta}^T]$ one obtains,

$$\begin{bmatrix} \Delta\dot{\underline{\omega}} \\ \Delta\dot{\underline{\delta}} \end{bmatrix} = \begin{bmatrix} [0] & -[D(\underline{h})]^{-1}[K] \\ [I] & [0] \end{bmatrix} \begin{bmatrix} \Delta\underline{\omega} \\ \Delta\underline{\delta} \end{bmatrix} \tag{2.1.18}$$

Eq. (2.1.18) is well known as the small signal model of a power system. The sub-matrix $[A_{12}]$ is defined as

$$[A_{12}] = -[D(\underline{h})]^{-1}[K] \tag{2.1.19}$$

From (2.1.14), one can show that $[K]$ can be expressed in a form based on the column vectors $\underline{f}_k$, $k = 1,2...N_L$, of the incidence matrix $[\Im]$,

$$[K] = \sum_{k=1}^{N_L} \alpha_k \underline{f}_k \underline{f}_k^T \tag{2.1.20}$$

with

$$\underline{f}_k^T = [0,,\ldots,0,\underset{\uparrow}{1}_{i},0,\ldots,0,\underset{\uparrow}{-1}_{j},0,\ldots,0] \qquad (i,j) \in k \tag{2.1.21}$$

Notice that matrix $\underline{f}_k \underline{f}_k^T$ is a rank-one matrix. In other words, each line k in the power grid contributes a rank one matrix $\alpha_k \underline{f}_k \underline{f}_k^T$ to [K] and [K] is a collection of rank-one matrices. .

2.2 Eigen properties of Matrix [A]

2.2.1 Relationship between [A] and [A_{12}]

The coefficient matrix $[A]$ in (2.1.18) is $2N_B$x$2N_B$ in size and non-symmetric. Therefore, its eigenvalues might have complex conjugate pairs and therefore computationally unattractive. Furthermore, its right and left eigenvectors are not the same. The following derivations show that the eigensystems of non-symmetric [A] could be computed via those of a symmetric matrix. The i^{th} right eigenvector of $[A]$ can be partitioned into speed and angle sub-arrays, so that,

$$\underline{u}_i^T = [\underline{u}_{\omega_i}^T, \underline{u}_{\delta_i}^T] \tag{2.2.1}$$

Similarly, the i^{th} left eigenvector of [A] is defined,

$$\underline{v}_i^T = [\underline{v}_{\omega_i}^T, \underline{v}_{\delta_i}^T] \tag{2.2.2}$$

Inserting $[A]$ from (2.1.18) and $\underline{u}_i$ from (2.2.1) into $[A]\underline{u}_i = \lambda_i \underline{u}_i$, where λ_i is the corresponding i^{th} eigenvalue, one obtains

$$[D(\underline{h})]^{-1}[K]\underline{u}_{\delta_i} = -\lambda_i \underline{u}_{\omega_i} \tag{2.2.3}$$

and

$$\underline{u}_{\omega_i} = \lambda_i \underline{u}_{\delta_i} \tag{2.2.4}$$

Substituting (2.2.4) back into (2.2.3) yields,

$$[D(\underline{h})]^{-1}[K]\underline{u}_{\delta_i} = -\lambda_i^2 \underline{u}_{\delta_i} \tag{2.2.5}$$

Equation (2.2.5) indicates the relationship between the eigenvalues of $[A_{12}]$ and those of [A] through,

$$\mu_i = -\lambda_i^2 \tag{2.2.6}$$

That is, if one has obtained the eigenvalues of $-[A_{12}] = [D(\underline{h})]^{-1}[K]$, then the corresponding eigensystems of $[A]$ are immediately available by (2.2.4), (2.2.5) and (2.2.6). Note that if the eigenvalue of $[D(\underline{h})]^{-1}[K]$ is positive, then the corresponding eigenvalues of $[A]$ would be in complex conjugate pairs because $\lambda_i = \pm j\sqrt{\mu_i}$ .Since $[A_{12}]$ is non-symmetric too, it is computationally unattractive.

2.2.2 Introducing Symmetry by Similar Transformation

Recall from eigenvalue theory, the eigenvalues as well as their algebraic & geometric multiplicity remain unchanged under similar transformation. The next

step is to find a similar transformation of $[D(\underline{h})]^{-1}[K]$ so that the resulting matrix is symmetric and its eigensystems can be solved more easily.

Consider applying a similar matrix $[D(\underline{h})]^{+\frac{1}{2}}$ to $[D(\underline{h})]^{-1}[K]$ and obtain a new matrix $[\psi]$.

$$[\psi] = [D(\underline{h})]^{+\frac{1}{2}}[D(\underline{h})]^{-1}[K][D(\underline{h})]^{-\frac{1}{2}} \tag{2.2.7}$$

This will result in,

$$[\psi] = [D(\underline{h})]^{-\frac{1}{2}}[K][D(\underline{h})]^{-\frac{1}{2}} \tag{2.2.8}$$

Since [K] is symmetric and $[D(\underline{h})]^{-\frac{1}{2}}$ is a diagonal matrix, $[\psi]$ is therefore symmetric. Since $[\psi]$ is obtained from similar transformation it has the eigenvalues of $[D(\underline{h})]^{-1}[K]$. Define $[\underline{\phi}_i, \mu_i]$ as the i^{th} eigenpair of matrix $[\psi]$. One has,

$$[\psi]\underline{\phi}_i = \mu_i \underline{\phi}_i \tag{2.2.9}$$

Substituting (2.2.9) into (2.2.5) one obtains

$$\underline{u}_{\delta_i} = [D(\underline{h})]^{-\frac{1}{2}} \underline{\phi}_i \tag{2.2.10}$$

The following conclusions can be made up to this point,

- The eigenvectors of $[A_{12}]$ can be obtained from those of $[\psi]$.
- The eigenvalues of $[A_{12}]$ are the same as those of $[\psi]$ except with opposite signs.
- From the definition of $[K]$ and $[D(\underline{h})]$, it follows that $[\psi]$ is a symmetric matrix and thus orthogonally diagonalizable and its eigensystem are real.
- Eigenvalues of [A] could be computed via those of $[\psi]$ based on (2.2.6).

Similarly, the left eigenvectors of $[A]$ satisfy

$$\underline{v}_i^T [A] = \lambda_i \underline{v}_i^T \tag{2.2.11}$$

With $\underline{v}_i^T$ defined in (2.2.2), following the same steps as above, it can be shown that

$$\underline{v}_{\delta_i}^T = \lambda_i \underline{v}_{w_i}^T \tag{2.2.12}$$

and

$$\underline{v}_{w_i}^T [A_{12}] = \lambda_i \underline{v}_{\delta_i}^T \tag{2.2.13}$$

Therefore substituting (2.2.12) into (2.2.13), one has

$$\underline{v}_{w_i}^T [D(\underline{h})]^{-1} [K] = -\lambda_i^2 \underline{v}_{\delta_i}^T \tag{2.2.14}$$

Define the same eigenpair of $[\psi]$ in (2.2.3), it can be shown that

$$\underline{v}_{w_i} = [D(\underline{h})]^{+\frac{1}{2}} \underline{\phi}_i \tag{2.2.15}$$

In summary, the eigensystem of $[A]$ are obtained by finding the eigensystem of $[\psi]$, a real, symmetric, N_BxN_B matrix.

Chapter 3 Theoretical Foundation

The B & B method [10] uses the results of the original work by Bunch and Nielsen, and Sorensen [11] to update the eigensystem of a symmetric matrix that is subjected to ROMs. This chapter will present the derivation of the B & B method. The contribution of the thesis consists of rendering the mathematical models in forms suitable for implementation by parallel processing. Discussions regarding ROMs of symmetric matrices are also found in [2, 16, 17 and 18].

3.1 Relationship of Matrices [K] and $[\psi]$

Recall from (2.1.20), $[K]$ is defined as $[K] = \sum_{k=1}^{N_L} \alpha_k \underline{f}_k \underline{f}_k^T$. Substituting into (2.2.8), one has

$$[\psi] = \sum_{k=1}^{N_L} \alpha_k \underline{\xi}_k \underline{\xi}_k^T \tag{3.1.1}$$

where $\underline{\xi}_k = [D(h)]^{-\frac{1}{2}} \underline{f}_k$ and f_k is defined in (2.1.21). Thus $[\psi]$ is also a collection of rank-one matrices.

When a transmission line, indexed by subscript ℓ, is switched into the power grid or alternatively when it is already the power grid but it is switched out, the $[K]$ matrix is modified by $\alpha_l \underline{f}_l \underline{f}_l^T$ which is an ROM of $[K]$. Since this modification affects $\alpha_l \underline{\xi}_l \underline{\xi}_l^T$ in (3.1.1) in the same way, it is also an ROM of $[\psi]$.

3.2 Connection of One Transmission Line

Before adding line ℓ, the matrix $[\psi]$ is denoted by $[\psi_0]$ and it is assumed that all the eigenvalues and eigenvectors of $[\psi_0]$ have been solved, i.e. eigenpairs $(\underline{\mu}, [\Phi])$. The vector of eigenvalues $\underline{\mu}$ can be put in a diagonal

matrix $[D(\underline{\mu})]$ and eigenvectors matrix $[\Phi]$ is partitioned into columns vectors. Assume all the μ_i 's are distinct. $[\psi_0]$ has the following property:

$$[\psi_0] = [\Phi][D(\underline{\mu})]][\Phi]^T \tag{3.2.1}$$

Note that $[\Phi]$, whose columns are the eigenvectors of $[\psi_0]$, is an orthogonal matrix, i.e. $[\Phi]^T[\Phi] = [\Phi][\Phi]^T = [I]$, where $[I]$ is the identity matrix.

3.2.1 Eigenvalues Computation

After line ℓ is added, the eigenvalues of matrix $[\psi]$ change from $[D(\underline{\mu})]$ to $[D(\sigma)]$ and the eigenvectors matrix changes from $[\Phi]$ to $[W]$. For any new eigenvalue σ with corresponding eigenvector $\underline{w}$, the following equation must be satisfied by the connection of line ℓ, which is embodied by α_l and $\underline{\xi}_l$ in $[\psi]$.

$$\left\{ [\psi_0] + \alpha_\ell \underline{\xi}_\ell \underline{\xi}_\ell^T \right\} \underline{w} = \sigma \underline{w} \tag{3.2.2}$$

The above equation can be put in the form,

$$\left\{ [\psi_0] - \sigma[I] \right\} \underline{w} = -\alpha_\ell (\underline{\xi}_\ell^T \underline{w}) \underline{\xi}_\ell \tag{3.2.3}$$

Solving for $\underline{w}$ from the left-hand-side of (3.2.3) leads to,

$$\underline{w} = -\alpha_\ell (\underline{\xi}_\ell^T \underline{w}) \left\{ [\psi_0] - \sigma[I] \right\}^{-1} \underline{\xi}_\ell \tag{3.2.4}$$

Forming $\underline{\xi}_\ell^T \underline{w}$ based on (3.2.4), one has

$$\underline{\xi}_\ell^T \underline{w} = -\alpha_\ell (\underline{\xi}_\ell^T \underline{w}) \underline{\xi}_\ell^T \left\{ [\psi_0] - \sigma[I] \right\}^{-1} \underline{\xi}_\ell \tag{3.2.5}$$

Substitute (3.2.4) and (3.2.5) back into (3.2.3), and dividing throughout by $\underline{\xi}_\ell^T \underline{w}$ (assuming $\underline{\xi}_\ell^T \underline{w} \neq 0$), one has

$$1 = -\alpha_\ell \underline{\xi}_\ell^T \left\{ [\psi_0] - \sigma[I] \right\}^{-1} \underline{\xi}_\ell \tag{3.2.6}$$

Using the fact that $[\psi_0] = [\Phi][D(\underline{\mu})][\Phi]^T$ and $\sigma[I] = [\Phi]\sigma[I][\Phi]^T$, (3.2.6) becomes

$$1 = -\alpha_\ell \underline{\xi}_\ell^T \left\{ [\Phi]([D(\underline{\mu})] - \sigma[I])[\Phi]^T \right\}^{-1} \underline{\xi}_\ell \tag{3.2.7}$$

Notice that $[\Phi]^{-1} = [\Phi]^T$, after some linear algebraic operations, (3.2.7) becomes

$$1+\alpha_\ell \sum_{k=1}^{N} \frac{(\underline{\phi}_k^T \underline{\xi}_\ell)^2}{\mu_k - \sigma} = 0 \tag{3.2.8}$$

This is because $\underline{\phi}_k$'s in (3.2.8) are the eigenvectors of $[\psi_0]$ which are orthogonal to each other and can be expressed as,

$$[\Phi] = [\underline{\phi}_1, \underline{\phi}_2, \underline{\phi}_3 \ldots\ldots \underline{\phi}_N] \tag{3.2.9}$$

The new eigenvalues of $[\psi]$ after the connection of line l are solved from (3.2.8). Note that (3.2.8) can also be derived from characteristic polynomial equations [17] of the coefficient matrix in (3.2.2).

3.2.2 Eigenvectors Computation

3.2.2.1 Eigenvectors for Non-zero Eigenvalues

Since eigenvectors remain valid with scalar factors, it is convenient to leave out the factor $-\alpha_\ell \underline{\xi}_\ell^T \underline{w}$ in the definition of $\underline{w}$ in (3.2.4). Then (3.2.4) becomes, based on $[[\Phi][\Phi]^T = I$ and $[\Phi]^{-1} = [\Phi]^T$,

$$\underline{w} = [\Phi]\{[D(\mu)] - \sigma[I]\}^{-1}[\Phi]^T \underline{\xi}_\ell \tag{3.2.10}$$

Recall that $\underline{\xi}_k = [D(h)]^{-\frac{1}{2}} \underline{f}_k$, with some linear algebraic operations based on (3.2.9), the corresponding eigenvector $\underline{w}_m$ of eigenvalue σ_m is,

$$\underline{w}_m = \sum_{k=1}^{N} \frac{\underline{\phi}_k^T \underline{\xi}_\ell}{\mu_k - \sigma_m} \underline{\phi}_k \tag{3.2.11}$$

Using the fact from (3.1.1), one has,

$$\underline{\phi}_k^T \underline{\xi}_\ell = (\phi_{ki} h_i^{\frac{1}{2}} - \phi_{kj} h_j^{\frac{1}{2}}) \tag{3.2.12}$$

Note that $\underline{\phi}_k^T \underline{\xi}_\ell \neq 0$ .Otherwise $\mu_k = \sigma_k$ [11] which contradicts with above assumption that all μ_i are distinct. This can also be seen from section 3.5, mathematical properties of (3.2.8). One can show that the assumption $\underline{\xi}_\ell^T \underline{w}_m \neq 0$ holds for all $\underline{w}_m$ except for $\underline{w}_1$ which corresponds to $\sigma_1 = 0.0$. Actually $\underline{\xi}_\ell^T \underline{w}_m = \frac{-1}{\alpha_l}$ holds based on (3.2.8).

3.2.2.2 Eigenvectors for Zero Eigenvalue

For $\sigma_1 = 0.0$, $\underline{w}_1$ cannot be obtained directly from (3.2.11). Recall from (2.1.4), i.e. $[\Im]^T \underline{\ell} = \underline{0}$, thus $[\Im]$ is singular. Therefore $[K]$ and $[\psi]$ are singular too from (2.1.13) and (2.2.8) respectively. (2.2.5) indicates that $\underline{u}_{\delta_1} = \underline{\ell}$ which corresponds to $\lambda_1 = 0$. Recall from (2.2.10), i.e. $\underline{u}_{\delta_i} = [D(\underline{h})]^{-\frac{1}{2}} \underline{\phi}_i$, one must have

$$\underline{w}_1 = [D(\underline{h})]^{+\frac{1}{2}} \underline{\ell} \tag{3.2.13}$$

3.3 Joining Two Sub-networks

Originally, there are two sub-networks. Sub-network 1 is characterized by its N_1xN_1 matrix $[\psi_1]$ and eigensystem $(\underline{\mu}_1, [\underline{\Phi}_1])$, while the sub-network 2 is characterized by its N_2xN_2 matrix $[\psi_2]$ and $(\underline{\mu}_2, [\underline{\Phi}_2])$. Here, one needs to solve the eigensystem of a system created by connecting two sub-networks through a single tie line. This is done by considering the two sub-networks as one system while they are disjoint. Joining them by a tie line is the same as adding a single line to the entire system. Thus, relations (3.2.8) and (3.2.11) become applicable and no new methodology is needed.

Viewing the two sub-networks as one requires knowledge of the disjoint system $[\psi_0]$ and its eigensystem $(\underline{\mu}, [\Phi])$. Based on (2.2.8), one has

$$[\psi_0] = \begin{bmatrix} [D(\underline{h}_1)]^{-\frac{1}{2}} & 0 \\ 0 & [D(\underline{h}_2)]^{-\frac{1}{2}} \end{bmatrix} \begin{bmatrix} [K_1] & 0 \\ 0 & [K_2] \end{bmatrix} \begin{bmatrix} [D(\underline{h}_1)]^{-\frac{1}{2}} & 0 \\ 0 & [D(\underline{h}_2)]^{-\frac{1}{2}} \end{bmatrix} = \begin{bmatrix} [\psi_1] & 0 \\ 0 & [\psi_2] \end{bmatrix} \tag{3.3.1}$$

The definition of $[\psi_0]$ in (3.3.1) infers that its eigensystem $(\underline{\mu}, [\Phi])$ can be expressed as,

$$\underline{\mu}^T = [\underline{\mu}_1^T, \underline{\mu}_2^T]$$
$$[\Phi] = \begin{bmatrix} [\Phi_1] & [0] \\ [0] & [\Phi_2] \end{bmatrix} \tag{3.3.2}$$

By simple inspection, one can show that this definition of $(\underline{\mu}, [\Phi])$ in (3.3.2) meets all conditions that the eigensystem of $[\psi_0]$ will satisfy. In doing so, (3.3.2)

can be more easily implemented in a parallel processing environment than considering two disjoint sub-networks separately.

3.4 Disconnection of One Line

Disconnection is similar to connection except the following points must be respected:

- Replace α_ℓ by $-\alpha_\ell$ in (3.2.8) and (3.2.11) respectively. This is because removing one line from the entire system is equivalent to adding a new line with negative α_ℓ between these two nodes.
- After line l is removed, the two separate sub-networks connected by line ℓ must still remain connected.

3.5 Properties of the Secular Equation (3.2.8)

Define

$$f(\sigma) = 1 + \alpha_\ell \sum_{k=1}^{N} \frac{(\underline{\phi}_k^T \underline{\xi}_\ell)^2}{\mu_k - \sigma} \tag{3.5.1}$$

(3.5.1) is a secular equation, according to [16].

One immediately notices that $f(\sigma)$ has N-1 roots and they are the corresponding non-zero eigenvalues of the new matrix after ROM of $[\psi_0]$. $f(\sigma)$ of (3.5.1) has the following properties based on [16]. The simple proofs are given below based on the hints in [16]:

1) f is a monotonic function in each sub-interval bounded by two consecutive numbers (μ_i, μ_{i+1}) arranged in ascending order in magnitudes. If $\alpha_\ell > 0$, f is an increasing function and if $\alpha_\ell < 0$, f is a decreasing function.

Proof: Taking the derivative of (3.5.1) with respect of σ, one has

$$\dot{f} = -\alpha_\ell \sum_{k=1}^{N} \frac{(\underline{\phi}_k^T \underline{\xi}_\ell)^2}{(\mu_k - \sigma)^2} \tag{3.5.2}$$

Since the sign of $\dot{f}$ remains unchanged in each sub-interval, then f must be a monotonic function. Observe that $f = -\infty$ at μ_i and $f = +\infty$ at μ_{i+1} since $\alpha_l > 0$,

one must have the above argument. Similar argument holds for $\alpha_l < 0$, in this case, f is a decreasing function.

2) f has only one solution in the interval (μ_i, μ_{i+1}) where the μ_i's have been placed in ascending order of magnitude.

Proof: It is obvious following from 1).

3) If $\alpha_\ell > 0$, $\mu_i < \sigma_i < \mu_{i+1}$

Proof: It is obvious following from 1) and 2).

4) Define $a = \left\| \underline{\xi}_\ell \right\|_2$, if $\alpha_\ell > 0$ then $\sigma_n < \mu_n + a^2 \alpha_\ell$ and $\left\| \underline{\xi}_\ell \right\|_2$ can be solved based on non-zero entries corresponding the ones in $[D(\underline{h})]$.

Proof: From $[\psi_0] = [\Phi][D(\underline{\mu})][\Phi]^T$ and $[\psi] = [\psi_0] + \alpha_\ell \xi_\ell \xi_\ell^T$, one has $[\psi] = [\Phi]([D(\mu) + \alpha_\ell b b^T][\Phi]^T$, where $b = [\Phi]^T \xi_\ell$. Recall that a vector preserves its length under orthogonal transformation, i.e. $\|b\|_2 = \left\| [\Phi]^T \xi_\ell \right\|_2 = a$.

Then one has

$$\left\| \xi_\ell \right\|_2^2 = a^2 = \frac{1}{h_i} + \frac{1}{h_j} \tag{3.5.3}$$

And the largest eigenvalue of $[\psi]$ is the largest eigenvalue of $D(\mu) + \alpha_\ell b b^T$ since Q is orthogonal. Recall that the trace of a matrix is equal to the sum of its eigenvalues, one has

$$\sum_{i=1}^{n} (\mu_i + \alpha_l b_i^2) = (\sum_{i}^{n} \mu_i) + \alpha_\ell \|a\|_2^2 = \sum_{i=1}^{n} \sigma_i \tag{3.5.4}$$

Using the fact that $\mu_i < \sigma_i$ from 3), one must have

$$\sigma_n < \mu_n + a^2 \alpha_\ell \tag{3.5.5}$$

5) If $\alpha_\ell < 0$, $\mu_{i-1} < \sigma_i < \mu_i$

6) If $\alpha_\ell < 0$ then $\mu_1 + a^2 \alpha_\ell < \sigma_1$

Proofs of 5) and 6) can be done by following the similar steps above. The details are not given here.

Fig. 3.1 shows the plot of function f for some selected sub-intervals with $\alpha_\ell > 0$. The sample parameters come from the case of the 4062-system in Chapter 5 which will be discussed later.

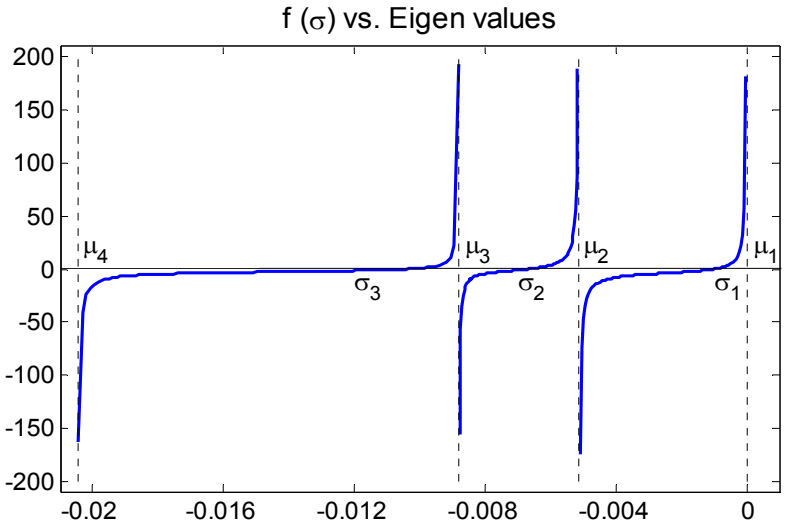

Fig. 3.1 Function f (σ) of (3.5.1), μ_k arranged in ascending order of magnitude

3.6 Numerical Analysis

3.6.1 Eigenvalue Computation

From the mathematical properties of secular equation (3.5.1) and based on inspection of Fig.3.1, one concludes that computing the new eigenvalues σ's is equivalent to finding the roots of function f in each sub-interval. During the research, the author tested the following numerical solvers in finding the roots of f: Newton-Raphson method, Secant method (Newton-based), the Modified secant method as well as Bisection Method [19]. The experience is that Newton-based methods are divergent sometimes due to the plot shape and that Newton-Raphason methods make use of the gradients of f. It is obvious from Fig.3.1 that the iterative estimates fluctuate widely if Newton-methods are used. Besides reliable convergence, very high accuracy of the root is also desired since the B & B method only addresses the case that a single line change in the system. As a power system continually have lines connected & disconnected and have load changes which affect α_ℓ, the continual updating of the eigensystems requires repeated solutions from $f(\sigma)$. Each time, the B & B method uses the results from

the previous stages computation. Therefore the errors accumulate for a series of ROMs. Assessing cumulative errors is beyond the scope of the thesis. However a simple error analysis is addressed in Chapter 5 for the purpose of interpretation of the results.

3.6.1.1 Bisection Method

Because of accuracy concerns, the bisection method [19] was employed to find the roots of $f(\sigma)$. For absolute error set at 10^{-16}, it costs around 50 iterations to find a root in each sub-interval. The bisection method requires knowing the upper and the lower limits. This does not present any difficulty because, as Fig. 3.1 shows, each subinterval is bounded by μ_i and μ_{i+1}, which are the original eigenvalues of $[\psi_0]$. However, the following problems need to be addressed first from the point of view of programming.

- At endpoints of each subinterval the values of function f are $\pm\infty$. In the proposed algorithms, this problem has been solved by adding and subtracting a tiny amount (e.g.10^{-14}) from μ_i and μ_{i+1} respectively. If eigenvalues are poorly separated, e.g. $|\mu_i - \mu_{i+1}|<10^{-9}$, one can assume that the new eigenvalue is equal to $\frac{\mu_{i+1}+\mu_i}{2}$.
- For $\alpha_\ell > 0$, upper limit for the largest eigenvalue is not known This problem could be dealt in two different ways:
  1) From the property of f described above in 4) of section 3.5, the largest eigenvalue is also bounded by $\sigma_n < \mu_n + a^2\alpha_\ell$ .This upper bound can be used in the bisection method too.
  2) Based on Theorem 1 in [11] and recall that the eigenvalues remain unchanged after orthogonal transformation, one has the following results:

$$\sigma_n = -\sum_{k=1}^{n-1}\sigma_k + \sum_{k=1}^{n}\mu_k + a^2\alpha_\ell \qquad (3.6.1)$$

  where a is defined in (3.5.3).

  If all the generators inertias are equal to one, then (3.6.1) simplifies to

$$\sigma_n = -\sum_{k=1}^{n-1} \sigma_k + \sum_{k=1}^{n} \mu_k + 2\alpha_\ell \tag{3.6.2}$$

3.6.1.2 Eigenvalues Sorting

From Fig. 3.1, one sees that the eigenvalues must be sorted and placed in ascending order so that in each sub-interval there is only one root. This is also due to the fact that the eigenvalues array $\underline{\mu}^T = [\underline{\mu}_1^T, \underline{\mu}_2^T]$ of $[\psi_0]$ are in general not in ascending order. Therefore sorting is quite necessary. In the practice of programming, the author experimented both with sequential and parallel quick sort. However, parallel quick sort was intended for tens of millions of data. For this specific problem, a sequential quick sort can do a good job within the order of milliseconds, negligible time when comparing to the total computation time. Below is a brief review of quick sort, both sequential and parallel.

1. Quick sort [C. A. R. Hoare] is recognized as the fastest sequential sorting algorithm compared to bubble sort and other sorting algorithms. It is a typical divide-and-conquer (DC) method implementation which is to partition the complete tasks into relatively small ones and to solve recursively. The basic idea of quick sort is to pick a pivot element randomly in each iteration and partition the entire array into two sub-arrays with the pivot element in its right place. In other words, if ascending order of the array elements is desired, the left sub-array elements are all less than or equal to the pivot element and the ones in right sub-array are greater or equal to the pivot element. The algorithm is recursive until all the elements are sorted in their right positions. The total cost, averagely, is $O(n\log(n))$ and the worst case is $O(n^2)$ [20] when the array is already sorted either in an ascending or in a descending order. The attached quicksort function is based on the algorithm in [20]. The performance of this algorithm is really depending on pivot selection strategy and the input sequence. In the practice of programming, 3 pivot selection strategies have been attempted for the purpose of study, i.e. selecting the 1st element, the last element and random element in the sorting sequences as

pivot elements. Although sorting time is negligible for all the three choices, it is a good experience to learn.

The results are shown in Table 3-1.

Table 3-1: Sorting time comparison of different pivot selection strategies

Sequence size	First element	Last element	Random element
4062	0.000773	0.004281	0.000292
4419	0.000466	0.002769	0.000302

Note: The values in Table 3-1 are in seconds.

One concludes the followings,

- Strategy of choosing the pivot element randomly has the best performance for this specific problem. Choosing 1st element as pivot has similar performance.
- Strategy of choosing the last element as pivot has the worst performance for this specific problem.
- Both sequence size or data distribution and pivot selection strategies will determine the overall performance.

2. Parallel quick sort [20, 21 and 22] usually deals with large scale data. Based on the author's knowledge, the idea behind this is to partition the total data into p sub-blocks if n processors are available for sorting and each processor will do quick sort individually. Note that the following must be successfully overcome in order to have good performance with $p << n$:
   - p-1 pivot elements must be in their right positions after partitioning.
   - Partition the communicator, i.e. the collection of the processors into p groups.
   - In order for each group has almost balanced workload, the sorting tasks burden must be proportional to the number of processors in this group.
   - In the case that n is not a multiple of p, the remainder processors must be treated separately.

The readers are encouraged to implement parallel quicksort and to compare with the sequential one to see if some improvement can be done for a sequence of 5,000 floating point numbers.

Furthermore, in the practice of developing the algorithms, the eigenvalues were assigned in ascending order of magnitude to a new array after sorting, but tags were kept of their original subscript numbering. The cross referencing of subscript indexes enabled the corresponding eigenvector $\underline{\phi}_k$ to be located from [Φ]. Thus it was not necessary to reorder the columns of eigenvectors in [Φ], a not insignificant time-saver.

3.6.1.3 Multiple Zero Eigenvalues

Each original sub-network has one zero eigenvalue. In connecting two sub-networks, the resulting system has only one zero eigenvalue. There is no real problem here because one eigenvalue takes the usual zero value (i.e. $\sigma_1 = 0.0$) and has the normalized eigenvector consisting of identical values based on (3.2.13). The second eigenvalue is obtained by bisecting between μ_1=0.0 and μ_2 etc., the eigenvalues of $[\psi_0]$ which have been obtained from the sorted eigenvalues array.

3.6.1.4 Multiple Eigenvalues

Although the B & B method makes the assumption that all the eigenvalues are distinct before and after update, (3.2.8) is also applicable to repeated eigenvalues. The updated eigenvalues remain unchanged with algebraic multiplicity reduced by one. However, (3.2.11) cannot be applied directly to compute the eigenvectors. A new method needs to be developed in this case, e.g. deflation method [11]. Dealing with repeated eigenvalues is out of the scope of this thesis. Actually, there is little possibility of meeting multiple eigenvalues in power engineering field.

With all the above problems have been solved, bisection method can be implemented with no trouble. Note that computation of each eigenvalue is independent of each other, thus parallel computation can be implemented. This is also called data parallel which will be introduced later. The cost for computing all the eigenvalues by a single-processor machine is O (N^2).

3.6.2 Eigenvector Computation

Based on (3.2.11), one observes that computing eigenvectors only involves vector sums as well as vectors multiplied by scalars and they can be computed as long as the corresponding eigenvalues are solved. Again each eigenvector computing is independent and therefore it could be effectively implemented in a parallel processing environment. Still the updated eigenvalues are required to differ from all the original eigenvalues otherwise the sum in (3.2.11) will be dominated by infinity terms.

The cost for computing each eigenvector is O (N^2). Therefore the total cost for updating the entire eigenvectors matrix is O (N^3). The cost here is referred to sequential programming implementation.

Note that eigenvectors computation is the most costly part. This could be implemented with standard double FOR loops in #C for a single eigenvector computation. In the practice during the research, the concept of unroll looping [23] was employed and the computation speed was increased by a factor of 3 ~ 4. The author appreciates Mr. Francois Guertin from RQCHP for this improvement and helpful discussions. The main idea behind unroll looping is to increase the size (number of operations) of the loop body while reducing the number of "administration" instructions in the loop, e.g. conditional branches for indexing. It also increases the usage of cache during the computations. The unroll factor was set manually since the compiler is not smart enough to do so.

Note that converting eigenvectors of $[\psi']$ back to those of $[A]$ requires O (N^2) operations which are also negligible compared to the total cost.

Chapter 4 MPI Parallel Algorithms Design

4.1 Parallel Computing Overview

Parallel computing has experienced extraordinary development during the past decade as MPI standards developed and information technologies advanced. Due to its simplicity, efficiency and wide applications, currently many researchers are working in this field to deal with large scale problems which are usually impossible on traditional PCs. Basically, parallel computing is implemented on computer clusters or super-computers based on standardized protocols, e.g. MPI parallel programming in #C. Roughly speaking, parallel computing differs from sequential implementation on single-processor machines in the following ways as shown in Fig.4.1 and Fig.4.2.

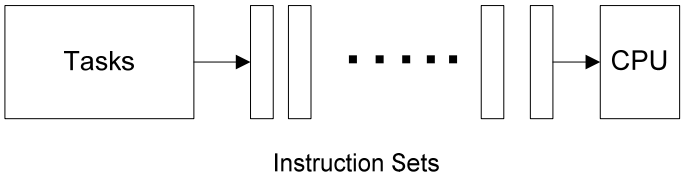

Fig. 4.1 Sequential computing illustration

As shown in Fig.4.1, tasks are implemented on a single CPU and the instructions are executed one after another.

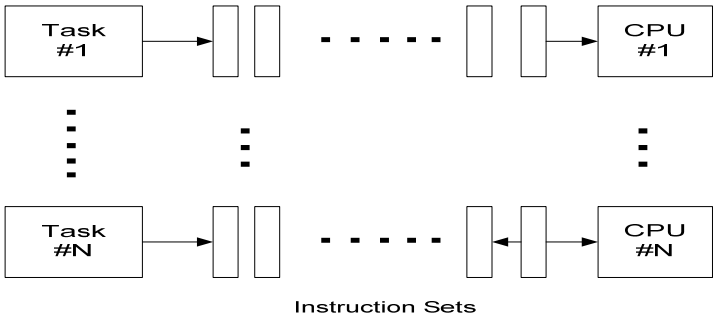

Fig. 4.2 Parallel computing illustration

Fig. 4.2 indicates that the total tasks are sub-divided into relatively small ones and each one is executed on different CPUs simultaneously. This is essentially the basic ideas of parallel computing. Note that, in Fig.4.2, instructions sets for each CPU could be either the same or different depends on real implementations. In the developed programs, different processors execute the same codes while working on different set of data.

4.2 Parallel Models

Several parallel algorithm models are available depending on the machine structures and protocols used, e.g. data-parallel model, task graph model, work pool model [20] etc. In general, parallel computing falls into two main categories: data parallel and task parallel.

4.2.1 Data parallelism

Data parallelism is one of the simplest parallel computing models [20]. Data parallelism enables all the processors to work on different sets of data with the same segments of code. For example, to compute the product of $[A]\underline{x}$, where $[A] \in R^{nxn}$ and $\underline{x} \in R^n$ .There are a few ways to implementing this based on MPI parallel programming, only one way is shown below. Although it is not implemented in the final developed algorithm, the design concepts are quite essential for parallel implementations. The MPI routines introduced below will be discussed shortly.

Suppose A has a dimension of multiple of 2 and two processors are to be employed to compute $A\underline{x} = b$, i.e. P_0 and P_1, the procedures are:

- Partition matrix [A] into two equal sub-matrices as shown in Fig.4.3, where $A1, A2 \in R^{\frac{n}{2}xn}$ and n=4.
- Using MPI_Scatter to send $[A_1]$ to P_0 and $[A_2]$ to P_1 respectively.
- Using MPI_Bcast to broadcast data $\underline{x}$, therefore $\underline{x}$ is known to both P_0 and P_1.
- Start computation, i.e. P_0 computes $A_1\underline{x}$ and P_1 computes $A_2\underline{x}$.
- Using MPI_Gather to collect 2 resulting sub-vectors $b_1, b_2 \in R^{\frac{n}{2}}$ to form the results $b \in R^n$

$$\begin{matrix} \\ A_1 \\ A_2 \end{matrix}\left[\begin{array}{cccc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right]\left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array}\right]$$

Fig. 4.3 Matrix partition illustration

In summary, each processor will execute the same segment of codes with different data sets of smaller size.

Cost analysis: Without considering the communication time, the total cost is $2n * \frac{n}{2} = n^2$ for parallel implementation shown in Fig.4.3, i.e. it is half of the original cost $2n * n = 2n^2$. Ideally if n processors are involved, the cost will be $2n$ flops (floating-point operation) without taking into account the communication time among processors. The total computation cost reduces greatly via parallel computing. However, in practice, if n is very large, it is not efficient in doing this way which can be seen shortly. In the following algorithms description in section 4.4, one can see that data parallel is quite efficient for this specific problem.

4.2.2 Task Parallelism

Task parallelism allows several independent subroutines to work simultaneously. For example, in this specific problem, if one wants to update eigensystems for three sub-networks, i.e. connect two sub-networks which are islanded and adding a line in a 3rd sub-network, and the connection and adding a line require to be done concurrently, these two tasks could be implemented in the following way:

- Develop two function routines, one is for connection update and the other is for adding update. Note these two function calls are very similar for this specific problem, this is just an illustrative example for task parallelism implementation.
- Partition the collection of processors involved, which is also referred to a communicator, into two sets depending on the workload for each task. This could be done by MPI_Comm_split. Usually the user will specify how many

processors in total will be used in the implementation before the program is executed. The sample MPI in #C pseudo codes for communicator partitioning are in Appendix I.

- Using MPI_Gather to collect all the results when both the two tasks are completed and free the communicator groups.

Some concepts or ideas might still be vague up to this point. They become clearer after going through the following sections.

4.3 Introduction to MPI Routines

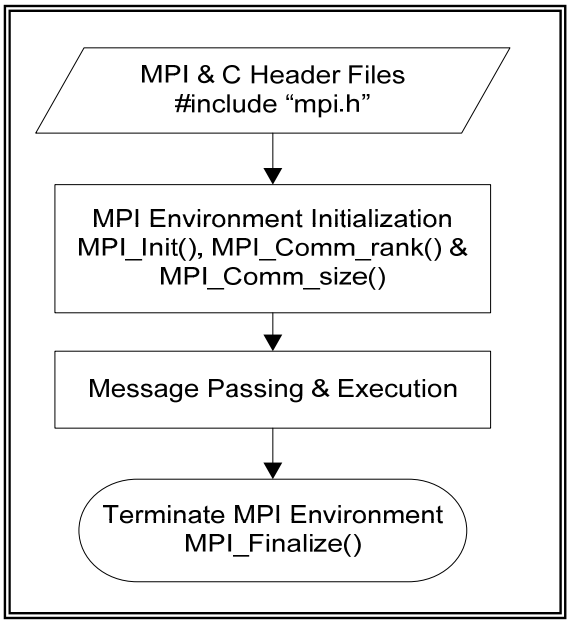

Fig.4.4 General software structure of MPI in #C

MPI is a collection of subroutines or libraries suitable for parallel implementation on super-computer or computer clusters. They recognize the multi-processor machine structures by message passing. Up to now, two standards are available, MPI-I and MPI-II. MPI-II has more advantages, e.g. it allows parallel I/O, while in MPI-I, only the root processor, i.e. P_0, can read data input. MPI could be implemented in #C, C++ as well as in FORTRAN. Fig.4.4 shows a general software structure of MPI parallel programs. The readers will have a clearer picture while studying the algorithms design in section 4.4. The algorithms illustrated in this section are written with MPI-I in #C. The readers are encouraged to implement with MPI-II to see if performance can be improved for

this specific problem. Before going into details for MPI routines, one needs to know a few important and basic MPI concepts.

- **Rank:** Each processor involved in the execution has a unique identification number which refers to its rank, it is an integer. For instance, if 10 processors are used in the computation, the processors ranks range from 0 to 9. Processor with rank 0 is called the root processor.
- **Communicator:** Communicator is a collection of processors which are involved in the computation. The default communicator is called *MPI_COMM_WORLD*. Communicators could have different names depends on real implementations or more than one communicator can be defined.
- **Speedup:** Amdahl's law[24] states that parallel program speedup is defined as follows,

$$speedup = \frac{1}{\frac{p}{n} + (1 - p)} \tag{4.1}$$

where p – parallel portion, n – number of processors.

For example, if p=0, speedup=1. Namely, there will be no speedup no matter how many processors are involved.

If p=0.5, ideally maximum speedup=2 when an infinite number of processors are used. However, in reality it is not possible. The real speedup will approach to 2 as the number of processors increases without considering the communication time among the processors. This concept will be discussed more in next chapter.

There are more than 100 routines in MPI-I. Here only several routines are introduced which are quite related to the designed algorithms. The description of all these routines could be found [14 and 15]. This thesis will focus on detailed implementation while describing the MPI codes.

4.3.1 Basic MPI Routines

The following 4 MPI routines are mandatory for all MPI programs.

- MPI_Init (int *argc,char ***argv) – Initializes MPI environment
- MPI_Finalize(void) – Terminates MPI program
- MPI_Comm_rank(MPI_Group group,int *rank) – Returns rank for each processor

- MPI_Comm_size (MPI_Group group,int *size) – Returns the number of processors in the group

4.3.2 MPI Routines Employed in the Algorithms

Before going into details for the algorithms, some MPI routines are briefly introduced here. Basically in MPI, the communication mechanism among processors is classified into two types: point-to-point communication, e.g. MPI_Send and MPI_Recv, and collective communication, e.g. MPI_Bcast. The following MPI routines are used intensively in the developed MPI codes. The details for parameters passing will be clearer when one studies real implementation as shown in the sample codes.

4.3.2.1 Point-to-point Communication

Point-to-point communication occurs between a pair of processors, that is, one processor sends message and the other processor receives, e.g. MPI_Send and MPI_Recv.

- MPI_Send: Send data from source processor to the destination processor.
- MPI_Recv: Receive data from source processor.

The APIs are shown below.

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);

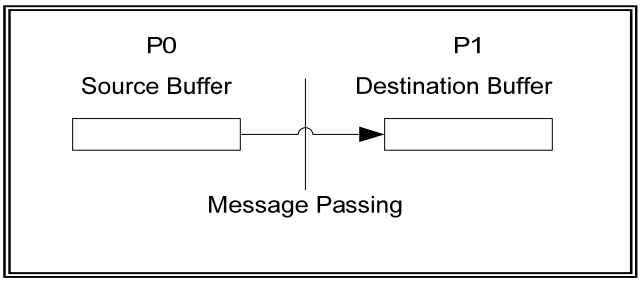

Fig. 4.5 MPI_Send and MPI_Recv illustration

As illustrated in Fig. 4.5, data is sent from source buffer of process P_0 to destination buffer of processor P_1. This pair of send and receive are also called blocking send and receive, that is, MPI_Send does not return until the sent message has safely arrived in the destination buffer. Although these two routines

are dropped in later codes development, these two routines are very basic ones for message passing among processors.

Note that MPI_Send and MPI_Recv must appear in pairs. For example, in the original code development, the following code segments in Fig. 4.6 have been employed to inform each processor that how many tasks it is supposed to do, i.e. tasks portioning. Here suppose n processors are employed for the computation.

```
if (rank = = 0) {
        for (i = 0; i<n; i++) {
            recvcnt=sendcnt[i];
            MPI_Send(&recvcnt,1,MPI_INT,i,0,MPI_COMM_WORLD);
        }
}
else
MPI_Recv(&recvcnt,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
```

Fig. 4.6 Sample codes for MPI_Send and MPI_Recv

Note that *sendcnt is predefined by a piece of codes depending on number of processors employed and total number of tasks. One can see that from Fig. 4.6, root processor P_0 sends each value of *sendcnt, i.e. recvcnt, to all the processors including itself. And all the other processors receive this message, i.e. recvcnt. The value for recvcnt could be either the same or different for each processor. This idea was dropped for later development since the codes segment above could be realized by only one MPI routine, i.e. MPI_Scatter, which will be clearer shortly. However, design concepts behind these two routines are essential for message passing.

4.3.2.2 Collective Communication

Roughly speaking, collective communication requires all the processors within the communicator or group be involved for message passing. The following collective communication MPI routines are used in the algorithms: MPI_Bcast, MPI_Scatterv, MPI_Scatter, MPI_Allreduce and MPI_Gatherv.

- MPI_Bcast:

As shown in Fig.4.7, data 'A' from root processor, i.e. P_0, is broadcast to processors P_1, P_2, and P_3 as illustrated in Fig.4.7. After this function call, processors P_0, P_1, P_2, and P_3 will have the data 'A'. API is shown below.

MPI_Bcast(void *buffer,int count,MPI_Datatype datatype,int root, MPI_Comm comm);

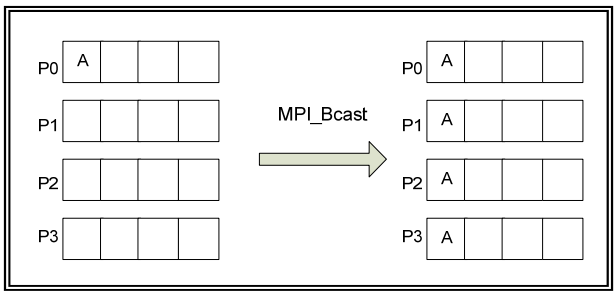

Fig.4.7 MPI_Bcast illustration

An example is shown in Fig.4.8.

```
MPI_Bcast (&N1, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Fig.4.8 Sample code for MPI_Bcast

N1 of data type MPI_INT is broadcast from P_0 to all the processors within the communicator MPI_COMM_WORLD.

- MPI_Scatter :

API for MPI_Scatter is shown below.

int MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);

As illustrated in Fig. 4.9, data ABCDEFGH from root process P_0 are scattered so that processors P_0, P_1, P_2, and P_3 will each receive AB, CD, EF, GH respectively and the data received by each processor have equal size. Actually MPI_Scatter could be considered as series of MPI_Send to each processor from the root processor P_0 as shown in the above example. Obviously the latter is less efficient for most of the cases. Take the above example for MPI_Send and MPI_Recv, it could be implemented using MPI_Scatter shown in Fig.4.10. Here MPI_Scatter is used instead of MPI_Scatterv due to that recvcnt is just an integer, i.e. the amount of data that each processor will receive is equal to ONE.

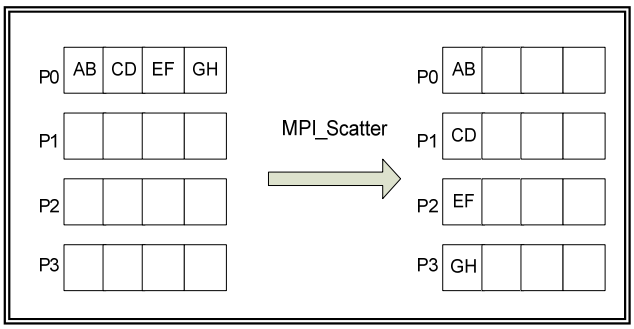

Fig.4.9 MPI_Scatter illustration

```
MPI_Scatter (sendcnt,1,MPI_INT,&recvcnt,1,MPI_INT,0,MPI_COMM_WORLD);
```

Fig.4.10 Sample code for MPI_Scatter

Fig. 4.10 indicates that data in buffer *sendcnt from P_0 are uniformly scattered to each processor in the communicator MPI_COMM_WORLD. And this code segment performs the same function as that in Fig. 4.6. In other words, *sendcnt is an integer array for this case. After this function is called, each processor will receive ONE integer, i.e. recvcnt. The value of each recvcnt could be either the same or different. However in this case, the values of the data, i.e. recvcnt, should not be the same otherwise MPI_Bcast will do the job since it is easy to implement.

- MPI_Scatterv :

API for MPI_Scatterv is shown below.

int MPI_Scatterv (void *sendbuf, int *sendcnts, int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);

As illustrated in Fig. 4.11, data ABCDEFGH from root process P_0 are scattered so that processors P_0, P_1, P_2 will each receive ABC, DEF, GH respectively and the data received by each processor have unequal size. Here suppose only 3 processors are used. If 4 processors are used, both MPI_Scatter and MPI_Scatterv can be employed in this case. In this sense, the latter also refers to non-uniform data scattering. Obviously, the difference between these two routines is that the former can only scatter the same amount of data to each processor based on pre-specified values. In other words, *sendcnts and recvcnt in above API must be

specified before this function call. However, the latter is more flexible, sizes of the data scattered to each processor are not necessarily the same as shown in Fig. 4.11. Definitely the parameters passing for these two routines are not the same either. Ideas behind this function routine are very important for all the developed MPI codes. It allows the MPI codes to be executed with any number of processors and each processor has almost even workload. Sample code is shown in Fig.4.12.

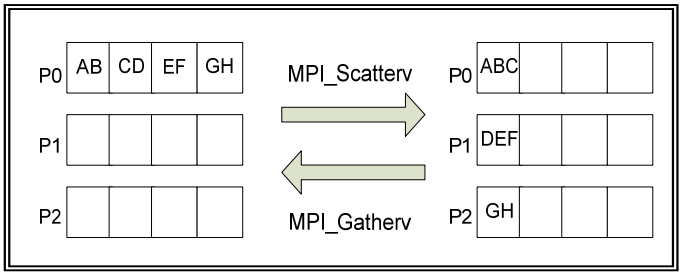

Fig.4.11 MPI_Scatterv and MPI_Gatherv illustration

```
MPI_Scatterv (EigenValuesDk,sendcnt,displs,MPI_DOUBLE,Lamda,recvcnt,
MPI_DOUBLE,0,MPI_COMM_WORLD);
```

Fig.4.12 Sample code for MPI_Scatterv

Eigenvalues array **EigenvaluesDk** is scattered non-uniformly to each processor depending on partitioning.

- MPI_Gatherv: It is the reverse operation of MPI_Scatterv as shown Fig. 4.11, data ABC, DEF, GH from P_0, P_1, P_2 respectively are collected to root processor P_0. However, parameters passing for these two routines are different. API is shown below.

int MPI_Gatherv (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)

For example, the following code is implemented in algorithm #3 as shown in Fig.4.13.

```
MPI_Gatherv (LamdaConnSingle, recvcnt, MPI_DOUBLE, LamdaConn, sendcnt,
displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Fig.4.13 Sample code for MPI_Gatherv

Data **LamdaConnSingle** from each processor is collected to **LamdaConn**.

- MPI_Allreduce: This function performs global reduction. It is equivalent to MPI_Reduce followed by MPI_Bcast. This could be seen clearly in Fig. 4.14. API is shown below.

MPI_Allreduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);

Many built-in functional operations [14] are available for MPI_Allreduce, i.e. MPI_Op in above API. The operation used in the algorithm is MPI_SUM. As illustrated in Fig. 4.14, data A, B, C, D in the corresponding buffer of processors P_0, P_1, P_2, P_3 are summed to form A+B+C+D and the sum is returned to the destination buffer of each processor. The data, A, B, C, D, can be either single element or arrays.

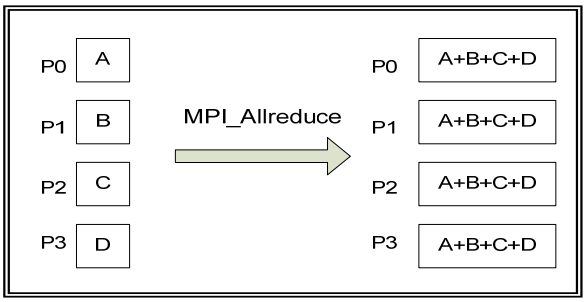

Fig.4.14 MPI_Allreduce illustration

The sample code is shown in Fig.4.15.

```
MPI_Allreduce (&Sum1, &Sum3, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Fig.4.15 Sample code for MPI_Allreduce

After this function is called, all the processors will have the value of **Sum3**, i.e. A+B+C+D in Fig. 4.14, which is obtained by adding **Sum1**, i.e. A, B, C, and D, from each processor respectively. Here A, B, C and D represent the same variable **Sum1**, however, they have different values in this case.

All the above MPI routines are essential for the developed codes although each algorithm only employs some of them.

4.4 Software Packages and APIs

This section will focus on detailed illustration for the algorithms developed. Several algorithms will be introduced and detailed performance analysis will be discussed in Chapter 5. The case of two islanded systems connected by a tie line will be studied here. The function to update eigensystems of this kind is called Connect. For the cases of connecting or disconnecting a line from the existing system, they are quite similar to connection except the followings:

- Eigen vector matrix and eigenvalue array are not to be rearranged as shown in (3.3.2) since only one eigensystem exists.
- Sorting for eigenvalues is not necessary since from previous stage the eigenvalues are already in an ascending order

In this sense, the function routine for adding or disconnecting one line is less complicated. Adding a line to the system was implemented by the function routine Install. On the other hand, in order to test on accumulating errors, the cases of a series of connections have been tested.

4.4.1 Main Function Structure

The general software structure of MAIN function is shown in Appendix B. As one can see that it includes three parts: A, B and C.

Part A:

It mainly includes MPI and #C header files and some global constants declaration. MPI environment requires MPI header files, i.e. # include “mpi.h”. The other #C header files and constant declarations are shown as comments in the attached programs.

Part B:

B1: MPI environment initialization

- MPI_Init(&argc, &argv)-Initialization of MPI environment
- MPI_Comm_rank(MPI_COMM_WORLD, &rank) – after this function call, each processor will have an unique rank number from 0 to n-1, where n represents the number of processors to be used and it is specified in an Portable Batch System (PBS) script file. A sample script file is found in Appendix J.

- MPI_Comm_size(MPI_COMM_WORLD, &nprocs) – This function call is equivalent to: nprocs = n.

B2: Data input and broadcasting

MPI-I requires that only root processor, i.e. processor with rank = 0, can read data input, instead, MPI-II allows parallel input and output (I/O). For the specific project problem, it is efficient enough using MPI-I and better to do so. The pseudo code for data input is shown in Fig. 4.16:

```
if ( rank= = 0 ) {
    Read data in text files from local directory…..
}
```

Fig.4.16 Pseudo code for data input

Note that the initial data come from MATLAB results which are eigensystems of two original islanded sub-networks. And data input is once for all since a series of ROMs to the entire network can be computed based on the previous stage results. It will be clearer later in the discussion of 4 series of connections of 4419-node system in Chaper 5. Since only P_0 read data input, they are unknown to other processors, after MPI_Bcast function call, all the processor will have the data input as discussed in Fig.4.7. For example, the following code is for data broadcasting shown in Fig.4.17.

```
MPI_Bcast (EigenVec1,MAX*MAX, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Fig.4.17 Sample code of MPI_Bcast for input data

Input:

EigenVec1 - data to be broadcast.

MAX*MAX – data size where MAX is a predefined constant

MPI_DOUBLE - data type in MPI

0 – Source processor of the data, i.e. where the data come from

MPI_COMM_WORLD - Default communicator in which this message passing occurs.

B3: Connect function call

This function performs update for eigensystems of matrices subject to ROMs. In other words, this is the main function call developed to solve (3.2.8) and (3.2.11). This function routine will be discussed in details in the algorithms section. A few versions will be illustrated.

B4: Print out computed results for later studies. It is implemented as follows shown in Fig.4.18:

```
if ( rank = = 0 ){
    Print output to local directories……
}
```

Fig.4.18 Pseudo code for output printout

Note that if without the condition *if*, all the processors will print duplicate sets of output which is usually not necessary.

Part C: Function termination

MPI_Finalize – this is a void function call which is to terminate MPI environment and program stops execution.

4.4.2 Connection Function Structure

This section focuses on detailed implementation of function Connect. After this function call, the eigensystems of specified matrix which is subjected to ROMs will be updated based on (3.2.8) and (3.2.11). Definitely the calculation is in parallel and the function has been designed in a way that it works for any number of processors, which is flexible to use and mainly for research purpose as well as future applications. Task partitioning will be discussed in details shortly. A few algorithms are illustrated below and related software structures as well as programs are attached. Only algorithm # 1, algorithm #3 and #4 will be discussed in details due to the following facts:

- Algorithms #2, 3 and 4 are similar except that the latter two algorithms are implemented with the concept of unroll looping in computing eigenvectors.

- Algorithm #1 is the first one to be developed. Although it turned out to be the least effective compared to other three algorithms, it is a good practice to learn. And it is good if only a few or part of the eigensystems are to be updated. Compared to sequential coding, it is still a lot faster which can be seen in the results sections.
- Algorithm #4 differs from #3 in the ideas of task partitioning and turns out to be the most efficient.

```
void Connect (
                    int ProcessorNum,
                    int Nodes1,
                    intNodes2,
                    double EigenVectors1[MAX][MAX],
                    double EigenVectors2[MAX][MAX],
                    double EigenValues1[MAX],
                    double EigenValues2[MAX],
                    int Tie1,
                    intTie2,
                    double Line,
                    double EigenVectorFinal[MAX][MAX],
                    double SigmaFinal[MAX]
                );
```

Fig.4.19 API for Connect function

Input:

ProcessorNum - number of processors to use; **Nodes1**- number of nodes in system1; **Nodes2** - number of nodes in system2; **EigenVectors1**- eigenvector matrix of system1;

EigenVectors2- eigenvector matrix of system2;

EigenValues1 - eigenvalues set of system1;

Eigen Values2 - eigenvalues set of system2; **Tie1** - Tie-in point in system1;

Tie2 - Tie-in point in system2; **Line** - α_l ;

Output:

EigenVectorFinal - Updated eigenvetors matrix;

SigmaFinal - Updated eigenvalues array;

Note that the following algorithms discussion will be based on 4062-node system and 64 processors implementation for the computation. The API for Connect function is shown in Fig.4.19 followed by inputs & outputs for this function call.

4.4.2.1 Algorithm #1

The original idea behind algorithm #1 is to employ as many MPI routines as possible to realize parallel computing. However, when developing this algorithm, a fact was ignored that too much communication occurred during the computation which is also the root cause of its inefficiency. The software structure is shown in Appendix C and MPI in #C source codes are shown in Appendix D.

Before further discussion, one needs to carefully observe the structure of (3.2.8) and (3.2.11). Note that secular equation (3.5.1) can be expanded as follows:

$$f(\sigma) = 1 + \alpha_\ell \left[\frac{(\phi_{1i} - \phi_{1j})^2}{\mu_1 - \sigma} + \frac{(\phi_{2i} - \phi_{2j})^2}{\mu_2 - \sigma} + ... + \frac{(\phi_{ni} - \phi_{nj})^2}{\mu_n - \sigma} \right] \tag{4.2}$$

In addition, based on (3.2.11) the corresponding eigenvector $\underline{w}_m$ of σ_m can be expressed as

$$\underline{w}_m = \frac{(\phi_{1i} - \phi_{1j})}{\mu_1 - \sigma_m} \underline{\phi_1} + \frac{(\phi_{2i} - \phi_{2j})}{\mu_2 - \sigma_m} \underline{\phi_2} + ... + \frac{(\phi_{ni} - \phi_{nj})}{\mu_n - \sigma_m} \tag{4.3}$$

During the research, for raw data creation, all the generators inertias have been set to one without loss of generality as long as the matrix $[\psi_0]$ is symmetric and its eigenvalues are distinct before and after ROMs. (4.2) is a sum of n terms plus one (constant term) and (4.3) is just the sum of the n original eigenvectors of $[\psi_0]$ multiplied by a scalar. (4.2) is computed based on bisection method [19] which is to find the roots of (4.2) iteratively. Briefly, for a single iteration, one needs to compute f based on an estimated value of σ . If $|\varepsilon| = \left| \frac{\sigma_{k+1} - \sigma_k}{\sigma_k} \right| < es$, where $es = 10^{-16}$ is the pre-defined bound error, then one assumes to have found the root within the specified interval and continue with next root-finding process until all the roots have been found, i.e. all the eigenvalues of newly connected system. Note that for a single iteration, one needs to compute a sum of *n* terms as shown in (4.2). (4.3) is to compute each eigenvector as long as its corresponding

eigenvalue has been solved. In sequential computing, usually (4.3) is implemented with double FOR loops. The cost is O (n^2) for a single eigenvector computation. Based on above analysis one concludes that (4.2) and (4.3) posses the nature of being implemented in a parallel processing environment. Before further discussion, two things are to be done.

- Eigensystems from two sub-networks are put into the form of (3.3.2), which is for the purpose of easy implementation with parallel computing.
- Quick sort was used to sort eigenvalues array in an ascending order. For a 4062-node system, the sorting could be completed within the order of milliseconds.

In the attached parallel algorithm #1, two MPI routines are used intensively: MPI_Allreduce and MPI_Scatterv. The basic idea for this parallel implementation is that different processors will execute the same segments of codes with different data. Before going into details for computing (4.2) and (4.3), one needs to partition the data, i.e. eigenvalue array and eigenvector matrix which have already been put in the form of (3.3.2). MPI_Scatterv was employed in this algorithm for data partitioning. The rule of thumb for data and task partitioning is that each processor will have almost balanced computation workload. Before MPI_Scatterv is called, one needs to specify the amount of data that each processor will receive, e.g. how many eigenvalues each processor will receive to compute the terms in (4.2) and (4.3).

Part A: Compute (4.2)

Step1: Partition eigenvalues array into 64 segment.

Define

$r = remainder(\frac{4062}{64})$ and $num = \text{int}(\frac{4062}{64})$. One has r = 30 and num = 63. Thus the first 30 processors, i.e. processor with rank 0 ~ 29 will receive 64 eigenvalues and the remaining 34 processors will receive 63 eigenvalues. This is could be seen as follows:

$$\underline{\mu} = \left(\underbrace{\mu_1, \mu_2 \cdots \mu_{64}}_{0(64)}, \underbrace{\mu_{65}, \mu_{66} \cdots \mu_{128}}_{1(64)} \cdots\cdots \underbrace{\mu_{1921}, \mu_{1922} \cdots \mu_{1983}}_{31(63)} \cdots\cdots \underbrace{\mu_{4000}, \mu_{4001} \cdots \mu_{4062}}_{64(63)} \right)$$

where $\underline{\mu}$ represents the entire eigenvalue array; $_{0(64)}$ represents processor 0 receives 64 eigenvalues. Similarly partition the i^{th} and j^{th} rows of [Φ] into 64 segments and the idea above applies to here. Note that one can not do the partition directly and needs to store the i^{th} and j^{th} rows of $[\Phi]$ into two new vectors $\in R^{1Xn}$. With partition above, each processor will only needs to compute a sum of 63 or 64 terms in (4.2) and (4.3) respectively. Each term is defined as $\frac{(\phi_{ki}-\phi_{kj})^2}{\mu_k-\sigma}$ during a single iteration.

Step2: Define the sub-total above as S_i for the i^{th} processor, the next step is to compute $\sum_{i=1}^{64} S_i$ in order to obtain f value in (4.2). Usually in sequential coding, to sum up the sum-totals S_i is done within a single For loop. However, MPI_Allreduce will compute it with only one operation, it is also called global reduction as discussed earlier. This could also be done with MPI_Reduce. The former will store the results in the corresponding address of each processor while the latter only in root processor, i.e. P_0 will obtain the final f value. Note that for a single iteration of bisection method, two f values are to be calculated, one is evaluated at endpoint of each new sub-interval, i.e. either new upper or lower limit and the other is evaluated at the middle point of the new sub-interval. For $es=10^{-16}$, the total iteration times are around 50 for a single eigenvalue calculation.

Part B: Compute (4.3)

One notices that the numerator of the coefficients for each term in (4.3) has been already computed while evaluating (4.2) except the i^{th} term is multiplied by $\frac{1}{\mu_i-\sigma_m}$, where σ_m is the corresponding newly computed eigenvalue. This time one needs to partition the eigenvector matrix $[\Phi]$. Following the similar ideas in part A, partition is done in the following way:

$$[\Phi]=[\underbrace{\underline{\phi}_1,\underline{\phi}_2\cdots\underline{\phi}_{64}}_{0},\underbrace{\underline{\phi}_{65},\cdots\underline{\phi}_{128}}_{1}\cdots\cdots\underbrace{\underline{\phi}_{1921},\underline{\phi}_{1922},\cdots\underline{\phi}_{1983}}_{31}\cdots\cdots\underbrace{\underline{\phi}_{4000},\underline{\phi}_{4001}\cdots\underline{\phi}_{4062}}_{64}]$$

Note that $[\Phi]$ matrix must be in its transposed form, i.e. the partition above is for rows of $[\Phi^T]$ instead of columns of $[\Phi]$.

Each processor will compute a sub-total of 63 or 64 vectors $\in R^{4062 X1}$ and the sum is also a vector $\in R^{4062 X1}$. For computing $\underline{w}_m$, MPI_Allreduce will perform the similar operation as in step 2 of part A. This time MPI_Allreduce manipulates on a vector $\in R^{4062 X1}$ from each processor and ends up global reduction with $\underline{w}_m$. Followed by normalization, $\underline{w}_m$ becomes $\underline{w}_m = \frac{\underline{w}_m}{\|\underline{w}_m\|_2}$. This is for computing a single eigenvector. The process continues until all the 4062 eigenpairs are computed.

In summary, the new eigensystem update is to repeat 4062 times the above processes. Suppose 50 iterations are required to obtain the new eigenvalue, in a single process, 100 MPI_Allreduce operations for eigenvalue and 1 MPI_Allreduce for eigenvector calculation. Based on research experience, it turned out that the communication time dominated the computation process especially when only a few processors are involved. Algorithms #2, #3 and #4 have employed new ideas which greatly improved the performance. Note that although this algorithm is inefficient for updating large eigensystems, it will work well if only several selected eigenpairs are to be updated in real applications.

4.4.2.2 Algorithm #3

Observe that from (4.2) and (4.3) computing a single eigenpair is independent of each other, i.e. the computation tasks are decoupled. In the following algorithms illustration, data parallel is used. In total, 4062 eigenpairs are to be updated, that is, each processor will compute only 63 or 64 eigenpairs when 64 processors are involved. The ideas illustrated in algorithm #1 also apply here so that both tasks partition and data partition are used. However, only eigenvalues array is partitioned, and partition of $[\Phi]$ is unnecessary. The software structure for this algorithm is shown in Appendix E and MPI in #C source codes in Appendix F. The basic steps are shown below.

Step 1: Partition eigenvalues array as follows:

$$\underline{\mu} = \left(\underbrace{\mu_1, \mu_2 \cdots \mu_{64}}_{0(64)}, \underbrace{\mu_{65}, \mu_{66} \cdots \mu_{128}}_{1(64)} \cdots\cdots \underbrace{\mu_{1921}, \mu_{1922} \cdots \mu_{1983}}_{31(63)} \cdots\cdots \underbrace{\mu_{4000}, \mu_{4001} \cdots \mu_{4062}}_{64(63)} \right).$$

At the same time, each processor will know the number of eigenpairs it is supposed to compute, i.e. 63 or 64 eigenpairs.

Step 2: After each processor has completed its own computation task, the root processor P_0 will gather the results. In this case, only two MPI_Gathervs are used, one for eigenvalues collection and the other is for eigenvectors.

In summary, the above process requires one MPI_Scatter, one MPI_Scatterv and two MPI_Gatherv. The whole process is equivalent, from the point of view of sequential coding, to computing 63 or 64 eigenpairs on a single-processor machine. The 4 MPI operations, i.e. 2 scattering and 2 gatherings could be completed within the order of milliseconds which is negligible compared to the total computation time. The pseudo code is shown in Fig.4.20 in part A.

Part A: Pseudo codes for computing eigenvectors

Start: MPI_Scatterv to partition eigenvalues array and tasks.

```
for ( i = 0; i < recvcnt; i++ ){
    63 or 64 eigenvalues computation…..
    63 or 64 eigenvectors computation….
}
```

Fig.4.20 Pseudo code for eigenpairs computation of algorithm #3

where *recvcnt* represents tasks number and its value is different for each processor depending on the real partitioning.

End: MPI_Gatherv is to collect the computed eigenpairs from each processor.

Note that eigenvector computation is the most costly part. In order to improve the performance, loop unrolling [23] concept was used in the algorithm. This also refers to code optimization technique. It turned out to increase the overall computation speed by a factor of 3 ~ 4 compared to that without loop unrolling. The unroll factor was set manually to unroll=8 and the speed was maximized for

this specific problem. With and without unrolling loops pseudo codes for computing a single eigenvector are shown in Fig.4.21 and Fig.4.22 in part B.

Part B: Pseudo codes comparison with and without unrolling loops

```
Without unrolling: double For loops
Start: /*Declaration of pointers for computation*/
        double *temp;
        /* Dynamic memory allocation for pointer*/
        temp= (double*)malloc(Nodes*sizeof(double));
        /* Initialization of temp vector*/
        for ( i = 0; i < Nodes; i++)
                temp [i] = 0;
        /* Compute a single vector*/
        for ( i = 0; i < Nodes; i++)
                for (j = 0;j < Nodes; j++)
                        temp[j] + = ( each tem in (4.3));
End: Single eigenvector normalization….
```

Fig.4.21 Without unrolling loops pseudo code illustration

Note that in Fig.4.22, unroll looping compute 8 eigenvectors for each iteration within the FOR loop. Of course the total computation time is not just simply 8 times faster than that of computing a single vector each time. It is usually slower depends on the type of compilers used. On the other hand, in part A For loop, *recvcnt* requires to be a multiple of unroll factor, i.e. $r = remainder(\frac{recvcnt}{unroll}) = 0$.

In the case, if it is not, a separate loop is used to compute the leftovers of the eigenvectors. The iterations are less than unroll factor, i.e. iterations<8 in this case.

Unrolling loop performance depends also on the compilers used. If a compiler vectorizes loops, e.g. icc compilers, the unrolling effect will interfere with loop vectorization function of the compilers. For a 4062-node system, the first 30 processors, i.e. rank ranges from 0 ~ 29, the second separate For loop actually does not perform any computation since 64 is a multiple of 8. However for other processors, i.e. rank ranges from 30 ~ 63 will compute and iteration times is 7. All

the processors must be synchronized before the results are collected to root processor.

```
Unrolling: triple For loops
Start: /* Define constant unroll factor*/
       #define unroll 8
       /* Declaration of pointers array*/
       double *temp[unroll];
       /* Dynamic memory allocation for the array of pointers*/
       for( i=0;i<unroll;i++)
              temp[i]= (double*)malloc(Nodes*sizeof(double));
       /* Initialization of temp vector*/
       for(i=0;i<unroll;i++)
              for (j=0;j<Nodes;j++)
                     temp[i][j]=0;
       /* Compute 8 vectors each time*/
       for (i=0;i<Nodes;i++)
              for (j=0;j<Nodes;j++)
                     for (u=0;u<unroll;u++)
                            temp[u][j]+=( each tem in (4.3) );
End: Vector normalization for 8 eigenvectors…
```

Fig.4.22 Unrolling loops pseudo code illustration

4.4.2.3 Algorithm #4

```
for ( i = iMin; i < iMax; i++ ){
    63 or 64 eigenvalues computation…..
    63 or 64 eigenvectors computation….
}
```

Fig.4.23 Pseudo code for eigenvectors computation of algorithm #4

This algorithm is similar to #3 except that only one MPI routine used in this algorithm, i.e. 2 MPI_Gatherv for collecting updated eigensystems. Since eigenvalues array has been broadcast before the Connect function is called, all the processors know the entries of the entire array. Instead of partitioning the eigenvalues array, one may partition the indices, i.e. 1 ~ 4062. The partitioning of

indices follows the same idea as shown in algorithm #3. The indices partitioning is shown below.

$$index = \left[\underbrace{1,2,3,\cdots 64}_{0}, \underbrace{65,66,\cdots 128}_{1}, \cdots\cdots \underbrace{1921,1922,\cdots 1983}_{31}, \cdots\cdots, \underbrace{4000,4001,\cdots 4062}_{64} \right]$$

The For loop of part A in section 4.4.2.2 then becomes, as shown in Fig.4.23,

For example, P_0 will have array $\underline{\mu_0} = (\mu_1, \mu_2, \cdots, \mu_{64}, \mu_{65})$ as its upper and lower bounds. And $iMin = 0, iMax = 64$; P_1 will have array $\underline{\mu_1} = (\mu_{65}, \mu_{66}, \cdots, \mu_{128}, \mu_{129})$ as its upper and lower bounds. And $iMin = 64, iMax = 128$ etc. See also the attached programs for detailed implementation. In doing this way, eigenvalues array is not necessary to be partitioned as done in algorithm #3.

4.4.3 Adding or Removal Function Structure

```
void Install (
                int ProcessorNum,
                int Nodes,
                double EigenVectors[MAX][MAX],
                double EigenValues[MAX],
                int Tie1,
                intTie2,
                double Line,
                double EigenVectorFinal[MAX][MAX],
                double SigmaFinal[MAX]
            );
```

Fig.4.24 API for Install function

As mentioned earlier, connection or disconnection of line within the existing system can be computed in a similar way. API for this function is shown in Fig.4.24. Note that the inputs and outputs preserve the same notation as those of Connect function except only one eigensystem is input to the function call. And the algorithm design is similar.

4.4.4 A Series of ROMs Update Software Structure

In order to test if cumulative errors occur when more than one ROM is required, a series of 4 connections testing was conducted during the research. To perform

this, no new functions are needed to be designed. The API for this case is shown in Fig.4.25 and only MPI in #C source code for algorithm #4 is shown in Appendix G since algorithm #3 have the same structures except that the detailed implementation of Connect and Install function differ as discussed in section 4.4.

```
Connect (nprocs, N1, N2, Evec1, Evec2, Eval1, Eval2, Ti1, Tj1, L1, Vec1, Sigma1);
Connect (nprocs, N1+N2, N3, Vec1, Evec3, Sigma1, Eval3, Ti2, Tj2, L2, Vec2, Sigma2);
Connect (nprocs, N1+N2+N3, N4, Vec2, Evec4, Sigma2, Eval4, Ti3, Tj3, L3,Vec3, Sigma3);
Install (nprocs, N1+N2+N3+N4, Vec3, Sigma3, Ti4, Tj4, L4, Vec4, Sigma4);
```

Fig.4.25 API for four ROMs

Output:

Vec4 – Updated eigenvector matrix after the 4^{th} ROM
Sigma4 – Updated eigenvalues array after the 4^{th} ROM

Note that for previously designed functions, if they are applied to a series of connections, two more MPI functions are needed at the end of the program, i.e. MPI_Bcast. Since the functions employ MPI_Gatherv to collect the results, i.e. after this function, only root processor P_0 has the entire updated eigensystems, the others only have part of them. In order for the computed results to be used for the next stages, two MPI_Bcast are required at the end of the program. The readers are encouraged to employ MPI_Allgatherv to collect the results to see the difference in performance. In this case, MPI_Bcast is not needed.

Chapter 5 Results and Discussions

This chapter will discuss the tests which have been conducted during the research to evaluate the computational speed, robustness, efficiency and accuracy of the B & B method. The developed MPI codes were executed on Krylov cluster of CLUMEQ and Mammouth Series II cluster of RQCHP. The compiler used is intel64/11.1.038. The tests were performed on two system topologies:

1. A 4062-node system which is formed by connecting two isolated systems as shown in Fig.5.1. The testing results are described in section 5.1 together with detailed timing analysis and comparisons. .
2. A 4419-node system, as shown in Fig. 5.7, originally consists of four isolated systems, #1, #2, #3 and #4. After three successive connections, each similar to that in section 5.1 and the four isolated systems are integrated as one, 1-2-3-4. The 4^{th} connection is implemented between two nodes within the entire system, i.e. connecting #1 and #4 to form 1-2-3-4-1. The testing results are described in section 5.2.

Prior to the tests, the information regarding the system is obtained by forming [K] using (2.1.19). Then it is transformed to [ψ_0] using (3.1.1). Since the tests are concerned about computation speed and accuracy of the results, all the generator inertias are set to one without loss of generality. The eigenpairs ($\underline{\mu}$,[Φ]) in equation (3.2.1) have been computed by MATLAB, which are considered as input data of the programs. In addition, error analysis is presented in section 5.4. All the results in this section were obtained from Mammouth Series II cluster of RQCHP.

5.1 4062-Node System

This section is a record of the trial-and-error tests which have been performed to maximize the computation speed.

Fig. 5.1 shows the testing network originally including sub-network # 1 composed of 2419 nodes and sub-network # 2 composed of 1643 nodes. The sub-networks are united by a tie line to form a 4062-node system. The new eigensystems are computed using the B & B method (3.2.8) and (3.2.11). Four algorithms have been proposed in order to find the shortest computation time. The results are tabulated in Table 5-1 and plotted in Fig. 5.2 and Fig. 5.3.

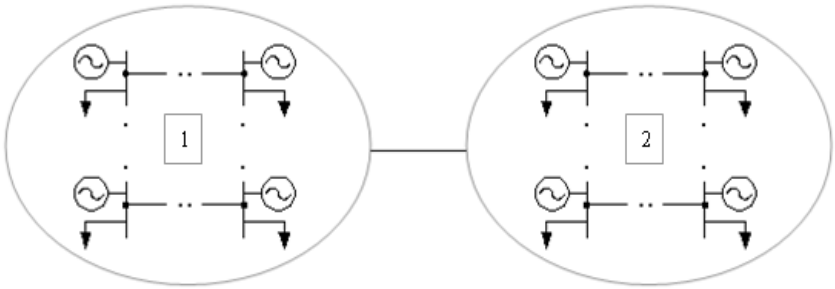

Fig.5.1 Single line diagram of 4062-node test system

Table 5-1: Overall computation time of algorithms #1 - #4

Number of CPUs		1	4	8	16	32	64	72	80	128
Algorithm #	1	177.1	77.18	51.63	43.89	43.93	x	X	x	x
	2	217.0	73.92	37.19	18.86	9.82	5.55	5.06	4.72	x
	3	92.01	25.78	13.14	7.00	4.02	2.69	2.41	2.39	3.12
	4	58.90	16.98	8.74	4.78	2.85	1.87	1.48	1.61	2.21

Note: x's in Table 5-1 represent the data unavailable.

- *Algorithm #1*: All the N CPUs are used to compute one eigenpair at a time. The computation process continues until all the eigenpairs have been computed.
- *Algorithm #2:* One CPU is supposed to compute 63 or 64 eigenpairs if 64 CPUs are employed. Thus 64 CPUs evaluate 4062 eigenpairs simultaneously. Total speed is limited to 64 eigenpairs computation without considering the communication time among processors and the time spent on the executions other than eigenpairs evaluation.

- *Algorithm #3*: This algorithm is the same as algorithm #2, except that the program speedup technique, i.e. unrolling For loops, has been incorporated as discussed in Chapter 4.
- *Algorithm #4:* This is the same as algorithm #3 except for the ways of task partitioning and ways of collecting the computed results.

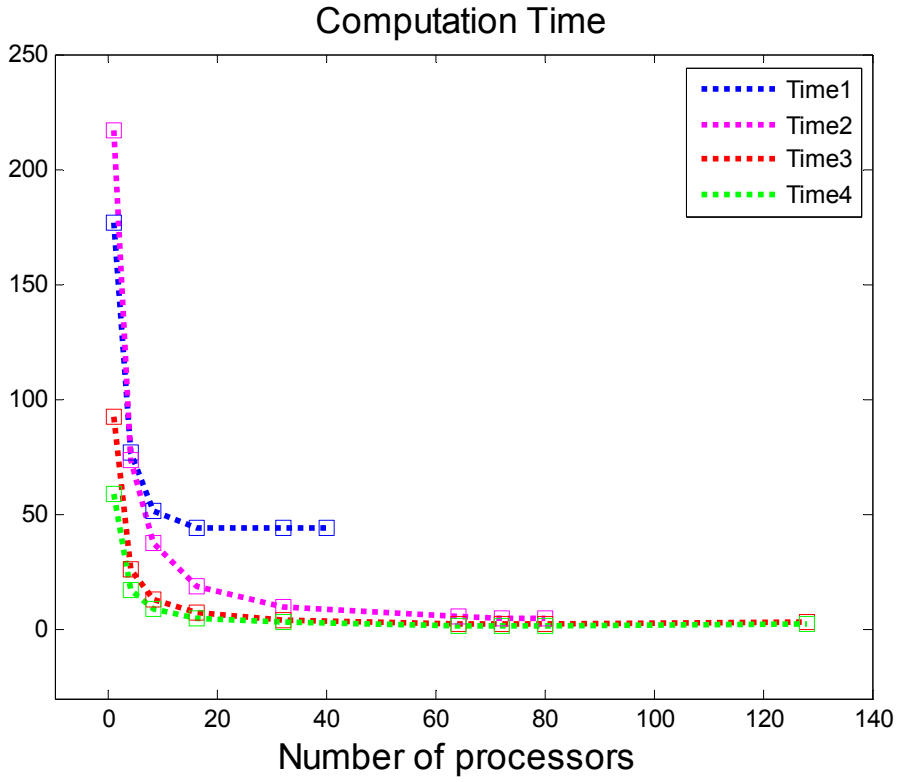

Fig. 5.2 Overall computation speed of algorithm #1, #2, #3 and #4

From Table 5-1 and Fig. 5.2, one notices that the maximum computing speed for updating a 4062-node system is 1.48 seconds achieved by algorithm #4 when 72 processors are used. Algorithm #3 has similar speed and it has its maximum speed at 2.39 seconds when 80 processors are used. The results are reasonable since these two algorithms are similar as discussed earlier. The least efficient algorithm is #1. Algorithm #2 is slower than #3 and #4 since unrolling loops are not used. For the same number of processors, e.g. N=64, algorithm #4 improves the computation time of algorithm #2 by a factor of about 3 ~ 4 since the programming optimization technique of "unroll the For loops" has been incorporated in computing the eigenvectors. The improvement is not retained for

more processors as shown in Table 5-1 when 80 or more processors are used. Based on the author's understanding, one interpretation might be that "unrolling loops" effect is mitigated when each processor has less computing tasks. In other words, as more processors are involved in the computation, unrolling loops will affect less on the computing speed. Therefore it is not surprising that algorithm #2 has higher speedup ratio since it costs more time to compute when only one processor is used.

5.1.1 Parallelism Portion Analysis

Recall from (4.1) and Amdahl's law [24], speedup ratio of an algorithm is related to its parallelism portion and the number of processors used. Ideally, if an infinite number of processors are available to compute, (4.1) simplifies $Speedup = \frac{1}{1-p}$.

Here speedup represents the maximum speedup ratio that an algorithm can achieve. Returning to Table 5-1, one draws the conclusion that:

- Algorithm #1 has its maximum $speedup = \frac{177.1}{43.89} = 4.04$ which means its parallelism portion is around 75%. As mentioned in Chapter 4, communication time among the processors dominated the computing process for both eigenvalues and eigenvectors, which has led to its inefficiency and low parallelism portion.
- Algorithm #2 has its maximum $speedup = \frac{217.0}{4.72} = 45.97$ which means its parallelism portion is around 99.8%
- Algorithm #3 has its maximum $speedup = \frac{92.01}{2.39} = 38.50$ which means its parallelism portion is around 99.75%
- Algorithm #4 has its maximum $speedup = \frac{58.9}{1.48} = 39.8$ which means its parallelism portion is a bit higher than algorithm #3. It makes sense since these two algorithms are similar and algorithm #4 is more efficient.

The resulting speedup ratios for all the algorithms are plotted in Fig. 5.3. It highlights the improvement as more processors are used. It shows that a maximum speedup of about 40 is achieved for algorithm #2, #3 and #4.

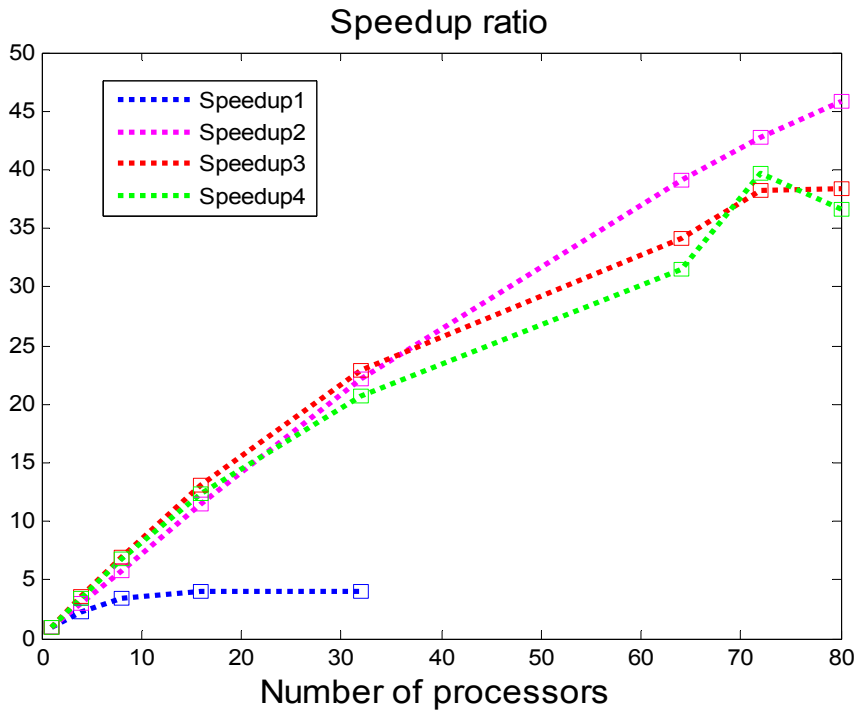

Fig.5.3 Speedup ratio of algorithm #1, #2, #3 and #4

5.1.2 Comparison with MATLAB Results

The time for MATLAB built-in function **eig** to compute the eigensystem of this 4062-node system on a desktop PC is around 336.1971 seconds. The parameters for the PC are: Intel® Core(TM) 2 Duo CPU and E6750 @ 2.66GHz and 3.48 GB of RAM. One can see that the total computation speed differs a lot from those with parallel implementations. Take algorithm #4 as an example, when one processor is employed, the updating speed is around 59 seconds which is around 6 times faster. Note that when computing with MATLAB, the starting time is after $[\psi]$ matrix has been formed. The reasons are obvious,

- MATLAB **eig** function employs QR algorithm or Jacobi method to compute the entire eigensystem which cost around $25N^3$ flops as discussed earlier. The proposed mathematical model (3.2.11) only costs $4N^3$ flops if no "unrolling

loops" effects are involved. The computing cost for (3.2.8), i.e. eigenvalues computation, is around $\frac{1}{18}$ of that based on testing.

- MATLAB **eig** uses iterative methods to compute eigensystems and iteration times are different for different matrices. (3.2.8) and (3.2.11) only involve sums and multiplications which can be called direct method in this sense. And the original information, i.e. the eigensystem before ROM is used based on the B & B method, while MATLAB does not.
- Another reason might be the CPU speeds are different for these two machines.

The purpose for this comparison is to show that parallel computing is indeed a powerful tool compared to sequential implementations.

5.1.3 Comparison of Performance with and without Unrolling Loops

Table 5-2: Comparison of unrolling effect of algorithm #3

Number of CPUs	1	2	4	8	16	32	64	72	80	96	128
Unrolling	87.89	44.09	24.16	12.28	6.33	3.36	1.52	1.40	1.35	1.65	0.99
W/o unrolling	212.8	x	72.57	36.31	18.10	9.82	4.56	4.07	3.64	x	x

Note: x's in Table 5-2 represent the data unavailable.

Table 5-3: Comparison of unrolling effect of algorithm #4

Number of CPUs	1	4	8	16	32	64
Unrolling	54.6590	15.4911	7.8230	4.0865	2.2163	1.1605
W/o Unrolling	169.1959	71.7664	35.8720	17.8142	8.9043	4.4667

As mentioned in Chapter 4, "unrolling loops" concept was incorporated for computing eigenvectors. The main idea is to increase the index strides and thus to reduce the conditional instructions administration. The most costly part of the

algorithms is to compute eigenvectors. Therefore it is necessary to conduct a detailed analysis. The unroll factor was manually set to 8 and with this factor the performance is maximized. Actually with unrolling loops the speed for computing eigenvectors only increases by a factor of 3 or 4. This can be seen in Table 5-2 and 5-3 and the resulting plot is shown in Fig. 5.4. Note that the timings in these tables are for eigenvector computation only.

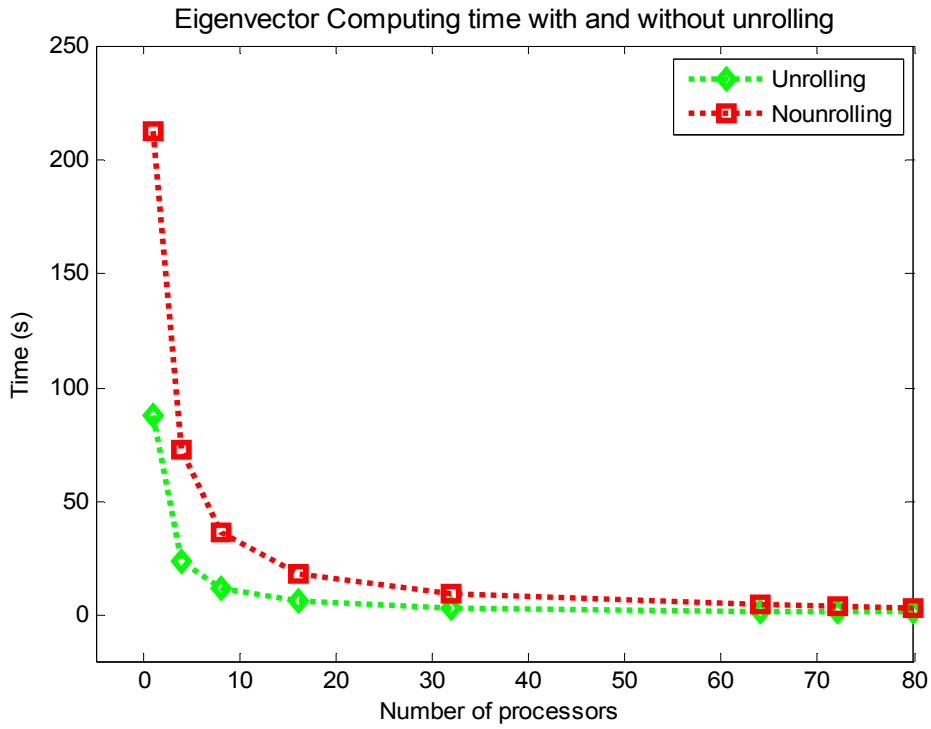

Fig.5.4 Unrolling loops effect of algorithm # 4

From Table 5-2, Table 5-3 and Fig.5.4, one concludes that the employment of "unrolling loops" has indeed increased the speed by a factor of 3 ~ 4. The trend of curve in Fig. 5.4 will continue as more processors are involved. However, from the speedup ratio curves in Fig. 5.3, the overall performance deteriorates after the maximum speed has reached. The reason is that the overall performance should also take into account the communication time and other execution time during the computation. The "unrolling loops" effect is diminished as more processors are used. Ideally, the computation time of the eigenvector part is limited to compute a single eigenvector if unrolling loops are not used and 4062 processors are used.

5.1.4 Detailed Timing Analysis

This section presents a breakdown timing analysis for each part of algorithms #3 and #4. Table 5-4 lists the detailed timings for algorithm #3 when 64 processors are used and Fig.5.5 is the related notations. Table 5-5 lists the detailed timing of algorithm #4 and Fig. 5.6 is the related notations. The purpose of this testing is to explore the performance of MPI routines and the designed algorithms.

Table 5-4: Breakdown timing for algorithm #3 with 64 processors implementation

Init.	Input	Bcast	Sorting	Gather	Scat1	Scat2	Reduce
1.843492	4.30214	3.284619	0.000287	0.418082	0.000123	0.458545	0.010804
TotalC	Value	Vector	Other1	Other2	Other3	OT	Total
0.887554	0.059552	1.523112	0.218934	0.000048	0.000012	0.218994	2.689499

Notations :

Init – Initialization time
Input – Data input time
Bcast – Data broadcast time before function call Connect
Sorting – Quicksort time for eigenvalues
Gather – Gathering time for eigensystems when computation is done
Scat1 – Scattering time for 'recvcnt' to each processor
Scat2 – Scattering time for eigenvalues array
Reduce – MPI_Reduce time for computing largest eigenvalue.
TotalC – Total communication time for one function call Connect
Value – Computing time for eigenvalues
Vector – Computing time for eigenvectors
Other1 – Eigensystem rearrangement in the form of (3.3.2)
Other2 – Time for tasks partitioning before scattering
Other3 – First eigenvector computation time
OT – Other1+Other2+Other3
Total – Total computing time

Fig.5.5 Notations for Table 5-4

Note the following,

- The total computing time does not include MPI environment initialization, data input and broadcast time before the actual computation starts.
- Initialization, data input and broadcast should be the same since the codes and data are the same. The average initialization time is around 1~2 seconds. Even if it is counted into the computation time (it comes before Connect function call), the total computing time is still quite satisfactory.
- The average data input time is around 4 ~ 6 seconds and once for all since for a series of connections the data input comes from the previous ROMs results. In this sense, it is not a concern any more.
- The average broadcasting time before Connect function call is around 3 ~ 4 seconds. The reason for ignoring this is the same as that of the data input time.

Table 5-5: Breakdown timing for algorithm #4 with 64 processors implementation

Init.	Input	Bcast	Sort	Gather	OT1	OT2	OT3	Val	Vec	T
1.731	4.179	3.295	0.0003	0.226	0.213	0.0005	0.186	0.062	1.161	1.869

Notations:
Init – Initialization time
Input – Data input time
Bcast – Broadcast time before the function call Connect.
Sort – Quick sort time for eigenvalues
Gather – Gathering time for eigensystems when the computation is done
Val – Computing time for eigenvalues
Vec – Computing time for eigenvectors
OT1 – Eigensystems rearrangement in the form of (3.3.2)
OT2 - Time for tasks partitioning - FOR loops
OT3 - First eigenvector computation
T – Total computing time

Fig.5.6 Notations for Table 5-5

From Table 5-4 and Table 5-5, one concludes,

- Initialization, data input, broadcast, sorting, OT1, OT3 time should be the same for both algorithms since they executed the same segments of the codes. The tiny difference is due to the instability of super-computers.

- The indexing used in the two algorithms is different although the computation algorithms are the same. Therefore, the computation time for both eigenvalues and eigenvectors are slightly different.
- Gather time is different although these two algorithms both use two MPI_Gathervs to collect the results, i.e. eigenvalues and eigenvectors, the implementation is bit different.
- The main reason that algorithm #3 has slower rate is due to the scattering time on eigenvalue array and the time difference of eigenvector computation.

From the above analysis, algorithm #4 is definitely faster than algorithm #3. More complex models for timing analysis can be found in [20].

5.2 4419-Node System

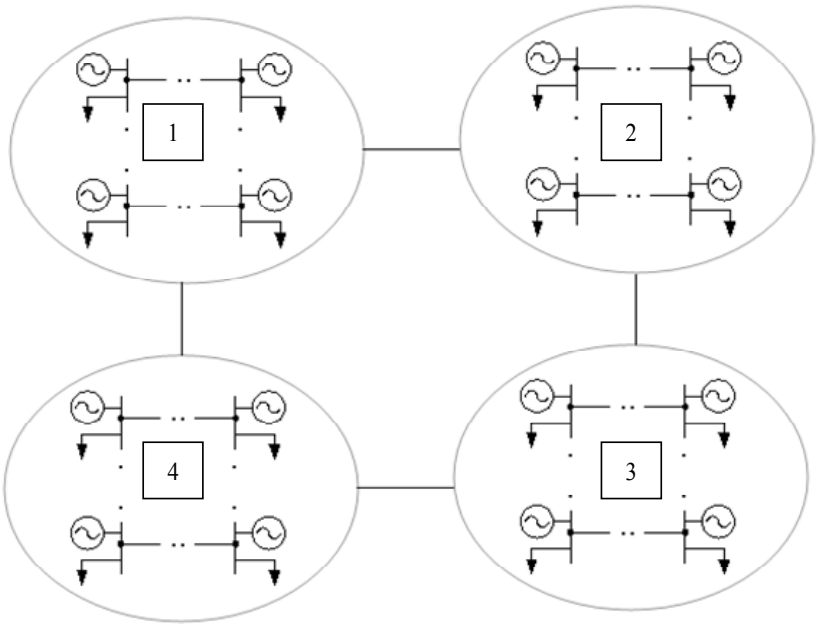

Fig.5.7 Single line diagram of 4419-node test system

The 4419-node system in Fig. 5.7 is intended to demonstrate that:

- The B & B method can be applied in successive connections of sub-networks such as #1, #2, #3 and #4 to form an integrated system.
- The B & B method also applies in the situations of connecting & disconnecting a single line between any two nodes in the entire system.
- The 4419-node system of Fig. 5.7 is to evaluate if cumulative errors occur when a series of ROMs are conducted.

5.2.1 System Parameters

In Fig.5.7, sub-network # 1 is composed of 651 nodes; sub-network # 2 is composed of 1312 nodes; sub-network # 3 is composed of 1470 nodes and sub-network # 4 is composed of 986 nodes. The complete power grid is thus formed by 4419 nodes. Table 5-6 shows the line parameters and tie-in points for this system.

Table 5-6: 4419 nodes system parameters

Sub-networks Interconnected	Connection Line	Tie-in points	
		i	J
1 - 2	2.03	31	652
1,2 - 3	3.08	684	1964
1, 2, 3 - 4	6.08	2005	3434
1, 2, 3, 4-1	4.5	4419	631

5.2.2 Overall Computation Speed

Table 5-7: Overall computation speed for 4419-node system for algorithm #3 and #4

Number of CPUs		1	4	8	16	32	64	72	80
Algorithm	3	307.1	86.91	45.47	25.12	14.57	10.05	9.56	9.11
	4	180.86	53.83	29.52	17.68	11.12	9.13	8.38	8.04

To compute the eigensystem of this ring network is equivalent to computing four ROMs with three Connect functions and one Install function as shown in Chapter 4. Only algorithms #3 and #4 were tested for this system. The results are satisfactory for both computing speed and accuracies. Table 5-7 shows the overall computation speed and Fig.5.8 shows the resulting plots.

One can see that computing speed of algorithm # 4 is faster than that of algorithm #3. And its maximum speed occurs at 8.04 seconds with 80 processors

implementations, which is around 5 times of that for one ROM in Fig. 5.1. The total computing speed is not 4 times of that for single ROM, which has its maximum of 1.48 seconds as shown in Table 5-1. It is due to the following facts:

- The size of 4419-node system is bit larger.
- As mentioned in Chapter 4, Connect function will have two more MPI_Bcast routines at the end if it applies to a series of connections. Each MPI_Bcast routine will cost around 0.4 seconds, therefore three connections and one adding will take around 1.6 more seconds.

The resulting plot is shown in Fig. 5.8.

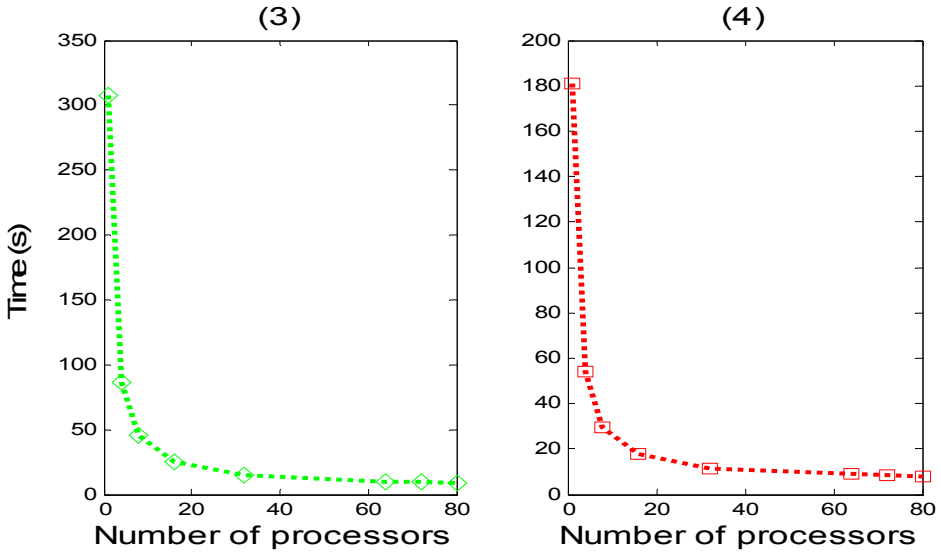

Fig.5.8 Overall computation speed for 4 ROMs – algorithm #3 and #4

5.2.2 Breakdown Timing for Each ROM Step

Table 5-8: Breakdown timing of 4419-node system for algorithm #4

Number of CPUs		1	4	8	16	32	48	56	64	72	80
Sub-networks Speed	**1-2**	6.91	2.41	1.64	1.16	0.98	0.94	0.93	0.96	0.94	0.91
	1,2-3	33.4	10.2	5.79	3.58	2.30	2.07	1.93	1.91	1.81	1.60
	1,2,3-4	70.1	20.7	11.0	6.42	3.88	3.23	3.29	3.11	2.78	2.73
	1,2,3,4-1	70.5	20.7	11.0	6.51	3.95	3.67	3.21	3.16	2.84	2.80

The breakdown timing is shown in Table 5-8 and Fig. 5.9 for each connection. One notice that ROM step 1 takes the least time since system 1-2 only has 1963 nodes. ROM step 2 takes almost double time of that for step 1 since 1, 2-3 system has 3433nodes, i.e. the computation workload almost doubles too. The connections of 1, 2, 3-4 and 1, 2, 3, 4-1 both have 4419 nodes. Thus it is not surprising that they take around the same time for computing, 2.73 and 2.80 seconds respectively in Table 5-8, when 80 CPUs are used.

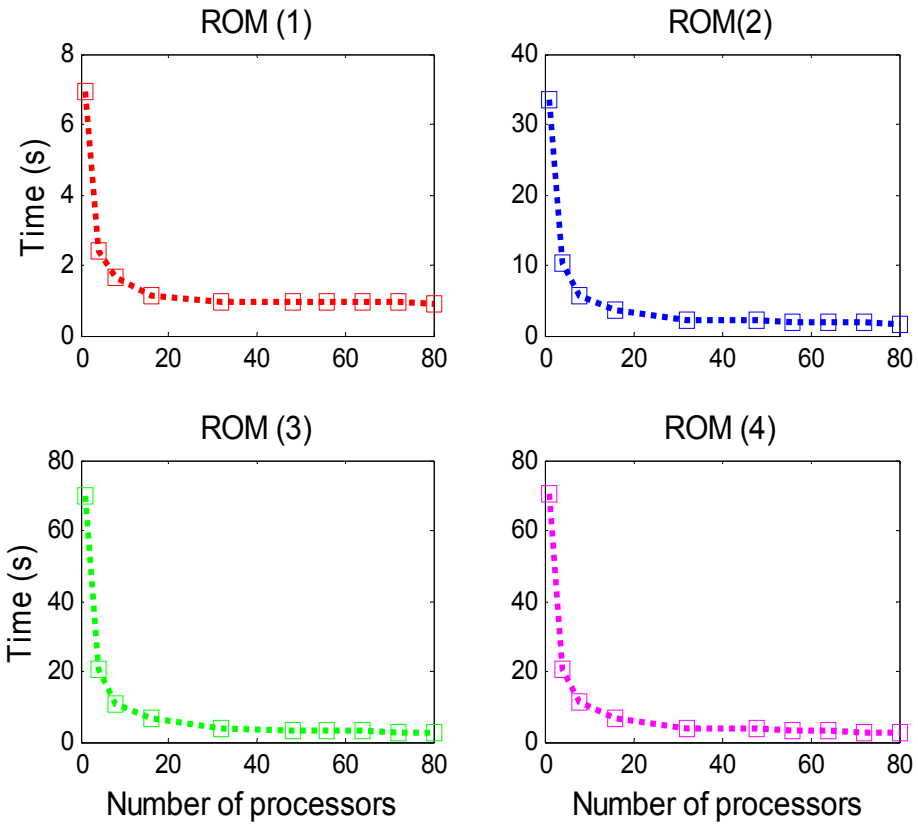

Fig.5.9 Breakdown timing of 4419-node system

5.3 Accuracy of Computed Eigensystems

The randomly selected computed eigenvalues of these two systems are shown in Table 5-9 and Table 5-10. The bar plots of randomly selected eigenvectors are shown in Fig. 5.10 and Fig. 5.11. In order to achieve higher accuracy, double precision was adopted for all the non-integer type variables and bisection method was used to compute the eigenvalues. The stopping criteria were set

to $|\sigma_k - \sigma_{k+1}| < 10^{-16}$. Absolute errors instead of relative errors, i.e. $\left|\frac{\sigma_k - \sigma_{k+1}}{\sigma_k}\right|$, are used due to the following reasons:

- Bisection method was used to compute eigenvalues in the developed algorithms. Instead, [11] suggested using Newton-Raphson method safeguarded by bisection method. Although Newton-Raphson method has quadratic convergence, with parallel implementation, computation speed is not a concern anymore and high accuracy is desired, bisection method is preferable for this specific problem.
- Bisection method will give any desirable accuracy if absolute errors criterion is adopted as long as machine precision allows without considering round-off errors. However, the results precision may vary if relative errors are used. This could also be observed from Fig. 5.12 and Fig. 5.13.

Table 5-9: Selected computed eigenvalues of 4062 nodes system

Mode Number	Eigenvalues	Methods
10	0.0**241485583335**8796	B & B
	0.0**241485583353**777	MATLAB
101	0.**4041664524525**856	B & B
	0.**4041664524524**702	MATLAB
1002	3.**3306185882139**125	B & B
	3.**3306185882133**694	MATLAB
2003	6.**1180137593990**267	B & B
	6.**1180137593991**253	MATLAB
3004	11.**3258240185703**407	B & B
	11.**3258240185702**821	MATLAB
4005	28.**1886533875**804197	B & B
	28.**1886533875**798513	MATLAB

The eigenvalues obtained by the B & B method are compared with MATLAB results shown in Table 5-9 and Table 5-10.The agreement (highlighted in bold) is within 10 to 12 significant figures for the 1st ROM, i.e. 4062-node system. However, by studying the complete eigenvalues set of 4419-node system, i.e. the updated eigenvalues after 4 ROMs, one observes that some of the eigenvalues do not preserve the same accuracy as those of only one ROM step. In other words, after a series of connections, the updated eigensystems might lose some accuracy. The reasons are many and will be discussed in section 5.4.

Table 5-10: Selected computed eigenvalues of 4419-node system – 4th ROM

Mode Number	Eigenvalues	Methods
15	0.0**381469370**733729	B & B
	0.0**3814693707**12625	MATLAB
106	0.**4717964773**112000	B & B
	0.**4717964773**988226	MATLAB
1007	4.**127396114**2319902	B & B
	4.**127396114**3560278	MATLAB
2008	9.**147881841**2687062	B & B
	9.**147881841**4624588	MATLAB
3009	15.**94463289455**35818	B & B
	15.**94463289455**38340	MATLAB
4010	28.**586626198**9053996	B & B
	28.**586626198**9134784	MATLAB

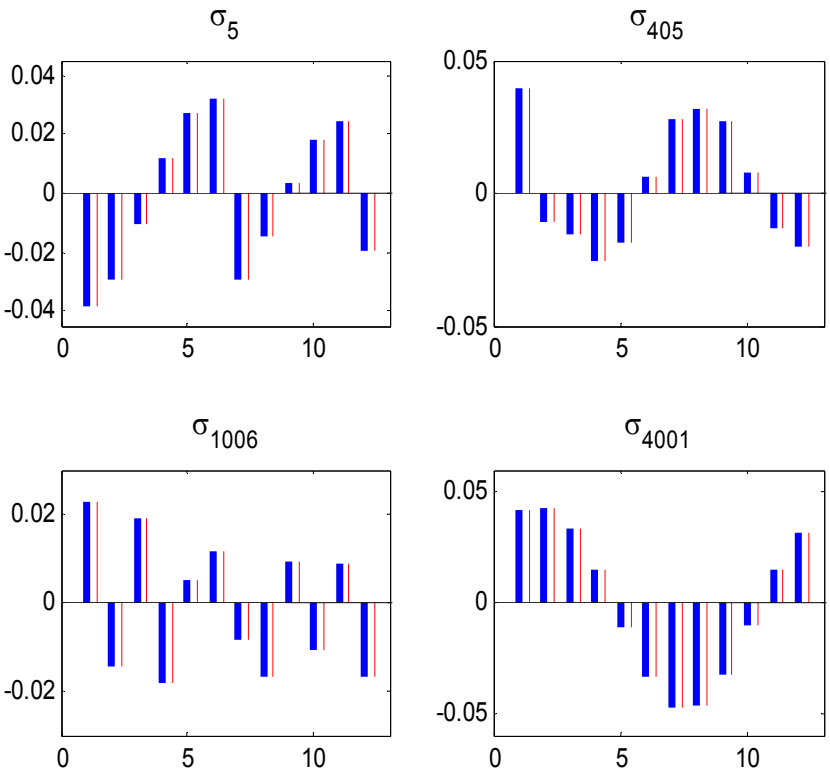

Fig.5.10 Selected eigenvector for 4062-node system

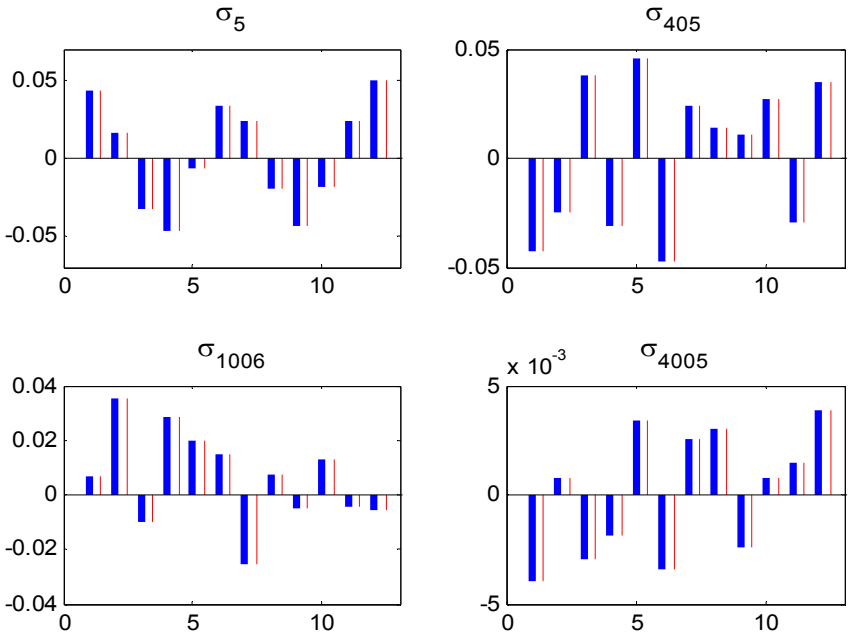

Fig.5.11 Selected eigenvector for 4419-node system

The bar plots in Fig. 5.10 and Fig 5.11 are the values of selected entries in the normalized eigenvector $\underline{w}_m$ with corresponding eigenvalue σ_m. The B & B values are placed beside those from MATLAB. The intent is to show that they have identical mode shapes. The accuracy could also be seen in error analysis section for a series of ROMs.

Fig. 5.12 – Fig. 5.17 show the relative and absolute errors of computed eigenvalues for these two systems under study. The reference data are from MATLAB and assumed to be "exact". $\sigma_i M$ represents the eigenvalues from MATLAB and $\sigma_i C$ represents the computed eigenvalues based on the B & B method. For one ROM step, i.e. 4062-node system, the accuracy is as good as expected. As shown in Fig. 5.12 and Fig. 5.13, errors of almost all the computed eigenvalues fall into the range around from 10^{-11} to 10^{-13}. The relative errors of lower mode eigenvalues are relative large since they are relatively small in magnitude, while absolute errors plot of Fig.5.13, demonstrates better and stable accuracies. Up to now one concludes that both accuracy and updating speed are desirable for one ROM step. Keen readers might have already observed that the accuracy has deteriorated as more ROMs are conducted, e.g. Fig.5.14 to Fig.5.17 for 4419-node system. The accuracy of the 1st ROM step is not shown since the errors can be checked by later ROM steps. The 1st three ROMs steps are still acceptable as shown in Fig. 5.14 to Fig. 5.16. However, for the last step, i.e. adding one line to the entire 1-2-3-4 system, some of the eigenvalues demonstrated relatively larger errors. The largest errors are up to 10^{-2} to 10^{-3}. The reasons will be discussed briefly in next section.

Note: In Fig.5.13, Fig.5.14, Fig.5.16 and Fig.5.17, the y-axes are in log scales.

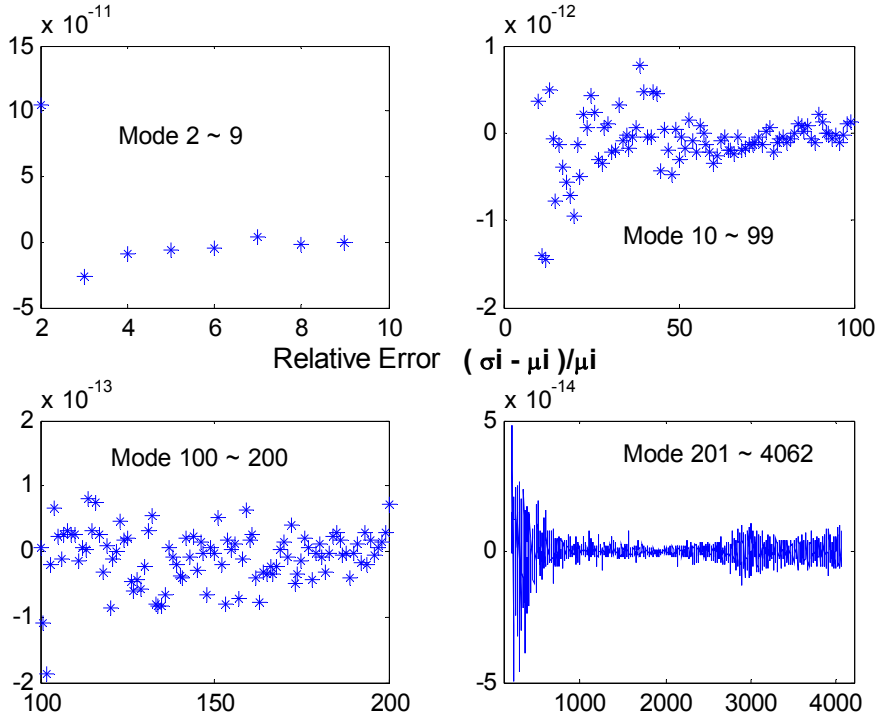

Fig.5.12 Relative errors of computed eigenvalues - 4062-node system

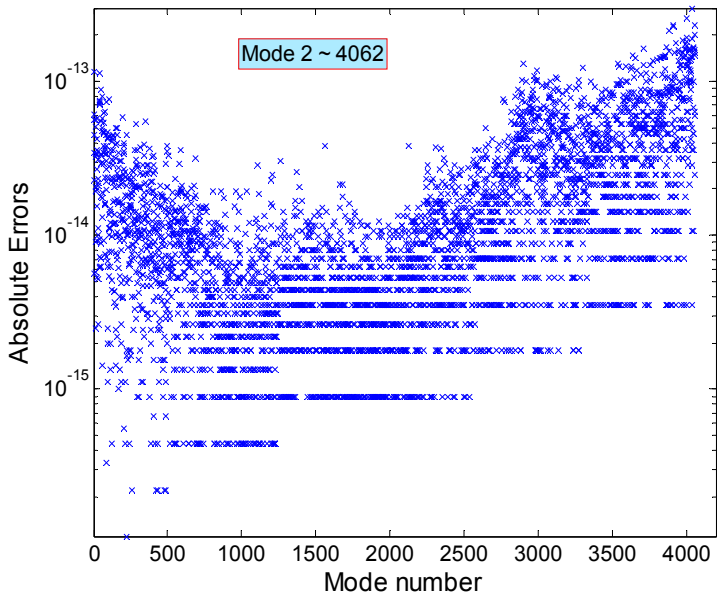

Fig.5.13 Absolute errors of computed eigenvalues - 4062-node system

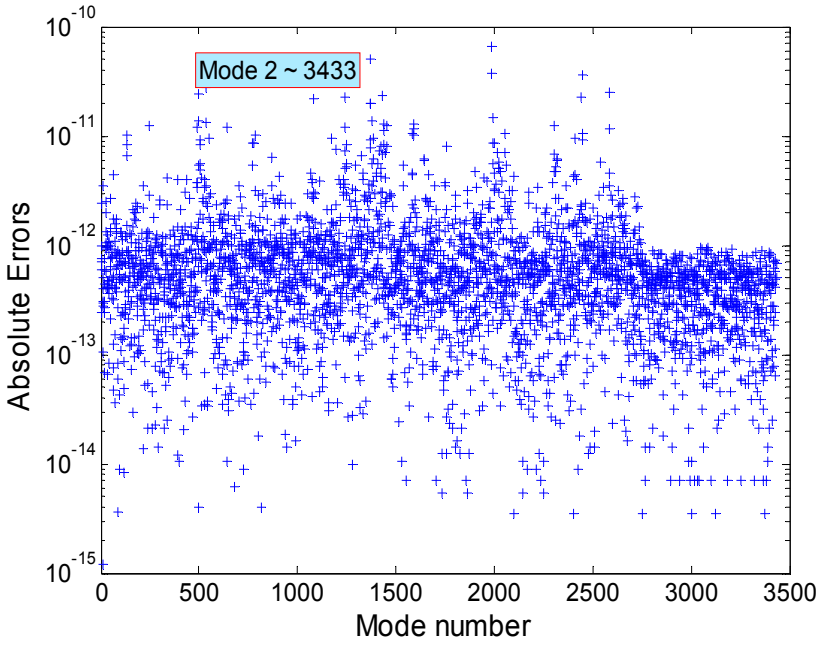

Fig.5.14 Absolute errors of eigenvalues - 4419-node system 1, 2 - 3

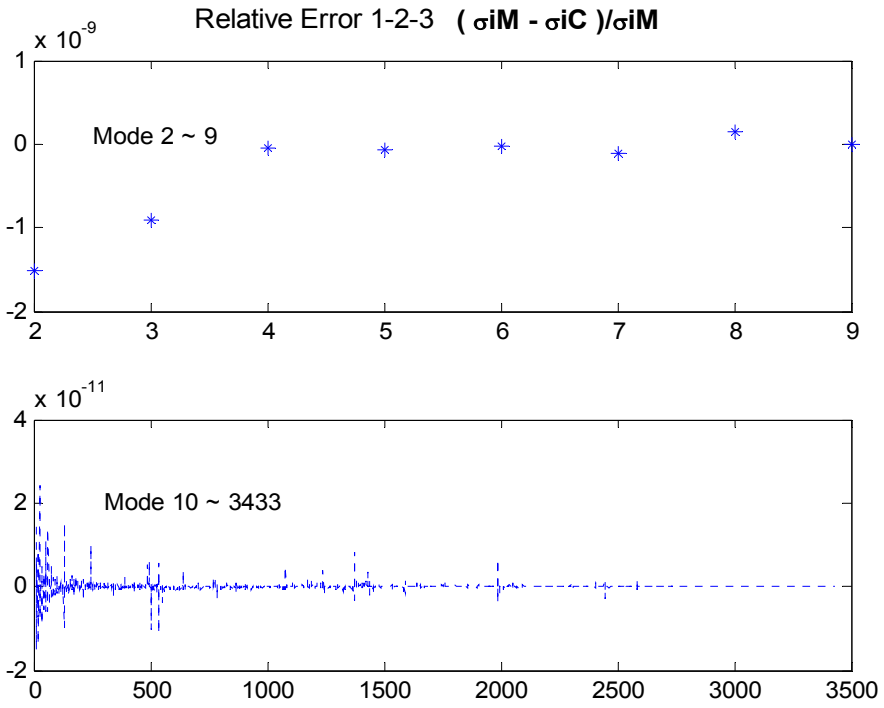

Fig.5.15 Relative errors of eigenvalues - 4419-node system 1, 2 - 3

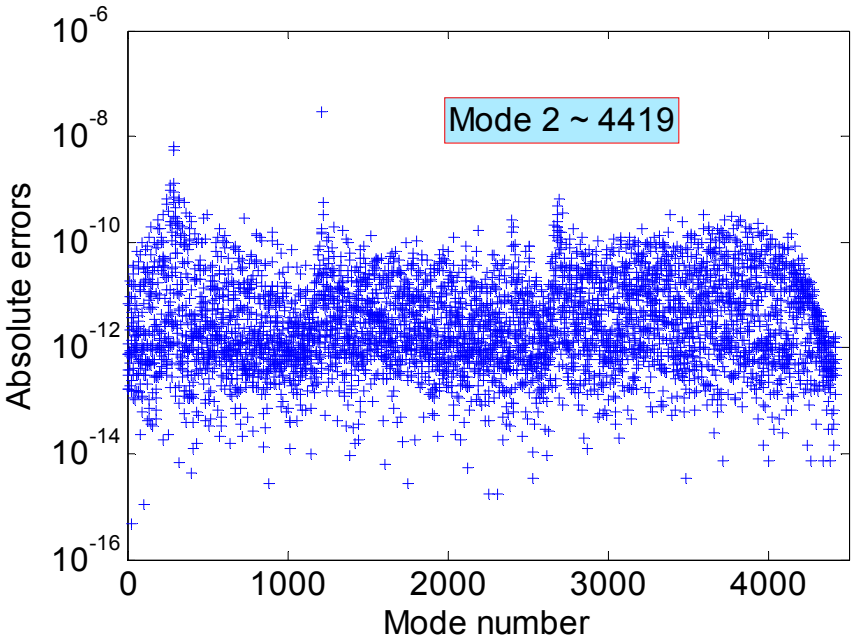

Fig.5.16 Absolute errors of eigenvalues – 4419-node system 1, 2, 3-4

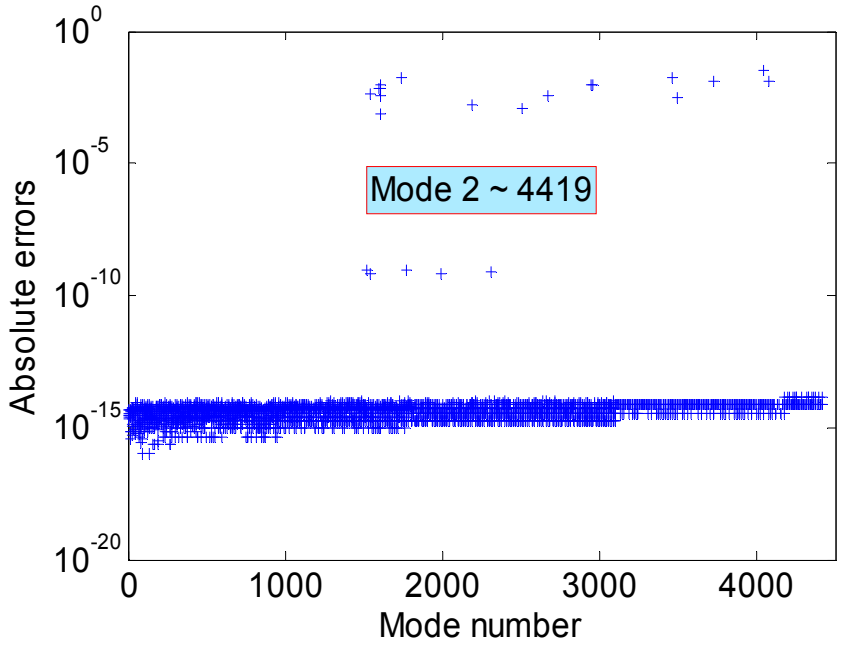

Fig.5.17 absolute errors of eigenvalues – 4419-node system 1-2-3-4-1

5.4 Error Analysis

Many discussions have been given regarding the sensitivity of the computed eigenvectors [11], the boundness for computed eigenvalues and disorthogonality of eigenvectors [18]. This section will conduct some preliminary error analyses

for the computed results and related explanations will be given based on the author's understanding.

5.4.1 Results Interpretation

According to the author's knowledge and the research experience, the deviation between the computed results based on (3.2.8) & (3.2.11) and MATLAB results might be due to the following:

- Loss of orthogonality for computed eigenvectors

Recall that (3.2.8) is derived based on the fact that [Φ] is orthogonal, i.e. the eigenvector matrix of $[\psi_0]$ before ROM steps. Therefore to keep the orthogonality of the computed eigenvectors are quite important for further ROM steps. MATLAB employs QR iterative method to compute eigensystems and it guarantees the orthogonality of the resulting eigenvectors since the eigenvectors matrix is formed by the product of many rotation matrices, each is orthogonal. One can assume that the computed eigensystems from MATLAB are "accurate" since it is well-known that QR-type methods are numerically stable, i.e. the results from these methods are the exact solutions of a matrix with perturbations. The proposed method, on the other hand, does not ensure that the computed eigenvector matrices are orthogonal unless for some serious situation, e.g. poorly separated eigenvalues present, a special treatment, e.g. modified Gram Schmidt method (MGS) is used to maintain the orthogonality. In these cases, the performance of proposed method will be deteriorated and parallel computing might not be as effective as expected. These analyses are out of the scope of this project.

- Cancellation errors

As many researchers in numerical field know that cancellation errors might occur when subtracting two very close floating point numbers (both have the same sign). It can be shown that, the error for the computed results is inverse proportional to the difference between these two numbers.

For example, one wants to compute $a-b$. Define $flop(a-b)=(a-b)(1+\delta)$, where $flop$ represents the floating point operation in computers; δ is the relative

error of the computed result and $(a-b)$ is the exact solution. Then one has $\delta \propto \frac{1}{(a-b)}$. In other words, if $a \approx b$ is the case, the resulting relative error could be extremely large. This might be another reason that leads to inaccuracy of the computed eigenvalues since numerators in (3.2.8) and (3.2.11) are obtained by subtracting two floating point numbers.

5.4.2 Orthogonality Study

In order to determine the orthogonality of the computed eigenvectors, the relative matrix residues have been computed for several cases.

Define $RMR = \frac{\left\| \Phi^T \Phi - I \right\|_2}{\left\| I \right\|_2} = \left\| \Phi^T \Phi - I \right\|_2$ since $\left\| I \right\|_2 = 1$, where RMR represents the relative matrix residue of the orthogonal matrix $[\Phi]$. Mathematically, $RMR = 0$ since $[\Phi]$ is orthogonal. However in practice the orthogonality of $[\Phi]$ may lose due to round-off errors after a series of ROMs. The following relative matrix residues have been computed for the purpose of study. Define,

RMR_O - RMR of original eigenvectors matrix from MATLAB, i.e. before any ROM steps. RMR_i - RMR of the i^{th} ROM step. The computed RMRs are shown in Fig. 5.18.

4062-node system – 1 ROM step

$RMR_O = 2.5295\text{e-}014$ and $RMR_1 = 7.6455\text{e-}010$

4419-node system – 4 ROM steps

$RMR_O = 2.0770\text{e-}014$,

$RMR_1 = 8.8037\text{e-}010$, $RMR_2 = 2.4694\text{e-}007$,

$RMR_3 = 2.5521\text{e-}007$, $RMR_4 = 1$.

Fig.5.18 Computed RMRs of two systems

One concludes the following based on the observation of the results:

- The orthogonality of the eignevectors from MATLAB is always good and RMR_o falls into the order of 10^{-14}, i.e. the eigenvectors from MATLAB are always orthogonal.
- RMR_1, i.e. RMR of one step ROM is also good. And the results of 4062-node system have good accuracy as expected.
- Accuracies are getting worse as more ROMs occur. Loss of orthogonality might be one of the reasons.
- For the 4419-node system with the testing data illustrated, the results of further ROMs might not be good since RMR_4=1. The orthogonality of the resulting eigenvectors is totally lost. No further testing results are available for more ROM steps.

On the other hand, the vector residue $\|\psi_0 q_i - \lambda q_i\|_2$ of 4062-system has been computed too as shown in Fig.5.19. The results are as good as expected. The residues are below 10^{-10}

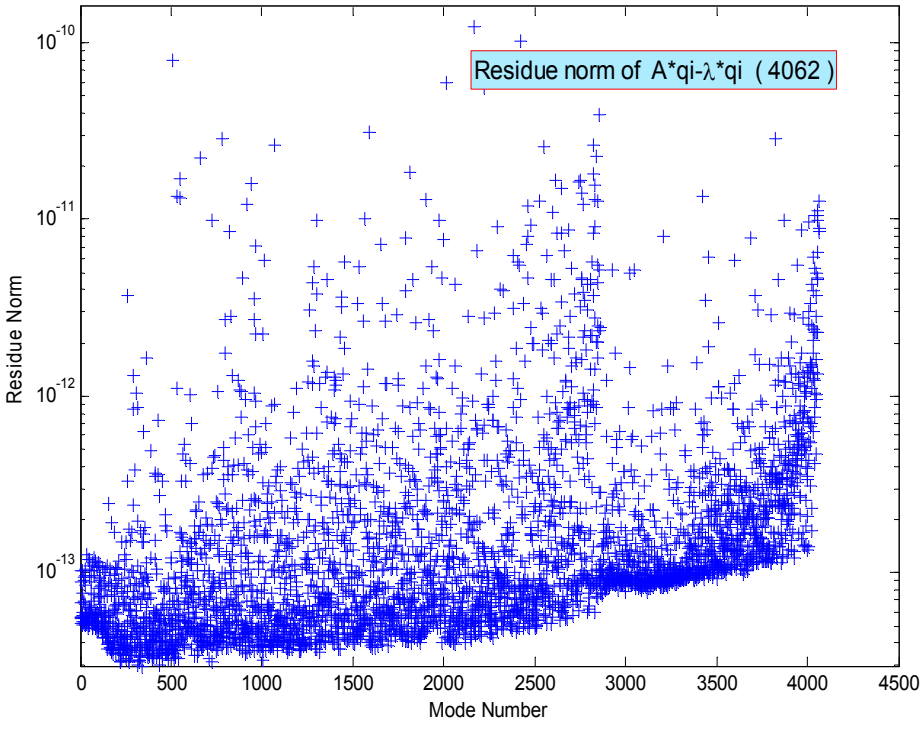

Fig.5.19 Residue norm-2 of computed eigenvectors - 4062-node system

5.4.3 Robustness of the Designed Algorithms

In the algorithms design discussed in Chapter 4, bisection method was adopted instead of other approaches [11, 18]. The rationale for choosing bisection method is that it has stable convergence and guarantees accuracies. Theoretically it can be shown that the algorithms for computing eigenpairs based on the proposed method are numerically stable, i.e. the computed eigensystems are the "exact" solutions of a "nearby" problem - a matrix with small perturbations. Here below is a general analysis based on the testing results in Table 5-11 and 5-12.

Table 5-11: Computed function values and iterations of 4 ROM steps

# of ROM	Max Val	Min Val	Avg. Val	Max iter.	Min iter.	Avg. iter.
1-2	6.15e-8	0	3.08e-11	53	36	44.3
1,2-3	5.69e-3	4.44e-16	2.39e-6	53	33	43.8
1,2,4-4	6.07e-3	2.22e-16	1.24e-6	53	32	43.6
1-2-3-4-1	4.17e-3	3.55e-15	8.33e-7	51	35	43.7

Note: In Table 5-11, the function values are in scientific formats.

Max. Val. – Maximum absolute computed function values of (3.2.8) based on the eigenvalue
Min. Val. – Minimum absolute computed function values of (3.2.8) based on the eigenvalue
Avg. Val. – Average of absolute computed function values of (3.2.8) for all the iterations

Max. iter. – Maximum iterations to find roots of (3.2.8)
Min. iter. – Minimum iterations to find roots of (3.22.8)
Avg. iter. – Average iterations to find roots of (3.2.8)

Fig.5.20 Notations of Table 5-11

As one can see from Table 5-11 that the final function values of (3.5.1) for all the eigenvalues fall into relatively small range as expected. Based on simple observation of the results, the computed eigenvalues corresponding to the maximum values in Table 5-11 do not fall into the categories of eigenvalues which illustrate relatively large errors as shown in Fig. 5.17. One concludes the following,

1) Although some of the function values of (3.5.1) in Table 5-11 exhibit relatively large errors, they do not affect the accuracies of the computed eigenvalues. In other words, the final function values have no direct relationship with the accuracies of the corresponding eigenvalues unless extremely large function values present.
2) The designed algorithms converge to all the cases under study for eigenvalues evaluation. In other words, bisection method as well as its stopping criterion is a good choice.
3) The iteration times are reasonable since the pre-defined error bound is equal to 10^{-16}.
4) The designed algorithms are numerically stable based on testing.

Furthermore, in order to investigate the deviation behaviour of Fig. 5.17, the following test has been conducted as shown in Table 5-12.

Table 5-12: Computed function values for "errored" eigenvalues

Index	1543	1592	1605	1607	1608	1611	1743	2194	2507
Value	2.51e-7	6.05e-6	3.95e-7	9.17e-6	1.22e-4	1.15e-5	8.53e-7	2.94e-6	8.09e-6
Index	2673	2952	2958	3466	3497	3728	4042	4078	X
Value	3.74e-5	2.87e-3	1.03e-6	4.17e-3	2.58e-4	7.37e-6	2.22e-6	1.93e-6	X

Note x's in Table 5-12 represent the data unavailable.

Notation of Table 5-12:

Index: The indices of "errored" eigenvalues in Fig.17. The criterion for selecting these eigenvalues is that the deviation is greater than 10^{-9} when comparing to MATLAB results.

Value: The final function values of (3.5.1) computed in the last iteration of each eigenvalue.

From Table 5-12 one draws the conclusion that the final function value of (3.5.1) corresponding to the "errored" eigenvalues in Fig. 5.17 are relatively small and acceptable. Most of the function values in Table 5-12 fall into the range of

average values in Table 5-11. This is a further proof of arguments 2) and 4) above.

Table 5-13 shows the corresponding "errored" eigenvalues. Although some of the eigenvalues lost accuracies, the "inaccuracies" would not affect the accuracies for further ROMs based on the author's understanding.

Table 5-13: Comparison of MATLAB and B & B for "errored" eigenvalues

Index	1543	1592	1605	1607	1608	1611	1743	2194	2507
B & B	6.3859	6.6704	6.7640	6.7732	6.7740	6.7976	7.6012	10.067	11.748
MATLAB	6.3901	6.6778	6.7678	6.7740	6.7838	6.8049	7.5843	10.068	11.749
Index	2673	2952	2958	3466	3497	3728	4042	4078	X
B & B	12.878	14.813	14.847	20.432	20.733	23.952	29.165	29.778	X
MATLAB	12.882	14.822	14.856	20.449	20.736	23.964	29.201	29.792	X

Note x's in Table 5-13 represent the data unavailable.

Note that all the analyses above are based on the assumption that results from MATLAB are "accurate".

In addition, based on the observation of MATLAB results, the author noticed that there are around 22 eigenvalues from system 1-2-3-4 are quite close to those from the system 1-2-3-4-1. In other words, some of the eigenvalues from the 3rd ROM, which are also inputs to the 4th ROM, are quite close to the corresponding eigenvalues obtained from the 4th ROM. The differences of these eigenvalues fall into the range of $10^{-9} \sim 10^{-8}$. Namely, some "poorly separated" eigenvalues do present in this test for the 4th ROM. The corresponding eigenvalues and indices are not shown. With above analysis, another interesting test was conducted as shown in Table 5-14.

Table 5-14: Function values of (3.5.1) based on MATLAB results for the 4th ROM

# of ROM	**Max Val**	**Min Val**	**Avg. Val**
4th ROM	4.677e+6	1.66e-15	2.083e+3

The notations of Table 5-14 are the same as those in Table 5-11. The function values in Table 5-14 are computed based on (3.5.1) using MATLAB results, i.e. the computed eigensystem of the 3rd ROM based on MATLAB **eig** function on a PC. From this testing, one concludes that although the eigenvectors matrix from MATLAB is orthogogal, the computed function values of (3.5.1) present large errors due to cancellations.

In summary, a preliminary error analysis has been conducted based on the testing results. One concludes that,

- The designed algorithms are efficient, accurate and robust. If all the eigenvalues before and after ROMs are fairly separated (which is the usual case in reality), it will take many updates to have cumulative errors.
- Cancellation errors and loss of orthogonality for computed eigenvectors are the root causes for errors when comparing with MATLAB results.

Chapter 6 Closing Remarks

6.1 Summary

This thesis has pioneered parallel computing on the entire eigensystem of state matrices for large interconnected power grids. Eigensystem studies are important in assessing the small signal stability of power grids.

The starting point of the thesis is the B & B method which has been claimed to have features that enable the eigensystems of stiff matrices to be computed in a parallel way. Therefore the entire eigensystem of non-symmetric matrix [A] is obtainable. It is left to the author to demonstrate that the claims are realizable with super-computers. To accomplish the objectives, the author has to: (a) master programming using MPI in #C; (b) propose fast and robust algorithms (four have been proposed); (c) evaluate the performance using speed-up ratio and accuracy as criteria. The research of this thesis has shown that the proposed B & B method indeed upset traditional ways to dealing with small signal stability problems, which usually focus on calculations of selective or critical eigenpairs for large power grids. The author believes that the findings of this project will deepen the research in small signal stability study based on parallel computing and make on-line monitoring possible. The research results will definitely lead to extensive applications in future power systems operations and analysis.

6.2 Conclusion

Based on the research results, the following main conclusions can be drawn out of this thesis:

1. Section 5.1 shows that the eigensystem of a power grid composed of 4062 generator nodes can be updated within 1.5 seconds. The update consists of connecting or disconnecting a transmission line between two nodes. (The 1.5 seconds is based on 72 processors implementation. For larger system size and with more processors, the speed-up can be increased further.) The computation speed shows that real-time small-signal stability monitoring and

wide-area control (WAC) are within the reach of power system control engineers.

2. The B & B method is not only computation intensive but each new update will use the computed results of the preceding computation. Although thorough error analysis is beyond the scope of this thesis, one notes, in Fig. 5.13 for example, that the absolute errors after the1st ROM fall into the range of 10^{-13}. Therefore, it will take many updates for the errors to accumulate to affect the last 3 significant figures used by engineers. This preliminary assessment indicates that although eigenpairs may lose accuracies in some cases, it will take many updates before the data bank needs to be refreshed.
3. Section 5.2 uses the 4419-node system in Fig. 5.7 to show how the eigensystem can be updated for a series of ROMs. The large system can be divided into a number of relatively small sub-networks. And the eigensystem of each sub-network can be solved by QR method simultaneously by parallel processing. The developed MPI source codes in Appendix I could be used to partition the entire processors set if necessary. Thereafter, the sub-networks can be integrated into the manner described. The research has demonstrated that integrating 4 sub-networks in the 4419-node example only requires 8 seconds.

In demonstrating that: (i) fast eigensystem updates and (ii) a series of ROMs is achievable, the research is bringing real-time monitoring of small signal stability and on-line control nearly to truth. The objectives of fast computation and high accuracy have been made possible by discovering the characteristics of both the B & B method and MPI protocols through trial and error. The discoveries are listed here as contributions:

- The computation of the eigenvector can be sped up by a factor of 3 to 4 when the "unrolling For loops" technique is incorporated, which has broken through the bottlenecking effect of larger systems computation.
- Many of operations required by the B & B method, such as "sort" and "normalization" take negligible time compared to the total update.

- The bisection method has been adopted as it yields the desired accuracy with acceptable number of iterations, although it has slower convergence compared to Newton-like methods.
- The developed algorithm has shown that speedup is not only limited by the ratio of parallelism portion but also by the time on executing MPI routines. By reducing such function calls, it has been able to use as many as 80 processors effectively compared to that of algorithm #1 which is limited to 16 processors implementation.
- The thesis has presented a detailed timing analysis of the developed algorithms as well as MPI parallel codes to guide future research.
- In Chapter 2, the B & B method demonstrates that computing eigensystem of the non-symmetric matrix [A] can be achieved by a similar transformation to the symmetric matrix $[\psi_0]$.

6.3 Future Research

Possible areas for future research include:

1. Develop reliable and effective methods to retain orthogonality as much as possible of the computed eigenvectors.
2. Use other protocols, e.g. MPI-II and OpenMP, instead of MPI-I, to illustrate if the performance could be improved.
3. Find effective ways to dealing with multiple eigenvalues.
4. Discover possibility of rank-two modifications (RTM) update that could be implemented efficiently in a parallel way. Some mathematical models have already been proposed [25].

References

[1] M. Crow, *"Computational methods for electric power systems*", CRC Press, LLC, London, 2003.

[2] G. H. Golub, C.F. Van Loan, *"Matrix Computations"*, Johns Hopkins University Press, Baltimore, 1996.

[3] A.G. Phadke, "Synchronized phasor measurements in power systems", IEEE CAP, vol. 6, Apr. 1993, pp.10–15.

[4] I. Kamwa, R. Grondin, Y. Hebert, "Wide-area measurement based stabilizing control of large power systems – a decentralized/hierarchical approach", IEEE Trans. On Power Systems, vol. 16, Issue 1, Feb. 2001, pp.136–153.

[5] Z. Huang, J. Nieplocha; "Transforming power grid operations via high performance computing", IEEE Power and Energy Society General Meeting, July 2008, pp. 1– 8.

[6] G. Angelidis, A. Semiyen, "Improved methodologies for the calculation of critical eigenvalues in small signal stability analysis", IEEE Trans. On Power Systems, vol. 11, No.3, Aug. 1996, pp. 1209-1217.

[7] J. Campagnolo, N. Martins, J. Periera, L. Lima, H. Pinto, D. Falcao, "Fast small-signal stability Assessment using parallel processing", IEEE Trans. On Power Systems, vol. 9, No.2, May. 1994, pp. 949-956.

[8] G. Angelidis, A. Semiyen, D. Falcao, "Efficient calculation of critical eigenvalue clusters in the small signal stability analysis of large power system", IEEE Trans. On Power Systems, vol. 10, No.1, Feb. 1995, pp. 427-432.

[9] J. Campagnolo, N. Martins, D. Falcao, "An efficient and robust eigenvalue method for small-signal stability assessment in parallel computers", IEEE Trans. On Power Systems, vol. 10, No.1, Feb. 1995, pp. 506-511.

[10] H. M. Banakar, "On computing eigensystems of large interconnected power grid", unpublished.

[11] J. R. Bunch, C.P. Nielsen, D. C. Sorensen, "Rank-one modification of the symmetric eigenproblem", Numer. Math. 31, 1978, pp. 31-48.

[12] W. Gropp, "*Using MPI*", MIT Press, London, 1999.

[13] W. Gropp, "*Using MPI-2*", MIT Press, London, 1999.

[14] G. E. Karniadakis, "*Parallel scientific computing in C++ and MPI*", Cambridge University Press, Cambridge, 2003.

[15] Available: http://www.mpi-forum.org

[16] D. Watkins, "*Fundamentals of matrix computations*", John Wiley, 2002.

[17] G.H. Golub, "Some modified eigenvalue problems", SIAM. R., vol. 15, No.2, Apr.1973, pp. 318-334.

[18] J. J. M. Cuppen, "A divide and conquer method for the symmetric tridiagonal eigenproblem", Numer. Math. 36, 1981, pp. 177-195.

[19] S.C. Chapra, "*Applied numerical methods with MATLAB for engineers and scientists*", McGraw-Hill, 2008.

[20] A.Grama, A. Gupta, G. Karypis, V. Kumar, "*Introduction to parallel computing*", the Benjamin/Cummings Publishing Company, 1994.

[21] P. Tsigas, Y. Zhang, "A simple, fast parallel implementation of quickset and its performance evaluation on SUN enterprise 10,000", 11th Euromicro conference on parallel, Feb 2007.

[22] P. Heidelberger, A. Norton, J.T. Robinson "Parallel quicksort using fetch-and-add", IEEE Trans. On computers, vol. 39, Issue 1, Jan 1990, pp.133-138.

[23] Available: http://leto.net/docs/C-optimization.php

[24] E.J. Kontoghiorghes, "Handbook of parallel computing and statistics", Chapman & Hall/CRC, 2006

[25] K. Gates, "A rank-two divide and conquer method for the symmetric tridiagonal eigenproblem", available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00234887

Appendix A. System Parameters of Super-computers

The developed MPI #C codes were executed on Mammouth Series II and Krylov respectively. Based on the testing experience, Mammouth Series II has a bit faster speed. The related parameters of these two machines are shown below.

Server Name: Mammouth-Série II

Type: Serial Cluster

Nodes: 308 (SGI XE320)

Processors: 616 Intel Xeon quad-cores, 2.8 GHz

Memory: 32 and 16 GB / node

Interconnect: Infiniband in blocks

Peak performance (measured): 21,600 Gflops

Fig.A-1 System parameters of Mammouth Series II - RQCHP

Server Name: Krylov

System: SUN Fire X 4100

Processor Type: 12 dual socket, dual core opteron 275

Clock Frequency: 2,200 MHz

RAM: 8GB

Local Disk: 80 GB SATA

Network Interfaces: 2 Gigabit interface & 1 infiniband SDR interface

Operating System: CentOS 5.2

Fig.A-2 System parameters of Krylov - CLUMEQ

Appendix B. Software Structure of MAIN Function

Fig.B-1 shows the MAIN function structure discussed in section 4.4.1.

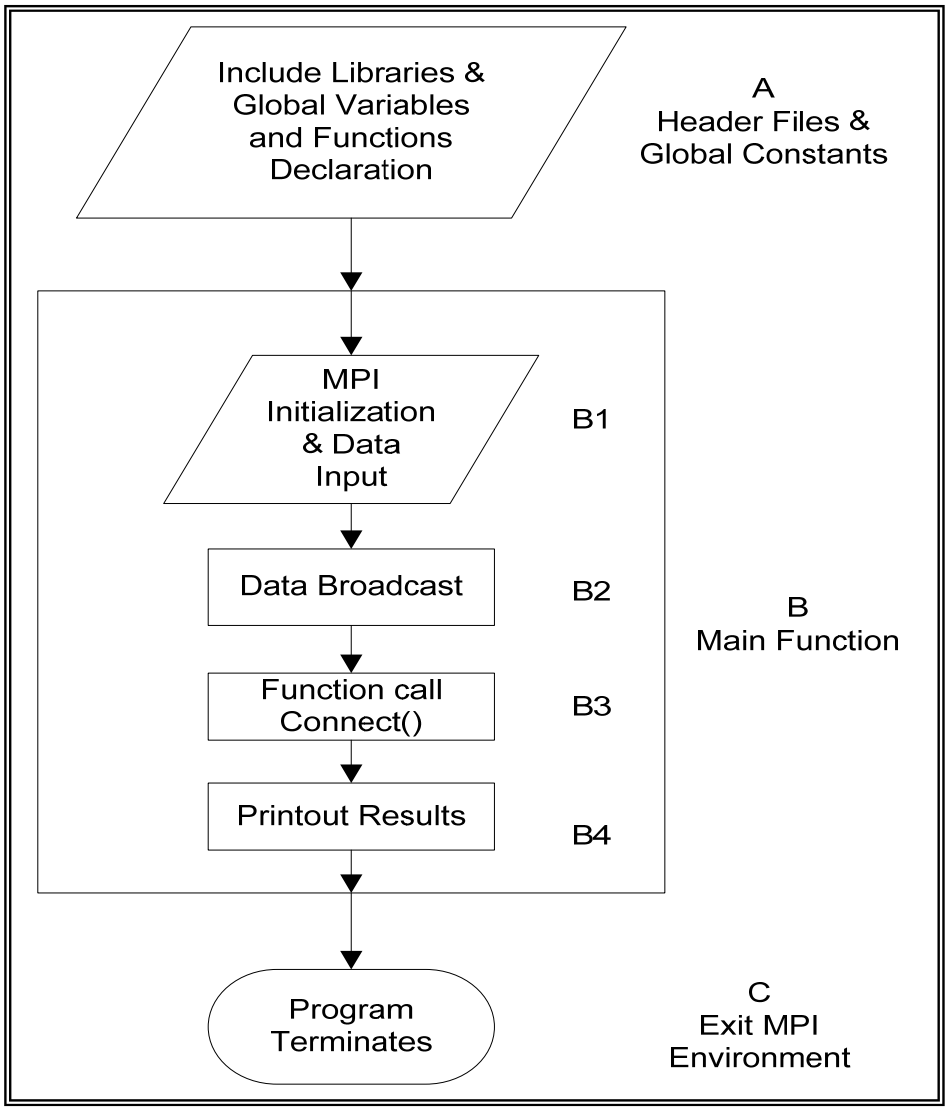

Fig.B-1 Software structure of MAIN function

Appendix C. Software Structure of Algorithm #1

Fig.C-1 shows the software structure of algorithm #1 discussed in section 4.4.2.1.

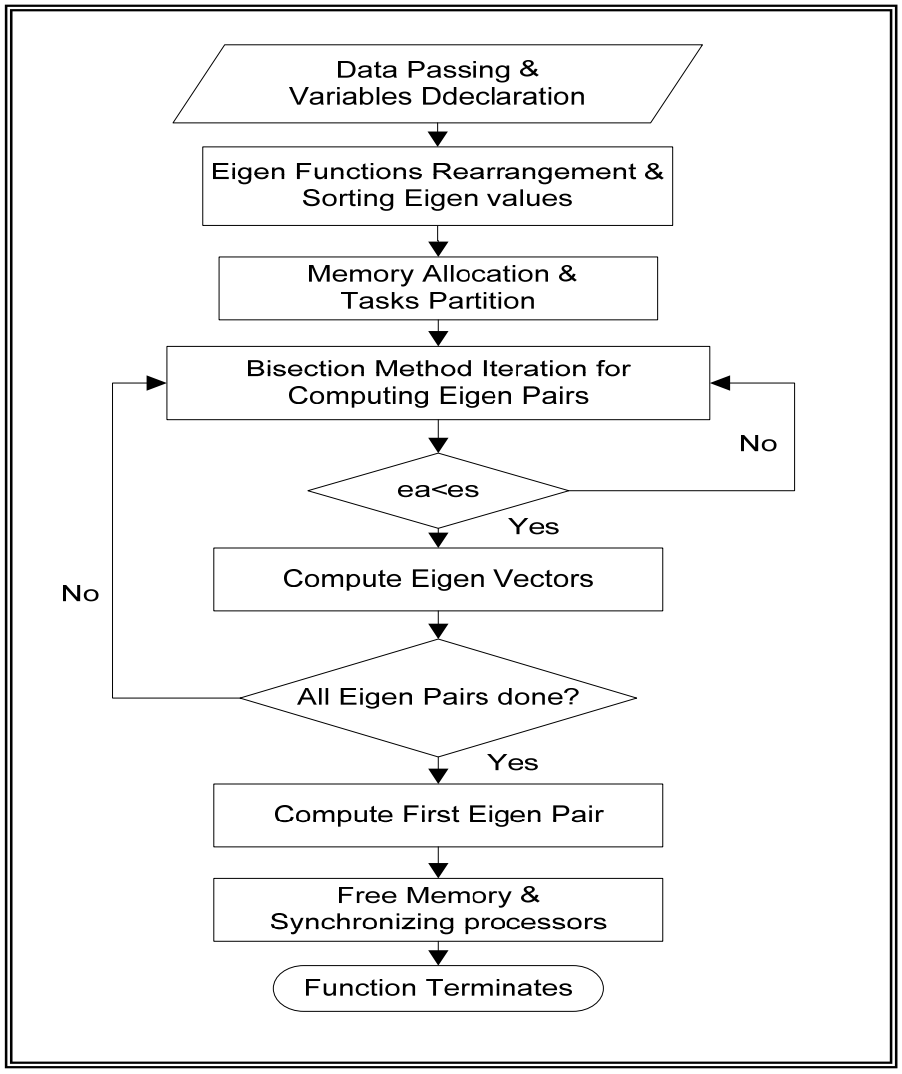

Fig.C-1 Software structure of Connect function of algorithm #1

Appendix D. MPI in #C Source Codes for Algorithm #1- 1 ROM

Appendix D will illustrate MPI #C source code of algorithm #1 discussed in 4.4.2.1. Note that only Connect function is shown below.

```
/*Connect function body */
void Connect ( int ProcessorNum, /* Number of processors used*/
                int Nodes1, /* Number of nodes in system 1*/
                int Nodes2, /* Number of nodes in systems 2*/
                double EigenVectors1[MAX][MAX],/* Eigenvectors matrix of #1*/
                double EigenVectors2[MAX][MAX], /* Eigenvectors matrix of #2*/
                double EigenValues1[MAX],/*Eigenvalues of #1*/
                double EigenValues2[MAX], /* Eigenvalues of #2*/
                int Tie1,/* Tie-in point from #1*/
                int Tie2, /* Tie-in point from #2*/
                double Line, /* Connection line*/
                double EigenVectorFinal[MAX][MAX],/* Computed eigenvectors matrix*/
                double LamdaConn[MAX] /* computed eigenvalues*/
              )
{
   int rank_in_world,k=1,i, j, maxit=100, Nodes, iter, p, q;
   int num,r, offset=0,displs[MAX],sendcnt[MAX], recvcnt;
   int offset1=0, displs1[MAX], sendcnt1[MAX], recvcnt1;
   double EigenVectorDtrans[MAX][MAX],EigenValuesDk[MAX],Lamda[MAX];
   double Lower, Upper, Xr, Xold, ea, test, Sum1,Sum2,Sum3,Sum4;
   double EigenValuesD[MAX], U[MAX],U1[MAX], Mag, FirstVector[MAX];
   double EigenValuesDk1[MAX], es=1e-12;
   double *bufA, *bufB,*VecA,*VecB, *EigenVectorOut, *EigenVectorElement, *temp;
   MPI_Comm_rank(MPI_COMM_WORLD,&rank_in_world);
   Nodes=Nodes1+Nodes2;
/*…..Continue with next text box…*/
```

Fig.D-1 MPI in #C source code of Connect function for algorithm #1 – part (1)

```
/* Place two input eigenvector matrices into the form of (3.3.2)*/
for (i=0;i<Nodes; i++){
      for (j = 0; j < Nodes; j++){
            if ( (i<Nodes1) && (j<Nodes1) )
                  EigenVectorDtrans[i][j] = EigenVectors1[j][i];
            else if( ( i>=Nodes1 ) &&    ( j>=Nodes1 ) )
                  EigenVectorDtrans[i][j] = EigenVectors2[j-Nodes1][i-Nodes1];
            else    EigenVectorDtrans[i][j]=0;
      }
}
/*Place two eigenvalues array into the form of (3.3.2)*/
for(i=0;i<Nodes;i++){
      if(i<Nodes1){
            EigenValuesD[i]=EigenValues1[i];
            EigenValuesDk[i]=EigenValuesD[i];
      }
      else {
            EigenValuesD[i]=EigenValues2[i-Nodes1];
            EigenValuesDk[i]=EigenValuesD[i];
      }

}

/*Sorting entire eigenvalues array*/
quicksort(EigenValuesDk,0,(Nodes-1));
/*Set the upper bound of the largest eigenvalue*/
EigenValuesDk[Nodes]=EigenValuesD[Nodes-1]+2*Line;
/*Determine minimum # of computation tasks each processor do*/
num=Nodes/ProcessorNum;
/*r is the remainder, equally distributed to r processors,
i.e. eigenvalues and rows of eigenvectors*/
r=Nodes%ProcessorNum;
/* …..Continue with next text box…*/
```

Fig.D-2 MPI in #C source code of Connect function for algorithm #1 – part (2)

```
/*Dynamic memory allocation*/
VecA=(double*)malloc(Nodes*sizeof(double));
VecB=(double*)malloc(Nodes*sizeof(double));
EigenVectorOut=(double*)malloc((Nodes)*sizeof(double));
EigenVectorElement=(double*)malloc((num+1)*sizeof(double));
temp=(double*)malloc(Nodes*sizeof(double));
double BufMatrix[num+1][MAX];
bufA=(double*)malloc((num+1)*sizeof(double));
bufB=(double*)malloc((num+1)*sizeof(double));
   if(rank_in_world==0){
        /*Place the ith and jth rows of eigenvector matrix into another sub-matrix for
        later scattering to each processor*/
       for(i=0;i<Nodes;i++){
            VecA[i]=EigenVectorDtrans[i][Tie1];
            VecB[i]=EigenVectorDtrans[i][Tie2];

       }

       /*Tasks partitioning, includes receive and send count, displacement, number
       of elements to scatter etc, for both eigenvalues array and rows of eigenvector matrix */
      /* Tasks partitioning for rank 0 to r-1*/
      for(i=0;i<r;i++){
                  displs[i]=offset;/*displacement for eigenvalues*/
                  displs1[i]=offset1;/* displacement for eigenvectors*/
                  offset+=(num+1);
                  offset1=offset1+(num+1)*MAX;
                  sendcnt[i]=(num+1); /* # of eigenvalues to be scattered to each processor*/
                  /* # of elements of eigenvector matrix to be scattered to each processor*/
                  sendcnt1[i]=(num+1)*MAX;

      }
/*….Continue with next text box*/
```

Fig.D-3 MPI in #C source code of Connect function for algorithm #1 – part (3)

```
/* …..Continue with partitioning…..*/
      for(i=r;i<ProcessorNum;i++){/* Tasks partitioning for rank r – ProcesorNum-1*/
            displs[i]=offset;
            displs1[i]=offset1;
            offset+=num;
            offset1=offset1+num*MAX;
            sendcnt[i]=num;
            sendcnt1[i]=num*MAX;
     }/* Actually the above two partitions could be combined as seen for algorithm #4*/

 }

/*Scatter to each processor the number of tasks it should do */
MPI_Scatter (sendcnt,1,MPI_INT,&recvcnt,1,MPI_INT,0,MPI_COMM_WORLD);
/*Scatter to each processor the number of elements of eigenvectors matrix */
MPI_Scatter(sendcnt1,1,MPI_INT,&recvcnt1,1,MPI_INT,0,MPI_COMM_WORLD);
/*Scatter transposed eigenvector matrix to each processor, each processor might
receive different rows, the difference is one row at most. */
MPI_Scatterv(EigenVectorDtrans,sendcnt1,displs1,MPI_DOUBLE,BufMatrix,recvcnt1,
MPI_DOUBLE,0,MPI_COMM_WORLD);
/*Scatter the ith and jth rows of eigenvectors to each processors */
MPI_Scatterv(VecA,sendcnt,displs,MPI_DOUBLE,bufA,recvcnt,MPI_DOUBLE,0,
MPI_COMM_WORLD);
MPI_Scatterv(VecB,sendcnt,displs,MPI_DOUBLE,bufB,recvcnt,MPI_DOUBLE,0,
MPI_COMM_WORLD);
/*Scatter the eigenvalues array to each processor*/
MPI_Scatterv(EigenValuesD,sendcnt,displs,MPI_DOUBLE,Lamda,recvcnt,
MPI_DOUBLE,0, MPI_COMM_WORLD);
/* Compute constant in( 3.2.8) and (3.2.11)*/
for(i=0;i<recvcnt;i++)
      U1[i]=bufA[i]-bufB[i];
/* …..Continue with next text box…..*/
```

Fig.D-4 MPI in #C source code of Connect function for algorithm #1 – part (4)

```
/*Start to compute eigenvalues based on bisection method, double infinite loops,
one for eigenvalues iteration and the other is for eigenpairs indexing*/
 while(1) {
          Upper=EigenValuesDk[k+1]-1e-14;/* Modified original upper bound*/
          Lower=EigenValuesDk[k]+1e-14;/* Modified original lower bound*/
          Xr=Lower;
          iter=0;
          while(1){
                if(fabs(EigenValuesDk[k]-EigenValuesDk[k+1])<=1e-9) {
                      Xr=(Upper+Lower)/2;
                      break;
                }
                Xold=Xr;
                Xr=(Upper+Lower)/2;
                iter++;
                if(Xr!=0) ea=fabs(Xr-Xold)/Xr;
                Sum1=0; Sum2=0;
          /*Each processor only compute part of the terms of (3.2.8) and sum up*/
          for (i=0;i<recvcnt;i++){
              /*Sum1 is sum evaluated at new lower bound */
              Sum1+=Line*pow(U1[i],2)/(Lamda[i]-Lower);
              /*Sum2 is evaluated at middle point of the new interval*/
              Sum2+=Line*pow(U1[i],2)/(Lamda[i]-Xr);
          }
/* …Continue with next text box….*/
```

Fig.D-5 MPI in #C source code of Connect function for algorithm #1 – part (5)

```
/*Global reduction to get the total of sub-totals above,
i.e. new f value at new lower bound and at new middle point*/
MPI_Allreduce(&Sum1,&Sum3,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
MPI_Allreduce(&Sum2,&Sum4,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
Sum3+=1; Sum4+=1;
test=Sum3*Sum4;/* Bisection criteria*/
if(test<0) Upper=Xr; /* Bisection criteria*/
else if(test>0) Lower=Xr; /* Bisection criteria*/
else ea=0; /* Bisection criteria*/
/*One iteration is done up to this point. If the root is found, then get out of this loop,
i.e. one eigenvalue was found for this sub-interval*/
if((ea<=es)||(iter>maxit)) break;
  }
  LamdaConn[k]=Xr;
/*Start to compute the corresponding eigenvector, i.e. compute (3.2.11). Again, each
processor only compute partial sum of (3.2.11)*/
Mag=0;
for (p=0;p<Nodes;p++)
      temp[p]=0;
      for (p=0;p<recvcnt;p++)
            for(q=0;q<Nodes;q++)
                  temp[q]+=U1[p]/(Lamda[p]-LamdaConn[k])*BufMatrix[p][q];
/*Global reduction for the sub-total eigenvectors, this time MPI_Allreduce
manipulates on vectors*/
MPI_Allreduce(temp10,EigenVectorOut, Nodes, MPI_DOUBLE,
MPI_SUM, MPI_COMM_WORLD);
/*….. Continue with next text box….*/
```

Fig.D-6 MPI in #C source code of Connect function for algorithm #1 – part (6)

```
   /*Normalization of the computed eigenvector*/
   for(p=0;p<Nodes;p++)
         Mag+=pow(EigenVectorOut[p],2);
   Mag=sqrt(Mag);
   for(i=0;i<Nodes;i++)
         EigenVectorFinal[i][k]=EigenVectorOut[i]/Mag;
   k++;
   /*If all the eigenpairs are done, exit the loop*/
   if (k==Nodes) break;
}
LamdaConn[0]=0.0;
/*Compute first eigenpair*/
double numerator=1.0,e;e=sqrt(Nodes);
for (i=0;i<Nodes;i++)
       EigenVectorFinal[i][0]=e;
        /*Delete memory allocated*/
        free(VecA);
        free(VecB);
        free(bufA);
        free(bufB);
        free(temp);
        free(EigenVectorOut);
        free(EigenVectorElement);
   /*Synchronizing all the processors*/
   MPI_Barrier(MPI_COMM_WORLD);
}/* Connect function ends….*/
/* …..Continue with next text box…*/
```

Fig.D-7 MPI in #C source code of Connect function for algorithm #1 – part (7)

Note that sorting function is not illustrated here. It is in Appendix H.

Appendix E. Software Structure of Algorithm #3 and #4

Fig.E-1 shows the software structure of Connect function of algorithm #3 and #4 discussed in section 4.4.2.2.

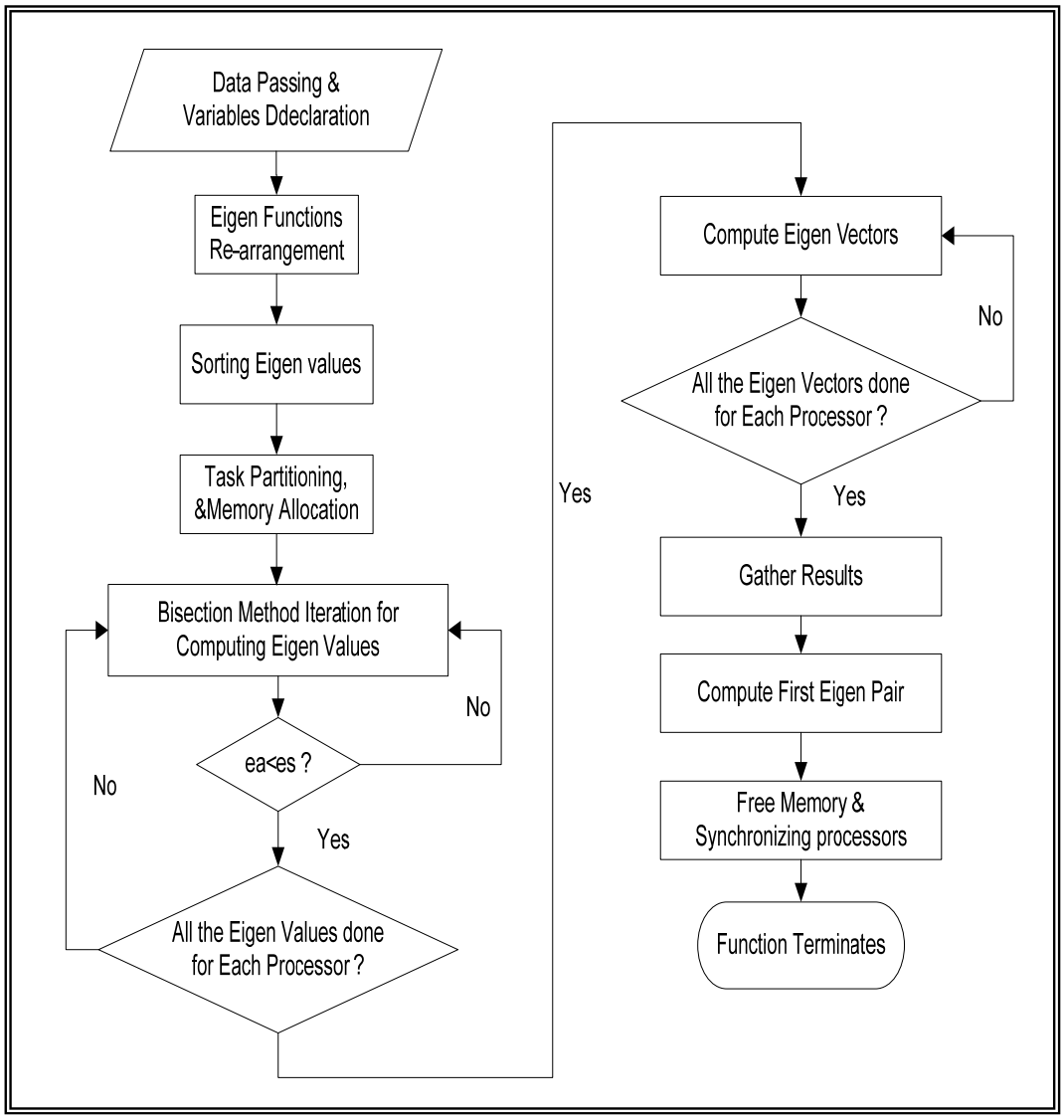

Fig.E-1 Software structure of Connect function of algorithm #3 and #4

Appendix F. MPI in #C Source codes for Algorithm #3 – 1 ROM

Note only Connect function is shown below. Install function is similar to this as discussed in section 4.4.3 and Install function is shown in appendix G.

```
/* Header files, constants and functions declaration*/
#include <math.h>
#include<stdio.h>
#include <stdlib.h>
#include "mpi.h"/* MPI header files*/
#define MAX 4100 /* Max matrix dimension*/
#define TimeDim 20
#define Ti 30/* Tie-in point from #1*/
#define Tj 651 /*Tie-in point from #2*/
#define NewLine 2.03 /*Connection line*/
int nprocs;/* Global variable of # of processors*/
/* Connect function declaration*/
void Connect(int ProcessorNum, /* # of processors*/
                    int Nodes1,/* # of nodes in system 1*/
                    int Nodes2,/* # of nodes in system 2*/
                    double EigenVectors1[MAX][MAX],/*Eigenvector matrix of #1*/
                    double EigenVectors2[MAX][MAX],/*Eigenvector matrix of #2*/
                    double EigenValues1[MAX],/* Eigenvalues of #1*/
                    double EigenValues2[MAX],/*Eigenvalues of #2*/
                    int Tie1,/* Tie-in point of #1*/
                    int Tie2,/*Tie-in point of #2*/
                    double Line,/* Connection line*/
                    double EigenVectorFinal[MAX][MAX],/* Computed eigenvectors*/
                    double LamdaConn[MAX],/* Computed eigenvalues*/
                    double TimeVector[TimeDim]); /* Timing record*/
/* Quicksort declaration*/
void quicksort(double a[],int L,int R);
/* …Continue with next text box….*/
```

Fig.F-1 MPI in #C source code for algorithm #3 -1 ROM – part (1)

```
/* MAIN function starts…*/
int main(int argc, char *argv[]){
      int i,j,Nodes,Nnodes,N1,N2,Times,p,rank,printindex;
      double EigenValueD[MAX],MatEigenValue[MAX];
      double time=0,starttime=0,stoptime=0;
      double LamdaC[MAX],MatlabVal[MAX];
      double EigenVec1[MAX][MAX],EigenVec2[MAX][MAX];
      double EigenVal1[MAX],EigenVal2[MAX], VectorResults1[MAX][MAX];
      double EigenVec1_2[MAX][MAX], EigenVal1_2[MAX],TimeVector[TimeDim];
      double tsinit,teinit,ttinit,tsinput,teinput,ttinput,tsbcast,tebcast,ttbcast;
      FILE *fp1,*fp2,*fp3,*fp4,*fp5,*fp6,*fpt1;
      tsinit=MPI_Wtime();/* Starting time for initialization*/
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
      teinit=MPI_Wtime();/*Ending time for initialization*/
      ttinit=teinit-tsinit;/*Total time for initialization*/
      tsinput=MPI_Wtime();/*Starting time for data input*/
      if(rank= =0){
          fp1 =fopen("Vector1_4062.txt", "r" );/* Eigenvector matrix of #1*/
                fscanf(fp1, "%d\n", &N1);/* Dimension of the matrix*/
                for (i=0;i<N1;i++){
                      for (j=0;j<N1;j++)
                            fscanf(fp1, "%lf", &EigenVec1[i][j]);
                fscanf(fp1,"\n");
                }
          fclose(fp1);
/* …Continue with next text box…*/
```

Fig.F-2 MPI in #C source code for algorithm #3-1 ROM – part (2)

```
/*Continue with data input….*/
fp2 =fopen("Vector2_4062.txt", "r" ); /* Eigenvector matrix of #2*/
fscanf(fp2, "%d\n", &N2);
for (i=0;i<N2;i++){
    for (j=0;j<N2;j++)
        fscanf(fp2, "%lf", &EigenVec2[i][j]);
    fscanf(fp2,"\n");
}
fclose(fp2);
fp3 =fopen("Value1_4062.txt", "r" );/* Eigenvalues of #1*/
for (i=0;i<N1;i++)
    fscanf(fp3, "%lf", &EigenVal1[i]);
fscanf(fp3,"\n");
fclose(fp3);
fp4 =fopen("Value2_4062.txt", "r" );/* Eigenvalues of #2*/
for (i=0;i<N2;i++)
    fscanf(fp4, "%lf", &EigenVal2[i]);
fscanf(fp4,"\n");
fclose(fp4);
fp5 =fopen("ValueT_matlab_4062.txt", "r" ); /* Eigenvalues 1-2 MATLAB*/
for (i=0;i<N1+N2;i++)
    fscanf(fp5, "%lf", &MatlabVal[i]);
fscanf(fp5,"\n");
fclose(fp5);
}
teinput=MPI_Wtime();/* Ending time of data input*/
ttinput=teinput-tsinput;/* Total data input time*/
/* ….Continue with next text box….*/
```

Fig.F-3 MPI in #C source code for algorithm #3-1 ROM – part (3)

```
/* Data broadcast and other*/
tsbcast=MPI_Wtime();/* Starting time for data broadcast*/
MPI_Bcast(&N1,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&N2,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(EigenVec1,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(EigenVec2,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(EigenVec1_2,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(EigenVal1,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(EigenVal2,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(EigenVal1_2,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(MatlabVal,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
tebcast=MPI_Wtime();/*Ending time for data broadcast*/
ttbcast=tebcast-tsbcast;/*Total time for data broadcast*/
starttime=MPI_Wtime();/*Starting time for computation*/
Connect(nprocs,N1,N2,EigenVec1,EigenVec2,EigenVal1,EigenVal2,Ti,Tj,
NewLine,VectorResults1,LamdaC,TimeVector);
MPI_Barrier(MPI_COMM_WORLD);
stoptime=MPI_Wtime();/* Ending time for computation*/
time=stoptime-starttime;/*Total computation time*/
   for (i=0;i<N1+N2;i++){/* Eigenvalues for print out before ROM*/
        if(i<(N1)) EigenValueD[i]=EigenVal1[i];
        else EigenValueD[i]=EigenVal2[i-N1];
   }
   quicksort(EigenValueD,0,(N1+N2-1));/*Sorting eigenvalues*/
   MPI_Barrier(MPI_COMM_WORLD);
/* ….Continue with next text box …..*/
```

Fig.F-4 MPI in #C source code for algorithm #3 -1 ROM– part (4)

```
/* Start to print out results….*/
  if (rank==0){
      Times=(N1+N2)/40;/* Each time print 40 eigenvectors for better view*/
      if ((N1+N2)%40!=0) Times+=1;
      printindex=0;
      fpt1=fopen("EigenVectors_4062.txt","w");
      for(p=0;p<Times;p++){
            for(i=0;i<N1+N2;i++){
                  for(j=printindex;j<printindex+40;j++)
                        fprintf(fpt1," %20.16lf",VectorResults1[j][i]);
                  fprintf(fpt1,"\n");
            }
            printindex+=40;
            fprintf(fpt1,"\n\n");
      }
      fprintf(fpt1,"%s",str);
      fprintf(fpt1,"\n\n");
      fpt1=fopen("EigenValues_4062_Jun_06.txt","w");/*Output printout*/
      fprintf(fpt1,"\n    k           Matlab Lamda_D                                    ");
      fprintf(fpt1,"Matlab Lamda_C                              ");
      fprintf(fpt1,"CLUMEQ Lamda_C\n");
      for(i=0;i<N1+N2;i++){
            fprintf(fpt1,"\n%3d     %20.16lf                    ",i+1,EigenValueD[i]);
            fprintf(fpt1,"%20.16lf                    ",MatlabVal[i]);
            fprintf(fpt1," %20.16lf                ",LamdaC[i]);
            fprintf(fpt1,"\n");
      }
/*…. Continue with next text box…..*/
```

Fig.F-5 MPI in #C source code for algorithm #3-1 ROM – part (5)

```
/*Continue with printout output….*/
fprintf(fpt1,"\nInitializationTime=%lf\n",ttinit); /* Time of initialization*/
fprintf(fpt1,"\nDataInputTime=%lf\n",ttinput); /*Time of data input*/
fprintf(fpt1,"\nBroadcastTime=%lf\n",ttbcast); /*Time of broadcast*/
fprintf(fpt1,"\nComputationTime=%lf\n",time); /* Total computation time*/
fprintf(fpt1,"\nSorTime=%lf\n",TimeVector[0]); /*Sorting time*/
fprintf(fpt1,"\nGatherTime=%lf\n",TimeVector[1]); /* Gather time*/
fprintf(fpt1,"\nScater1Time=%lf\n",TimeVector[2]); /*See chapter 5 all below*/
fprintf(fpt1,"\nScatter2Time=%lf\n",TimeVector[3]);
fprintf(fpt1,"\nReduceTime=%lf\n",TimeVector[4]);
fprintf(fpt1,"\nTotalCommunicationTime=%lf\n",TimeVector[5]);
fprintf(fpt1,"\nEigenValueTime=%lf\n",TimeVector[6]);
fprintf(fpt1,"\nEigenVectorTime=%lf\n",TimeVector[7]);
fprintf(fpt1,"\nOtherTime1=Vec+Val_ Arrangement=%lf\n",TimeVector[8]);
fprintf(fpt1,"\nOtherTime2=displ+Barrier=%lf\n",TimeVector[9]);
fprintf(fpt1,"\nOtherTime3=firstvector=%lf\n",TimeVector[10]);
fprintf(fpt1,"\nOtherTime=%lf\n",TimeVector[11]);
fprintf(fpt1,"\nTotalTime=%lf\n",TimeVector[12]);
fprintf(fpt1,"\n\n");
   }
   MPI_Finalize();
   return 0;
}
/*…….Continue with next text box…*/
```

Fig.F-6 MPI in #C source code for algorithm #3 -1 ROM– part (6)

```
/* Connect function starts…*/
void Connect(int ProcessorNum, int Nodes1,int Nodes2,
                    double EigenVectors1[MAX][MAX],
                    double EigenVectors2[MAX][MAX],
                    double EigenValues1[MAX],double EigenValues2[MAX],
                    int Tie1,int Tie2, double Line,
                    double EigenVectorFinal[MAX][MAX],
                    double LamdaConn[MAX], double TimeVector[TimeDim])
{
    #define unroll 8
    int rank_in_world,i,j,maxit=100,Nodes,p,q, iteration=0, nu=1.0,e,u;
    int num,r,offset=0,displs[MAX],sendcnt[MAX],recvcnt,r1;
    int offset1=0,displs1[MAX],sendcnt1[MAX],recvcnt1;
    double EigenVectorDtrans[MAX][MAX],EigenValuesDk[MAX],*Lamda;
    double Lower,Upper,Xr,Xold,ea,test,Sum1,Sum2, es=1e-12, Sigma,Mag;
    double *Temp[unroll],EigenValuesD[MAX], U[MAX],U1[MAX];
    double *LamdaConnSingle, *bufA, *bufB,*VecA,*VecB;
    double SigmaD=0,SigmaC=0,TotalD=0,TotalC=0;
    double tsscat1,tescat1,ttscat1,tsscat2,tescat2,ttscat2;
    double tsreduce,tereduce,ttreduce,tsgather,tegather,ttgather;
    double tsqs,teqs,ttqs,tsval,teval,ttval,tsvec,tevec,ttvec,double ttc;
    double tsother1,teother1,ttother1,tsother2,teother2,ttother2;
    double tsother3,teother3,ttother3,ttother,tt;
/* Continue with next text box…..*/
```

Fig.F-7 MPI in #C source code for algorithm #3-1 ROM – part (7)

```
/* Continue with Connect…*/
MPI_Comm_rank(MPI_COMM_WORLD,&rank_in_world);
Nodes=Nodes1+Nodes2;/* Total # of nodes of these two systems*/
tsother1=MPI_Wtime();/* Staring time for other 1,see chapter 1*/
for(i=0;i<Nodes;i++){/*Arrange 2 eigenvectors matrix in the form (3.3.2 and traanspose*/
      for (j=0;j<Nodes;j++){
          if ((i<Nodes1)&&(j<Nodes1))
                EigenVectorDtrans[i][j]=EigenVectors1[j][i];
          else if((i>=Nodes1)&&(j>=Nodes1))
                EigenVectorDtrans[i][j]=EigenVectors2[j-Nodes1][i-Nodes1];
          else EigenVectorDtrans[i][j]=0;
       }
 }
 for (i=0;i<Nodes;i++){/* Eigenvalues in the form for (3.3.2)
      if(i<Nodes1){
           EigenValuesD[i]=EigenValues1[i];
           EigenValuesDk[i]=EigenValuesD[i];
       }
       else{
           EigenValuesD[i]=EigenValues2[i-Nodes1];
           EigenValuesDk[i]=EigenValuesD[i];
      }
}
teother1=MPI_Wtime();/* Ending time for other 1*/
ttother1=teother1-tsother1;/* Total time for other 1*/
/* Continue with next text box….*/
```

Fig.F-8 MPI in #C source code for algorithm #3-1 ROM – part (8)

```
 /* Continue with Connect…*/
tsqs=MPI_Wtime();/* Start time for sorting*/
quicksort(EigenValuesDk,0,(Nodes-1));/* Quick sorting*/
teqs=MPI_Wtime();/* Ending time for sorting*/
ttqs=teqs-tsqs;/* Total time for sorting*/
tsother2=MPI_Wtime();/* Starttingtime for other 2*/
num=Nodes/ProcessorNum;/* See algorithm #1*/
r=Nodes%ProcessorNum; /* See algorithm #1*/
for (i=0;i<unroll;i++)/* Memory allocation*/
        Temp[i]=(double*)malloc((Nodes)*sizeof(double));
if (rank_in_world==0){/* Preserve the same notions as those in #1*/
     for(i=0;i<r;i++){
           displs[i]=offset;
           displs1[i]=offset1;
           offset+=(num+1);
           offset1=offset1+(num+1)*MAX;
           sendcnt[i]=(num+1);
           sendcnt1[i]=(num+1)*MAX;
      }
      for(i=r;i<ProcessorNum;i++){
            displs[i]=offset;
            displs1[i]=offset1;
            offset+=num;
            offset1=offset1+num*MAX;
            sendcnt[i]=num;
            sendcnt1[i]=num*MAX;
       }
}
teother2=MPI_Wtime();/* Endig time for other 2*/
ttother2=teother2-tsother2;/* Total time for other 2*/
tsscat1=MPI_Wtime();/* Starting time for scatter 1, see chapter 5*/
/* This scatter, uniform scatter, each receives one integer indicates # of tasks it should do*/
MPI_Scatter(sendcnt,1,MPI_INT,&recvcnt,1,MPI_INT,0,MPI_COMM_WORLD);
tescat1=MPI_Wtime(); ttscat1=tescat1-tsscat1;
/* Continue with next textbox…*/
```

Fig.F-9 MPI in #C source code for algorithm #3-1 ROM – part (9)

```
/* Continue with Connect……*/
/*Memory allocation for sub-array of each processor*/
Lamda=(double*)malloc((recvcnt+1)*sizeof(double));
/* Memory allocation for computed sub-array of eigenvalues of each processor*/
LamdaConnSingle=(double*)malloc((recvcnt)*sizeof(double));
/* Decalration for computed sub-matrices of eigenvectors for each processor*/
double EigenVectorSingle[recvcnt][MAX];
if (rank_in_world<r)/* Upper bound for largest eigenvalue in each sub array*/
   Lamda[recvcnt]=EigenValuesDk[(rank_in_world+1)*recvcnt];
else if ((rank_in_world>=r)&&(rank_in_world<ProcessorNum-1))
   Lamda[recvcnt]=EigenValuesDk[(rank_in_world+1)*recvcnt+r];
r1=recvcnt%unroll;/* number for of leftover eigenvectors*/
for(p=0;p<Nodes;p++){/* Same as in algorithm #1, constant terms in (3.2.8)*/
      U1[p]=EigenVectorDtrans[p][Tie1]-EigenVectorDtrans[p][Tie2];
      U[p]=Line*pow(U1[p],2);
}
tsscat2=MPI_Wtime();
/* Scatter sorted eigenvalues array to each processor*/
MPI_Scatterv(EigenValuesDk,sendcnt,displs,MPI_DOUBLE,Lamda,recvcnt,
MPI_DOUBLE,0,MPI_COMM_WORLD);
tescat2=MPI_Wtime(); ttscat2=tescat2-tsscat2;
/* Largest eigenvalue of the entire set is dealt separately. Actually, its upper bound
exists as in algorithm #1, here just to prove (3.6.2)*/
if (rank_in_world= =ProcessorNum-1) recvcnt-=1;
/*…..Continue with next text box…*/
```

Fig.F-10 MPI in #C source code for algorithm #3 -1 ROM– part (10)

```
/* Start to compute eigenvalues…*/
tsval=MPI_Wtime();/* Starting time of computing eigenvalues*/
for (i=0;i<recvcnt;i++){/* Each processor compute 'recvcnt' eigenvalues*/
      Upper=Lamda[i+1]-1e-14;/* Modified original upper bound*/
      Lower=Lamda[i]+1e-14;/* Modified original lower bound*/
      SigmaD+=Lower;/* Compute old eigenvalues sub-total*/
      Xr=Lower;
      ea=10;/* Predefined error*/
      while(ea>=es){
            if(fabs(Lamda[i]-Lamda[i+1])<=1e-9){
                  Xr=(Upper+Lower)/2; /* Same as in #1*/
                  break;
             }
             Xold=Xr;
             Xr=(Upper+Lower)/2;
             if(Xr!=0) ea=fabs(Xr-Xold);
             Sum1=0; Sum2=0;
             for(p=0;p<Nodes;p++){
                   Sum1+=U[p]/(EigenValuesD[p]-Lower);
                   Sum2+=U[p]/(EigenValuesD[p]-Xr);
             }
             Sum1+=1; Sum2+=1;
             test=Sum1*Sum2;
             if(test<0) Upper=Xr;
             else if(test>0) Lower=Xr;
             else ea=0;
       }
       LamdaConnSingle[i]=Xr;/* Newly computed eigenvalue*/
       SigmaC+=Xr;/* Sub-total of new eigenvalues*/
}
teval=MPI_Wtime();/* Ending time for computing eigenvalues*/
ttval=teval-tsval; /* Total time for computing eigenvalues*/
/* ….Continue with next text box….*/
```

Fig.F-11 MPI # in C source code for algorithm #3 -1 ROM– part (11)

```
/* Compute largest eigenvalue*/
/* For the last processor, recvcnt was subtracted by one, so missed one old eigenvalue*/
if (rank_in_world==ProcessorNum-1) SigmaD+=Lamda[recvcnt];
tsreduce=MPI_Wtime();
/* MPI_Allreduce to compute sub-totals, both old and new*/
MPI_Allreduce(&SigmaC,&TotalC,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
MPI_Allreduce(&SigmaD,&TotalD,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
tereduce=MPI_Wtime();
ttreduce=tereduce-tsreduce;
/* For the last processor, add recvcnt back with 1 for further eigenvector computation*/
if (rank_in_world= =ProcessorNum-1){
     recvcnt+=1;
     /* Finally, it is ready to compute the largest eigenvalue*/
     LamdaConnSingle[recvcnt-1]=TotalD-TotalC+2*Line;
}
  tsvec=MPI_Wtime();
  /* Now 'recvcnt1' is the multiple of unroll, this is for use of unrolling */
  recvcnt1=recvcnt-r1;
/*…Continue with next text box…*/
```

Fig.F-12 MPI in #C source code for algorithm #3 -1 ROM– part (12)

```
/* Start to compute eigenvectors*/
for(i=0;i<recvcnt1;i+=unroll){/* Compute 'recvcnt1' eigenvectors, index stride =unroll*/
      if ( LamdaConnSingle[i]!=0){/* Since 1st eigenvalue is zero, it may slow down the process*/
          for(u=0; u<unroll; u++)/* Initialization*/
               for(q=0;q<Nodes;q++)
                    Temp[u][q]=0;
           for(p=0;p<Nodes;p++)/* Compute (3.2.11)*/
                for(q=0;q<Nodes;q++)
                    for(u=0;u<unroll;u++)
                        Temp[u][q] +=
                         U1[p]/(EigenValuesD[p]-LamdaConnSingle[i+u]) * EigenVectorDtrans[p][q];
                 for (u=0;u<unroll;u++){/* Normalization*/
                     Mag=0;
                     for(q=0;q<Nodes;q++)
                         Mag += pow(Temp[u][q],2);
                         Mag=sqrt(Mag);
                         for(p=0;p<Nodes;p++)
                            EigenVectorSingle[i+u][p]=Temp[u][p]/Mag;
                }
       }
}
for (;i<recvcnt;i++){/* Compute leftovers*/
             Mag=0;
             for(q=0;q<Nodes;q++) Temp[0][q]=0;/* Initialization*/
             for(p=0;p<Nodes;p++){
                   double temp10=U1[p]/(EigenValuesD[p]-LamdaConnSingle[i]);
                   for(q=0;q<Nodes;q++)
                        Temp[0][q]+=temp10*EigenVectorDtrans[p][q];
             }
             for (p=0;p<Nodes;p++)
                    Mag+=pow(Temp[0][p],2);
             Mag=sqrt(Mag);
             for (p=0;p<Nodes;p++)
                    EigenVectorSingle[i][p]=Temp[0][p]/Mag;
}
tevec=MPI_Wtime();/*Ending time for eigenvector computation*/
ttvec=tevec-tsvec;/*Total time for computing eigenvectors*/
/*….Continue with next text box… */
```

Fig.F-13 MPI in #C source code for algorithm #3 -1 ROM– part (13)

```
/* Continue with Connect…..*/
if (rank_in_world==0) LamdaConnSingle[0]=0.0;/* 1st eigenvalue*/
tsgather=MPI_Wtime();/*Starting time for MPI_Gather, and gather results*/
MPI_Gatherv(LamdaConnSingle,recvcnt,MPI_DOUBLE,LamdaConn,sendcnt,displs,
MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Gatherv(EigenVectorSingle,recvcnt*MAX,MPI_DOUBLE,EigenVectorFinal,
sendcnt1,displs1,MPI_DOUBLE,0,MPI_COMM_WORLD);
tegather=MPI_Wtime();/* ending time for Gather*/
ttgather=tegather-tsgather;/* Total time for Gather*/
ttc=ttscat1+ttscat2+ttreduce+ttgather;/* Total communication time*/
tsother3=MPI_Wtime();/* See chapter 5*/
e=sqrt(nu/Nodes);
for(i=0;i<Nodes;i++) EigenVectorFinal[0][i]=e;/*1st eigenvector*/
      teother3=MPI_Wtime();/* See chapter 5*/
      ttother3=teother3-tsother3;
      ttother=ttother1+ttother2+ttother3;
      tt=ttother+ttqs+ttc+ttval+ttvec;
      TimeVector[0]=ttqs;
      TimeVector[1]=ttgather;
      TimeVector[2]=ttscat1;
      TimeVector[3]=ttscat2;
      TimeVector[4]=ttreduce;
      TimeVector[5]=ttc;
      TimeVector[6]=ttval;
      TimeVector[7]=ttvec;
      TimeVector[8]=ttother1;
      TimeVector[9]=ttother2;
      TimeVector[10]=ttother3;
      TimeVector[11]=ttother;
      TimeVector[12]=tt;
      for (i=0;i<unroll;i++)/*Delete memory allocated*/
            free(Temp[i]);
      free(Lamda);
      free(LamdaConnSingle);
      MPI_Barrier(MPI_COMM_WORLD);
}/*Connect function ends…..*/
```

Fig.F-14 MPI in #C source code for algorithm #3-1 ROM – part (14)

Appendix G. MPI in #C Source codes for Algorithm #4 – 4 ROMs

Here below are the source codes for 4 ROMs of algorithm #4. Note that the ideas for indices partitioning and MPI_Gatherv implementation are from Mr. Francois Guertin from RQCHP which are different from those in algorithm #3.

```
/* Header files and global variables declaration*/
#include <math.h>
#include<stdio.h>
#include <stdlib.h>
#include "mpi.h" /* MPI header file*/
#define MAX 4500 /* Define matrices dimension*/
#define Ti1 30 /* Tie-in point from 1 of 1st ROM*/
#define Tj1 651 /* Tie-in point from 2 of 1st ROM*/
#define Line1 2.03 /*Connection line of 1st ROM*/
#define Ti2 683/* Tie-in point from 1-2 of 2nd ROM*/
#define Tj2 1963/* Tie-in point from 3 of 2nd ROM*/
#define Line2 3.08 /*Connection line of 2nd ROM*/
#define Ti3 2004 /* Tie-in point from 1-2-3 of 3rd ROM*/
#define Tj3 3433/* Tie-in point from 4 of 3rd ROM*/
#define Line3 6.08 /*Connection line of 3rd ROM*/
#define Ti4 4418/* Tie-in point from 1-2-3-4 of 4th ROM*/
#define Tj4 630/* Tie-in point from 1 of 4th ROM*/
#define Line4 4.5 /*Connection line of 4th ROM*/
#define TimeDim 10
int nprocs; /* Define global variable for number of processors*/
```

Fig.G-1 MPI in #C source codes for algorithm #4 – 4 ROMs – part (1)

```
/* Functions prototype declaration*/
void Install (int ProcessorNum, int Nodes, double EigenVectors[MAX][MAX],
              double EigenValues[MAX], int Tie1,int Tie2, double Line,
              double EigenVectorFinal[MAX][MAX], double LamdaConn[MAX]);
void Connect (int ProcessorNum, int Nodes1,int Nodes2,
               double EigenVectors1[MAX][MAX],
               double EigenVectors2[MAX][MAX], double EigenValues1[MAX],
               double EigenValues2[MAX], int Tie1, int Tie2, double Line,
               double EigenVectorFinal[MAX][MAX],
               double TimeVector[TimeDim], double LamdaConn[MAX]
               );
void quicksort (double array[ ], int indexL, int indexR);
```

Fig.G-2 MPI in #C source codes for algorithm #4 – 4 ROMs – part (2)

```
/* Start of Main function*/
int main(int argc, char **argv){
        /* Local variables declaration*/
        int i, j, printindex, rank, N1,N2,N3,N,N4,Times,p;
        double EigenValueD[MAX],EigenVec1[MAX][MAX],
        double EigenVec2[MAX][MAX],EigenVal1[MAX],EigenVal2[MAX];
        double EigenVec3[MAX][MAX],EigenVec4[MAX][MAX];
        double EigenVal3[MAX],EigenVal4[MAX];
        double VectorResults1[MAX][MAX], LamdaC1[MAX];
        double VectorResults2[MAX][MAX], LamdaC2[MAX];
        double VectorResults3[MAX][MAX], LamdaC3[MAX];
        double TimeVector[TimeDim],VectorResults4[MAX][MAX];
        double EigenVal1_2_3[MAX],EigenVec1_2_3[MAX][MAX];
        double EigenVec1_2[MAX][MAX],EigenVal1_2[MAX];
        double EigenVal1_2_3_4[MAX],EigenVec1_2_3_4[MAX][MAX], LamdaC[MAX];
        double EigenVal_ring[MAX],tsbcast, tebcast, ttbcast, tsinit, teinit, ttinit;
        double tsinput, teinput, ttinput, ts1,te1,tt1, ts2,te2,tt2, ts3,te3,tt3;
        double ts4,te4,tt4,tt, time=0,starttime=0,stoptime=0;
        FILE *fp1,*fp2,*fp3,*fp4,*fp5,*fp6,*fp7,*fp8,*fp9,*fp10;
        FILE *fp11,*fp12,*fp13,*fp14,*fp15,*fp16,*fpt1;
        /*MPI environment initialization and time measurement*/
        tsinit=MPI_Wtime(); /* Starting time of initialization*/
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
        teinit=MPI_Wtime(); /* Ending time of initialization*/
        ttinit=teinit-tsinit; /* Total time for initialization*/
```

Fig.G-3 MPI in #C source codes for algorithm #4 – 4 ROMs – part (3)

```
/* Read data input from local directories */
tsinput=MPI_Wtime();/* Starting time data input*/
   if (rank= =0){
        fp1 =fopen("Vector1_4400.txt", "r" ); /* Eigenvectors of system1*/
        fscanf(fp1, "%d\n", &N1); /* Dimension of eigenvectors matrix*/
        for (i=0;i<N1;i++){
            for (j=0;j<N1;j++)
                fscanf(fp1, "%lf", &EigenVec1[i][j]);
             fscanf(fp1,"\n");
        }
        fclose(fp1);
        fp2 =fopen("Vector2_4400.txt", "r" ); /* Eigenvectors of system2*/
        fscanf(fp2, "%d\n", &N2);
        for (i=0;i<N2;i++){
            for (j=0;j<N2;j++)
                 fscanf(fp2, "%lf", &EigenVec2[i][j]);
            fscanf(fp2,"\n");
        }
        fclose(fp2);
        fp3 =fopen("Vector3_4400.txt", "r" ); /* Eigenvectors of system3*/
        fscanf(fp3, "%d\n", &N3);
        for (i=0;i<N3;i++){
             for (j=0;j<N3;j++)
                  fscanf(fp3, "%lf", &EigenVec3[i][j]);
             fscanf(fp3,"\n");
        }
        fclose(fp3);
        fp4 =fopen("Vector4_4400.txt", "r" ); /* Eigenvectors of system4*/
        fscanf(fp4, "%d\n", &N4);
        for (i=0;i<N4;i++){
             for (j=0;j<N4;j++)
                  fscanf(fp4, "%lf", &EigenVec4[i][j]);
             fscanf(fp4,"\n");
        }
        fclose(fp4); /* …….Continue with next textbox……*/
```

Fig.G-4 MPI in #C source codes for algorithm #4 – 4 ROMs – part (4)

```
/* …..Continue with data input…*/
fp5 =fopen("Value1_4400.txt", "r" ); /*Eigenvalues of system1*/
for (i=0;i<N1;i++)
      fscanf(fp5, "%lf", &EigenVal1[i]);
fscanf(fp5,"\n");
fclose(fp5);
fp6 =fopen("Value2_4400.txt", "r" ); /* Eigenvalues of system 2*/
for (i=0;i<N2;i++)
      fscanf(fp6, "%lf", &EigenVal2[i]);
fscanf(fp6,"\n");
fclose(fp6);
fp7 =fopen("Value3_4400.txt", "r" ); /* Eigenvalues of system 3*/
for (i=0;i<N3;i++)
      fscanf(fp7, "%lf", &EigenVal3[i]);
fscanf(fp7,"\n");
fclose(fp7);
fp8 =fopen("Value4_4400.txt", "r" ); /* Eigenvalues of system 4*/
for (i=0;i<N4;i++)
      fscanf(fp8, "%lf", &EigenVal4[i]);
fscanf(fp8,"\n");
fclose(fp8);
fp9 =fopen("Vector1_2_4400.txt", "r" ); /* Eigenvectors of 1-2 MATLAB*/
for (i=0;i<N1+N2;i++){
      for (j=0;j<N1+N2;j++)
            fscanf(fp9, "%lf", &EigenVec1_2[i][j]);
      fscanf(fp9,"\n");
}
fclose(fp9);
fp10 =fopen("Vector1_2_3_4400.txt", "r" ); /* Eigenvectors of 1-2-3    MATLAB*/
for (i=0;i<N1+N2+N3;i++){
     for (j=0;j<N1+N2+N3;j++)
           fscanf(fp10, "%lf", &EigenVec1_2_3[i][j]);
     fscanf(fp10,"\n");
}
fclose(fp10); /* …….Continue with next textbox…..*/
```

Fig.G-5 MPI in #C source codes for algorithm #4 – 4 ROMs – part (5)

```
/* ….Continue with data input….*/
fp11 =fopen("Vector1_2_3_4_4400.txt", "r" ); /* Eeigenvectors matrix of 1-2-3-4 MATLAB*/
for (i=0;i<N1+N2+N3+N4;i++){
      for (j=0;j<N1+N2+N3+N4;j++)
            fscanf(fp11, "%lf", &EigenVec1_2_3_4[i][j]);
      fscanf(fp11,"\n");
}
fclose(fp11);
fp12 =fopen("Value1_2_4400.txt", "r" ); /* Eigenvalues of 1-2 MATLAB*/
for (i=0;i<N1+N2;i++)
    fscanf(fp12, "%lf", &EigenVal1_2[i]);
fscanf(fp12,"\n");
fclose(fp12);
fp13 =fopen("Value1_2_3_4400.txt", "r" ); /* Eigenvalues of 1-2-3 MATLAB*/
for (i=0;i<N1+N2+N3;i++)
      fscanf(fp13, "%lf", &EigenVal1_2_3[i]);
fscanf(fp13,"\n");
fclose(fp13);
fp14 =fopen("Value1_2_3_4_4400.txt", "r" ); /*Eigenvalues of 1-2-3-4 MATLAB*/
for (i=0;i<N1+N2+N3+N4;i++)
      fscanf(fp14, "%lf", &EigenVal1_2_3_4[i]);
fscanf(fp14,"\n");
fclose(fp14);
fp15 =fopen("ValueT_4400_4th_connection.txt", "r" ); /* Eigenvalues of 1-2-3-4-1 MATLAB*/
for (i=0;i<N1+N2+N3+N4;i++)
    fscanf(fp15, "%lf", &EigenVal_ring[i]);
fscanf(fp15,"\n");
fclose(fp15);
  }/* …End of data input…*/
teinput=MPI_Wtime(); /* Ending time of data input*/
ttinput=teinput-tsinput; /* Total time of data input*/
/*…. Continue with next textbox….*/
```

Fig.G-6 MPI in #C source codes for algorithm #4 – 4 ROMs – part (6)

```
/* Broadcast data from root processor, P0*/
tsbcast=MPI_Wtime(); /* Starting time for data broadcast, for each files read from local directory*/
MPI_Bcast (&N1,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast (&N2,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast (&N3,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast (&N4,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVec1,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVec2,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVec3,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVec4,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVec1_2,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVec1_2_3,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVec1_2_3_4,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVal1,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVal2,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVal3,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVal4,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVal1_2,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVal1_2_3,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast (EigenVal1_2_3_4,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);/* Bcast ends*/
starttime = MPI_Wtime();/* Ending time for data broadcast , also starting time for computation*/
ttbcast=starttime-tsbcast; /* Total data broadcast time*/
/* …Continue with next textbox……*/
```

Fig.G-7 MPI in #C source codes for algorithm #4 – 4 ROMs – part (7)

```
/* Start computation, i.e. 4 ROMs*/
ts1=MPI_Wtime();/* Starting time for ROM 1 */
Connect(nprocs,N1,N2,EigenVec1,EigenVec2,EigenVal1,EigenVal2,Ti1,Tj1,Line1,
VectorResults1,TimeVector,LamdaC); /* ROM 1*/
te1=MPI_Wtime();/* Ending time for ROM 1 */
Connect(nprocs,N1+N2,N3,VectorResults1,EigenVec3,LamdaC,EigenVal3,Ti2,
Tj2,Line2,VectorResults2,TimeVector,LamdaC1); /* ROM 2*/
te2=MPI_Wtime(); /* Ending time for ROM 2 */
Connect(nprocs,N1+N2+N3,N4,VectorResults2,EigenVec4,LamdaC1,EigenVal4,
Ti3,Tj3,Line3,VectorResults3,TimeVector,LamdaC2); /* ROM 3*/
te3=MPI_Wtime(); /* Ending time for ROM 3 */
Install(nprocs,N1+N2+N3+N4,VectorResults3,LamdaC2,Ti4,Tj4,Line4,
VectorResults4,LamdaC3); /* ROM 4 and computation ends*/
stoptime=MPI_Wtime();/* Ending time for ROM 4 and total ending time */
tt1=te1-ts1; /* Computation time for ROM 1 */
tt2=te2-te1; /* Computation time for ROM 2 */
tt3=te3-te2; /* Computation time for ROM 3 */
tt4=stoptime-te3; /* Computation time for ROM 4 */
tt=tt1+tt2+tt3+tt4;
/* Rearrange eigenvalues from 4 systems for the purpose of printout*/
for (i=0;i<N1+N2+N3+N4;i++){
      if(i<(N1)) EigenValueD[i]=EigenVal1[i];
      else if ((i>=N1) && (i<N1+N2)) EigenValueD[i]=EigenVal2[i-N1];
      else if ((i>=N1+N2) && (i<N1+N2+N3)) EigenValueD[i]=EigenVal3[i-N1-N2];
      else EigenValueD[i]=EigenVal4[i-N1-N2-N3];
}
/*Quick sort for eigenvelues from 4 islanded systems for the purpose to print out*/
quicksort(EigenValueD,0,(N1+N2+N3+N4-1));
MPI_Barrier(MPI_COMM_WORLD); /* Synchronize all the processors*/
time=stoptime-starttime; /* Total computation time for 4 ROMs */
/* ….Continue with next textbox*/
```

Fig.G-8 MPI in #C source codes for algorithm #4 – 4 ROMs – part (8)

```
/*Print out all the computed eigenvectors and eigenvalues*/
if (rank= =0){
    /* Printout for good format, i.e. only 40 eigenvectors show on each line*/
    Times=(N1+N2+N3+N4)/40;
    if ((N1+N2+N3+N4)%40!=0) Times+=1;
    printindex=0;
    /*Eigenvectors printout*/
    fpt1=fopen("EigenVectors_4400_ring_check_May_24.txt","w");
    for(p=0;p<Times;p++){
          for(i=0;i<N1+N2+N3+N4;i++){
                for(j=printindex;j<printindex+40;j++)
                     fprintf(fpt1," %20.16lf",VectorResults4[i][j]);
                fprintf(fpt1,"\n");
          }
          printindex+=40;
          fprintf(fpt1,"\n\n");
    }
    fprintf(fpt1,"\n\n");
    /*Eigenvalues printout*/
    fpt1=fopen("EigenValues_4400_test_F_4_May_29.txt","w");
    fprintf(fpt1,"\n    k           Matlab Lamda_D                                    ");
    fprintf(fpt1,"Matlab Lamda_C                                ");
    fprintf(fpt1,"CLUMEQ Lamda_C\n");
    for(i=0;i<N1+N2+N3+N4;i++){
        /* Eigenvalues before ROM*/
        fprintf(fpt1,"\n%3d     %20.16lf                    ",i+1,EigenValueD[i]);
        fprintf(fpt1,"%20.16lf                      ",EigenVal_ring[i]);/* Computed eigenvalues*/
        fprintf(fpt1," %20.16lf                ",LamdaC3[i]);/* Eigenvalues from MATLAB*/
        fprintf(fpt1,"\n");
    }
/* …Ccontinue with next textbox….*/
```

Fig.G-9 MPI in #C source codes for algorithm #4 – 4 ROMs – part (9)

```
 /* Continue with results printout*/
            fprintf(fpt1,"\nComputationTime=%lf\n",time); /* Total computation time*/
            fprintf(fpt1,"\nTotalInitializationTime=%lf\n",ttinit);/* Initialization time*/
            fprintf(fpt1,"\nTotalDataInputTime=%lf\n",ttinput); /* Data input time*.
            fprintf(fpt1,"\nBcastTimeStart=%lf\n",ttbcast); /* Data broadcast time*/
            fprintf(fpt1,"\nBcastTimeEnd=%lf\n",TimeVector[0]);/* Bcast time at end of function*/
            fprintf(fpt1,"\nTT1=%lf\n",tt1); /* Computation time for ROM 1*/
            fprintf(fpt1,"\nTT2=%lf\n",tt2); /* Computation time for ROM 2*/
            fprintf(fpt1,"\nTT3=%lf\n",tt3); /* Computation time for ROM 3*/
            fprintf(fpt1,"\nTT4=%lf\n",tt4); /* Computation time for ROM 4*/
            fprintf(fpt1,"\nTT=%lf\n",tt); /* Total computation time*/
            fprintf(fpt1,"\n\n");
   }
   MPI_Finalize(); /* Terminates MPI environment*/
   return 0; /* Main function ends*/
}
/* Continue with next textbox*/
```

Fig.G-10 MPI in #C source codes for algorithm #4 – 4 ROMs – part (10)

```
/* Install function starts……*/
void Install (
                int ProcessorNum, int Nodes,
                double EigenVectors[MAX][MAX],
                double EigenValues[MAX],
                int Tie1, int Tie2, double Line,
                double EigenVectorFinal[MAX][MAX],
                double LamdaConn[MAX]
                )
   {
   int rank_in_world, k, i, j, maxit=100, iter[MAX],Times, p, q, u, inMin, inMax;
   int num, r, offsetVect=0, offsetMat=0, displsVect[MAX], displsMat[MAX];
   int sendcntVect, sendcntMat, recvcntVect[MAX], recvcntMat[MAX];
   double EigenVectorDtrans[MAX][MAX],EigenValuesDk[MAX];
   double EigenVectorFinal_1[MAX][MAX], EigenValuesD[MAX];
   double Lower,Upper,Xr,Xold,ea,test,double Sum1,Sum2,starttime,stoptime,time,Mag;
   double U[MAX],U1[MAX],FirstVector[MAX],es=1e-12;
   double *VecA,*VecB, *TempVec[unroll];
   /*declaration for unrolling factor manually*/
   #define unroll 8
   MPI_Comm_rank(MPI_COMM_WORLD,&rank_in_world); /*Rank determination*/
   for (i=0;i<Nodes;i++) /*Transpose input eigenvectors matrix*/
         for (j=0;j<Nodes;j++)
               EigenVectorDtrans[i][j]=EigenVectors[j][i];
   /*Set upper bound for largest eigenvalues*/
   /* Note the largest eigenvalue could be computed using (3.6.2)*/
   EigenValues[Nodes]=EigenValues[Nodes-1]+2*Line;
/* …Continue with next text box….*/
```

Fig.G-11 MPI in #C source codes for algorithm #4 – 4 ROMs – part (11)

```
/* Tasks partitioning and other*/
num=Nodes/ProcessorNum; /* Compute minimum tasks each processor will do*/
r=Nodes%ProcessorNum; /* r is remainder tasks and distribute to r processors, each has one*/
/* Tasks partitioning, MPI_Gather displacement, send and receive counts etc*/
For (i=0; i<ProcessorNum; i++){
      int rest = 0;
      if (i<r) rest = 1;
      displsVect[i]=offsetVect; /* Displacement for eigenvalue for gather*/
      offsetVect += (num+rest);
      recvcntVect[i]=(num+rest);/* # of eigenvalues that each processor provides*/
      displsMat[i]=offsetMat; /* Displacement of eigenvector matrix for gather*/
      offsetMat += ((num+rest)*MAX);
      recvcntMat[i]=(num+rest)*MAX; /* # of elements of eigenvectors that each    provides*/
}
sendcntVect = recvcntVect[rank_in_world]; /* # of eigenvalues gathered from each processor*/
sendcntMat = recvcntMat[rank_in_world]; /*# of eigenvector elements gathered from each */
inMin = displsVect[rank_in_world]; /* Indices partitioning*/
if (rank_in_world<r) inMax =    inMin + num + 1; /* rank 0 ~ r-1 compute one more tasks*/
else inMax =    inMin + num;
if (rank_in_world= =0) inMin++;/* 1st eigenpair needs to be computed separately*/
/* Dynamic memory allocation for temp variable of eigenvector computation*/
for(i=0;i<unroll;i++)
      TempVec[i]=(double*)malloc((Nodes)*sizeof(double));
VecA=(double*)malloc(Nodes*sizeof(double));/* Temp memory allocation*/
VecB=(double*)malloc(Nodes*sizeof(double));/* Temp memory allocation*/
/* Store the ith and jth row of input eigenvector matrix to two arrays for later use*/
For (i=0;i<Nodes;i++){
      VecA[i]=EigenVectorDtrans[i][Tie1];
      VecB[i]=EigenVectorDtrans[i][Tie2];
}
/* …. Continue with next text box….*/
```

Fig.G-12 MPI in #C source codes for algorithm #4 – 4 ROMs – part (12)

```
/*Eigenvalues computation starts…….*/
for (k=inMin;k<inMax;k++) (/* Each processor compute (inmAX-in Min) eigenvalues*/
        Upper=EigenValues[k+1]-1e-14; /* Modify original upper bound*/
        Lower=EigenValues[k]+1e-14;      /* Modify original lower bound*/
        Xr=Lower;
        iter[k]=0; /* Count iterations if necessary*/
        while(1) {/* Infinite loop for each eigenvalue*/
                /* If two eigenvalues are close, assume the average is the root*/
                if (fabs (EigenValues[k]-EigenValues[k+1])<=1e-9){
                    Xr=(Upper+Lower)/2;
                    break;
                }
                Xold=Xr;
                Xr=(Upper+Lower)/2;
                iter[k]++;
                if(Xr!=0) ea=fabs(Xr-Xold); /* Compute absolute errors*/
                Sum1=0;
                Sum2=0;
                for(i=0;i<Nodes;i++){
                    U1[i]=VecA[i]-VecB[i];
                    U[i]=Line*pow(U1[i],2);
                    Sum1+=U[i]/(EigenValues[i]-Lower);
                    Sum2+=U[i]/(EigenValues[i]-Xr);
                }
                Sum1+=1; /* Function f evaluated at new lower bound*/
                Sum2+=1; /* Function f evaluated at new middle point*/
                test=Sum1*Sum2;/* Test the sign of the product, then reset bounds*/
                if(test<0)    Upper=Xr; /* Bisection criteria*/
                else if(test>0)    Lower=Xr; /*Bisection criteria*/
                else ea=0; /* Bisection criteria*/
                /* If falls into the predefined error, stops and continue with next root-finding*/
                if((ea<=es)||(iter[k]>maxit)) break;
        }
        LamdaConn[k]=Xr; /* Newly computed eigenvalue*/
}/*…… Continue with next text box….*/
```

Fig.G-13 MPI in #C source codes for algorithm #4 – 4 ROMs – part (13)

```
/*Eigenvector computation starts….. and normalize */
for (k=inMin;k+unroll-1<inMax;k+=unroll){/* Index stride is unroll instead of one*/
      for(u=0; u<unroll; u++) /* Initialization of temp vector*/
            for(q=0;q<Nodes;q++)
                  TempVec[u][q]=0;
      for(p=0;p<Nodes;p++)    /* Compute (3.2.10)*/
            for(q=0;q<Nodes;q++)
                 for (u=0;u<unroll;u++)
                     TempVec[u][q] +=
                     U1[p]/(EigenValues[p]-LamdaConn[k+u]) * EigenVectorDtrans[p][q];
      for(u=0;u<unroll;u++) {/* Normalization of eigenvectors*/
            Mag=0;
            for(q=0;q<Nodes;q++) Mag += pow(TempVec[u][q],2);
            Mag=sqrt(Mag);
            for(i=0;i<Nodes;i++) EigenVectorFinal_1[k+u][i]=TempVec[u][i]/Mag;
      }
}
/* Infinite loop to compute the leftovers if (inMax-inMin) is not a multiple of unroll factor*/
for(;k<inMax;k++){
      for(q=0;q<Nodes;q++)
          TempVec[0][q]=0;
      for(p=0;p<Nodes;p++){
              double temp = U1[p]/(EigenValues[p]-LamdaConn[k]);
              for(q=0;q<Nodes;q++)
                    TempVec[0][q] += temp * EigenVectorDtrans[p][q];
      }
      Mag=0;
      for(q=0;q<Nodes;q++)
          Mag += pow(TempVec[0][q],2);
      Mag=sqrt(Mag);
      for(i=0;i<Nodes;i++)/* Normalization of eigenvectors*/
          EigenVectorFinal_1[k][i]=TempVec[0][i]/Mag;
}/* Continue with next text box*/
```

Fig.G-14 MPI in #C source codes for algorithm #4 – 4 ROMs – part (14)

```
/* First eigenpair, gather and broadcast results and free memory */
LamdaConn[0]=0.0;/* 1st eigenvalue*/
double numerator=1.0,e; e=sqrt(numerator/Nodes);
for(i=0;i<Nodes;i++) EigenVectorFinal_1[0][i]=e;
/*Gather eigenvalues*/
MPI_Gatherv(&LamdaConn[displsVect[rank_in_world]],sendcntVect,MPI_DOUBLE,
LamdaConn,recvcntVect,displsVect,MPI_DOUBLE,0, MPI_COMM_WORLD);
/*Gather eigenvectors*/
MPI_Gatherv(((double*)EigenVectorFinal_1)+displsMat[rank_in_world],
sendcntMat,    MPI_DOUBLE, EigenVectorFinal_1, recvcntMat, displsMat,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
/* Broadcast data for next ROM to use as input*/
MPI_Bcast(LamdaConn,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(EigenVectorFinal_1,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
/* Transpose back the computed eigenvectors matrix*/
for(i=0;i<Nodes;i++)
      for(j=0;j<Nodes;j++)
            EigenVectorFinal[i][j]=EigenVectorFinal_1[j][i];
/* Release memory that has been allocated*/
free(VecA);
free(VecB);
for(i=0;i<unroll;i++)
      free(TempVec[i]);
MPI_Barrier(MPI_COMM_WORLD);/* Synchronization*/
}/* Install function ends….and continue with Connect function*/
```

Fig.G-15 MPI in #C source codes for algorithm #4 – 4 ROMs – part (15)

```
/*Connect function starts…. Note the comments for the same codes are ignored*/
void Connect ( int ProcessorNum, int Nodes1, int Nodes2,
                  double EigenVectors1[MAX][MAX],double EigenVectors2[MAX][MAX],
                  double EigenValues1[MAX], double EigenValues2[MAX],
                  int Tie1, int Tie2, double Line, double EigenVectorFinal[MAX][MAX],
                  double TimeVector[TimeDim], double LamdaConn[MAX]
                )
  {
  int rank_in_world,k=1,i,j,maxit=100,iter[MAX], Times, Nodes, p, q, u, inMin, inMax;
  int num,r,offsetVect=0,offsetMat=0,displsVect[MAX],displsMat[MAX];
  int sendcntVect,sendcntMat,recvcntVect[MAX],recvcntMat[MAX];
  double EigenVectorDtrans[MAX][MAX],EigenValuesDk[MAX];
  double EigenVectorFinal_1[MAX][MAX], FirstVector[MAX],EigenValuesD[MAX];
  double Lower,Upper,Xr,Xold,ea,test, Sum1,Sum2,tsbcast,tebcast,ttbcast,Mag;
  doubleU[MAX],U1[MAX], es=1e-12;
  double *VecA,*VecB, *TempVec[unroll];
  #define unroll 8
  MPI_Comm_rank(MPI_COMM_WORLD,&rank_in_world);
  Nodes=Nodes1+Nodes2; /* Compute total nodes number */
  /* Arrange two input eigenvector matrices and eigenvalues arrays into the form (3.3.2)
  for(i=0;i<Nodes;i++) {
      for(j=0;j<Nodes;j++){
            if((i<Nodes1)&&(j<Nodes1))
                  EigenVectorDtrans[i][j]=EigenVectors1[j][i];
             else if((i>=Nodes1)&&(j>=Nodes1))
                  EigenVectorDtrans[i][j]=EigenVectors2[j-Nodes1][i-Nodes1];
             else
                  EigenVectorDtrans[i][j]=0;
      }
  }
/* …….Continue with next text box……*/
```

Fig.G-16 MPI in #C source codes for algorithm #4 – 4 ROMs – part (16)

```
for (i=0;i<Nodes;i++) {
      if(i<Nodes1){
            EigenValuesD[i]=EigenValues1[i];/* For computation*/
            EigenValuesDk[i]=EigenValuesD[i];/*For sorting*/
      }
      else{
            EigenValuesD[i]=EigenValues2[i-Nodes1];
            EigenValuesDk[i]=EigenValuesD[i];
      }
}
quicksort(EigenValuesDk,0,(Nodes-1));/* Sorting eiegenvalues in ascending order*/
EigenValuesDk[Nodes]=EigenValuesDk[Nodes-1]+2*Line;
num=Nodes/ProcessorNum;
r=Nodes%ProcessorNum;
    for (i=0; i<ProcessorNum; i++){
          int rest = 0;
          if (i<r) rest = 1;
          displsVect[i]=offsetVect;
          offsetVect += (num+rest);
          recvcntVect[i]=(num+rest);
          displsMat[i]=offsetMat;
          offsetMat += ((num+rest)*MAX);
          recvcntMat[i]=(num+rest)*MAX;
    }
    sendcntVect = recvcntVect[rank_in_world];
    sendcntMat = recvcntMat[rank_in_world];
    inMin = displsVect[rank_in_world];
    if (rank_in_world<r)    inMax =    inMin + num + 1;
    else    inMax =    inMin + num;
    if (rank_in_world==0)    inMin++;
    for(i=0;i<unroll;i++)
        TempVec[i]=(double*)malloc((Nodes)*sizeof(double));
/*….. Continue with next text box……*/
```

Fig.G-17 MPI in #C source codes for algorithm #4 – 4 ROMs – part (17)

```
/* ….Continue with above….*/
  VecA=(double*)malloc(Nodes*sizeof(double));
  VecB=(double*)malloc(Nodes*sizeof(double));
  for(i=0;i<Nodes;i++){
       VecA[i]=EigenVectorDtrans[i][Tie1];
       VecB[i]=EigenVectorDtrans[i][Tie2];
  }
  for(k=inMin;k<inMax;k++){
          Upper=EigenValuesDk[k+1]-1e-14;
          Lower=EigenValuesDk[k]+1e-14;
          Xr=Lower; iter[k]=0;
          while(1) {
                    if (fabs(EigenValuesDk[k]-EigenValuesDk[k+1])<=1e-9){
                              Xr=(Upper+Lower)/2;
                              break;
                     }
                     Xold=Xr; Xr=(Upper+Lower)/2;
                     iter[k]++;
                     if(Xr!=0) ea=fabs(Xr-Xold);
                     Sum1=0; Sum2=0;
                     for (i=0;i<Nodes;i++) {
                          U1[i]=VecA[i]-VecB[i];
                          U[i]=Line*pow(U1[i],2);
                          Sum1+=U[i]/(EigenValuesD[i]-Lower);
                          Sum2+=U[i]/(EigenValuesD[i]-Xr);
                     }
                     Sum1+=1; Sum2+=1;
                     test=Sum1*Sum2;
                     if(test<0)    Upper=Xr;
                     else if (test>0) Lower=Xr;
                     else ea=0;
                     if((ea<=es)|| (iter[k]>maxit)) break;
          }
          LamdaConn[k]=Xr;
  }
/* ….Continue with next text box….*/
```

Fig.G-18 MPI in #C source codes for algorithm #4 – 4 ROMs – part (18)

```
/* …..Continue with above…..*/
 for (k=inMin;k+unroll-1<inMax;k+=unroll){
      for(u=0; u<unroll; u++)
           for(q=0;q<Nodes;q++)
                TempVec[u][q]=0;
      for(p=0;p<Nodes;p++)
           for(q=0;q<Nodes;q++)
                for(u=0;u<unroll;u++)
                     TempVec[u][q] +=
                     U1[p]/(EigenValuesD[p]-LamdaConn[k+u]) * EigenVectorDtrans[p][q];
       for (u=0;u<unroll;u++){
           Mag=0;
           for (q=0;q<Nodes;q++)
                Mag += pow(TempVec[u][q],2);
           Mag=sqrt(Mag);
           for(i=0;i<Nodes;i++)
                EigenVectorFinal_1[k+u][i]=TempVec[u][i]/Mag;
       }
 }
 for(;k<inMax;k++){
      for(q=0;q<Nodes;q++)
          TempVec[0][q]=0;
      for(p=0;p<Nodes;p++){
          double temp = U1[p]/(EigenValuesD[p]-LamdaConn[k]);
          for(q=0;q<Nodes;q++)
               TempVec[0][q] += temp * EigenVectorDtrans[p][q];
      }
      Mag=0;
      for(q=0;q<Nodes;q++) Mag += pow(TempVec[0][q],2);
      Mag=sqrt(Mag);
      for(i=0;i<Nodes;i++) EigenVectorFinal_1[k][i]=TempVec[0][i]/Mag;
 }
/* …..Continue with next text box…*/
```

Fig.G-19 MPI in #C source codes for algorithm #4 – 4 ROMs – part (19)

```
/* ….Continue with above…*/
LamdaConn[0]=0.0;
double numerator=1.0,e; e=sqrt(numerator/Nodes);
   for(i=0;i<Nodes;i++)    EigenVectorFinal_1[0][i]=e;;
MPI_Gatherv(&LamdaConn[displsVect[rank_in_world]],sendcntVect,MPI_DOUBLE,
LamdaConn,recvcntVect,displsVect,MPI_DOUBLE,0, MPI_COMM_WORLD);
MPI_Gatherv(((double*)EigenVectorFinal_1)+displsMat[rank_in_world], sendcntMat,
MPI_DOUBLE, EigenVectorFinal_1, recvcntMat, displsMat, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
tsbcast=MPI_Wtime();
MPI_Bcast(LamdaConn,MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(EigenVectorFinal_1,MAX*MAX,MPI_DOUBLE,0,MPI_COMM_WORLD);
tebcast=MPI_Wtime();
ttbcast=tebcast-tsbcast;
TimeVector[0]=ttbcast;
for(i=0;i<Nodes;i++)
        for(j=0;j<Nodes;j++)
              EigenVectorFinal[i][j]=EigenVectorFinal_1[j][i];
free (VecA);
free (VecB);
for(i=0;i<unroll;i++)
      free(TempVec[i]);
 MPI_Barrier(MPI_COMM_WORLD);
}/* Function connect ends…*/
/* ….Continue with next text box…*/
```

Fig.G-20 MPI in #C source codes for algorithm #4 – 4 ROMs – part (20)

Appendix H. Sequential Quick Sort function

Appendix H shows 3 versions of quick sort functions based on the algorithm in [20]. Actually the algorithms are similar, only the pivot element selection strategies differ, i.e. selecting the 1st, the last and random element as the pivot.

```
/* Quick sort has 1st element as pivot*/
/*Swap two elements*/
void swap(double array[ MAX], int i, int j){
        double temp = array[i];
        array[i] = array[j];
        array[j] = temp;
}
/*Main body of quicksort*/
void quicksort (double array[MAX ], int q, int r) {
        int pivot = q, i; double x=array[q];
        if (q < r){/* Only sort with # of element>1*/
            for (i = q + 1; i <= r; i++)
                if (array[i] <= x){
                    pivot++;
                    swap (array,pivot,i);
            }
            swap(array,q,pivot); /* pivot is in its right place*/
            quicksort(array,q,pivot); /* Sort lefy sub-array*/
            quicksort(array,pivot+1,r);/*Sort right sub-array*/
      }
}
```

Fig.H-1 Quick sort #1 – the 1st element is pivot

```
/* Quick sort has the last element as pivot*/
void swap(double array[MAX], int i, int j){
        double temp = array[i];
        array[i] = array[j];
        array[j] = temp;
}
void quicksort(double array[ MAX], int q, int r) {
        int pivot = r, i;
        double x=array[r];
        if (q < r){
                for (i = r-1; i >= q; i--)
                        if (array[i] >=x){/*Compare with the last element*/
                            pivot--;
                            swap(array,pivot,i);
                        }
                swap(array,r,pivot);
                quicksort(array,pivot,r);
                quicksort(array,q,pivot-1);
        }
}
```

Fig.H-2 Quick sort #2 – the last element is pivot

```
/* Quick sort with random pivot*/
/*Swap two elements*/
void swap(double array[MAX], int i, int j){
        double temp = array[i];
        array[i] = array[j];
        array[j] = temp;
}
/* Generate randomly a pivot element*/
int Random(int i, int j) {
     //srand(time(NULL));/*need #include time.h*/
       return (i + rand( ) % (j-i+1));
}
void quicksort(double array[MAX ], int q, int r) {
        int pivot = q, i;
        if (q < r){
             swap(array,q,Random(q,r));/*Place random pivot in the 1st place*/
             for (i = q + 1; i <= r; i++)
                   if (array[i] <= array[q]){
                         pivot++;swap(array,pivot,i);
                   }
             swap(array,q,pivot);/*Pivot in the right place*/
             quicksort(array,q,pivot-1);
             quicksort(array,pivot+1,r);
     }
```

Fig.H-3 Quick sort #3 – random pivot element

Appendix I. MPI in #C Source codes for 2 Communicators

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
void set_groups(MPI_Comm*,MPI_Comm*);
int main (int argc, char** argv) {
   int rank,nprocs;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm comm_1, comm_2;
   set_groups (&comm_1, &comm_2);/*create new communicators*/
     if (rank = = 0)
           Connect (…., comm._1);
   else
           Install (…., comm._2);
  ……..
   MPI_Comm_free (&comm._1);
   MPI_Comm_free (&comm._2);
   MPI_Finalize();
   return 0;
}
void set_groups(MPI_Comm *comm_1,MPI_Comm *comm_2){
int key,color,Newrank;
MPI_Comm_rank(MPI_COMM_WORLD,&Newrank);
color=(Newrank%2);
key=Newrank;
MPI_Comm_split(MPI_COMM_WORLD,color,key,comm_1);
MPI_Comm_split(MPI_COMM_WORLD,color,key,comm_2);
}
```

Fig.I-1 MPI in #C pseudo codes for two communicators

Appendix J. Sample PBS Script File

Fig.J-1 shows a sample PBS script file submitted to super-computer system for running jobs.

```
#!/bin/bash
#PBS -N Job name
#PBS -l walltime=00:30:00
#PBS -m be
#PBS -M yu.jiao@mail.mcgill.ca
#PBS -l nodes=16: ppn=4
#PBS -o MPI-stdo.output
#PBS -e MPI-stderr.output
#PBS -V
# ----------------------------
mpirun -np 64 -machinefile $PBS_NODEFILE ./<Output>
```

Fig.J-1 Sample PBS script file