Old Dominion University
ODU Digital Commons

Civil & Environmental Engineering Theses & Dissertations
Civil & Environmental Engineering

Fall 2016

Efficient Domain Decomposition Algorithms and Applications in Transportation and Structural Engineering

Paul W. Johnson III
Old Dominion University, pjohn018@odu.edu

Follow this and additional works at: https://digitalcommons.odu.edu/cee_etds

Part of the Civil Engineering Commons, Structural Engineering Commons, and the Transportation Engineering Commons

Recommended Citation

Johnson, Paul W.. "Efficient Domain Decomposition Algorithms and Applications in Transportation and Structural Engineering" (2016). Doctor of Philosophy (PhD), Dissertation, Civil & Environmental Engineering, Old Dominion University, DOI: 10.25777/h2zk-9h76
https://digitalcommons.odu.edu/cee_etds/11

This Dissertation is brought to you for free and open access by the Civil & Environmental Engineering at ODU Digital Commons. It has been accepted for inclusion in Civil & Environmental Engineering Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.

EFFICIENT DOMAIN DECOMPOSITION ALGORITHMS AND APPLICATIONS IN TRANSPORTATION AND STRUCTURAL ENGINEERING

by

Paul W. Johnson, III

B.S. December 2006, Old Dominion University
M.S. December 2008, Clemson University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

CIVIL ENGINEERING

OLD DOMINION UNIVERSITY
December 2016

Approved by:

Duc Nguyen (Director)

ManWo Ng (Co-Director)

Mecit Cetin (Member)

Gene Hou (Member)

ABSTRACT

EFFICIENT DOMAIN DECOMPOSITION ALGORITHMS AND APPLICATIONS IN TRANSPORTATION AND STRUCTURAL ENGINEERING

Paul W. Johnson, III
Old Dominion University, 2016
Director: Duc Nguyen, Co-Director: ManWo Ng

Domain decomposition is a divide-and-conquer strategy. In the first part of this dissertation, a new/simple/efficient domain decomposition partitioning algorithm is proposed to break a large domain into smaller sub-domains, in such a way as to minimize the number of system boundary nodes and to balance the work load for each sub-domain. This new domain decomposition algorithm is based on the network's shortest path solution. Numerical results indicate that the new Shortest Distance Decomposition Algorithm outperformed the most widely used METIS algorithm in 21 out of 27 tested (transportation) examples. In the second part of this dissertation, another new/simple and highly efficient shortest path algorithm is described for finding the shortest path from all-to-all (all source nodes to all destination nodes). This new Domain Decomposition-based Shortest Path algorithm basically finds the SP from all-to-all for each sub-domain, and assembles each sub-domains' shortest path solution to correctly obtain the original (un-partitioned) network's shortest path solution. Numerical results for real-life transportation networks have shown that the algorithm is much faster than the existing Dijkstra's shortest path algorithm. Finally, the Shortest Distance Decomposition Algorithm has also been shown to perform better than METIS when minimizing the non-zero fill-in terms of structural engineering stiffness matrices used during the finite element simultaneous linear equation solution process.

Copyright, 2016, by Paul W. Johnson, III, All Rights Reserved.

This manuscript is dedicated to my family. Without their sacrifices and steadfast support, this would never have been possible.

ACKNOWLEDGMENTS

Many people have contributed to the successful completion of this dissertation. I would be remiss not to extend my heartfelt thanks and gratitude to my committee members, Drs. Duc Nguyen, ManWo Ng, Mecit Cetin, and Gene Hou, for their patience, their hours of guidance on my research, and their editing of this manuscript, although the determined efforts of my committee chair, Dr. Nguyen, deserve special recognition. Without his guidance, this accomplishment would not have been possible.

Technical assistance with running the Turing Cluster is also appreciated. The members of the Old Dominion University's High Performance Computing Center, specifically Terry Stillwell, John Pratt, and Je'aime Powell were especially helpful. I would also like to thank the Old Dominion University Research Foundation and the TranLIVE Tier I University Transportation Center for the partial financial support, via seed grant #533561.

Finally, I would like to thank my family. To my wife, Christin: thank you for your unwavering support and your continued encouragement in allowing me to follow my dreams, spending countless hours listening to my concerns when no one else had an answer. And to my children, Calista, Harper, Paul, and Emersyn: your numerous sacrifices which were required during the years leading up to the completion of this document will not be forgotten, and will forever be appreciated.

NOMENCLATURE

DD	Domain Decomposition
DDSP	Domain Decomposition Based Shortest Path Algorithm
FEA	Finite Element Analysis
inode	Represents the source node
jnode	Represents the destination node
LCA	Label Correcting Algorithm
NP	Number of partitions or Number of sub-domains
P-LCA	Polynomial Label Correcting Algorithm
SBN	System Boundary Node
SDDA	Shortest Distance Decomposition Algorithm
SP	Shortest Path
ST	Shortest Time in the shortest path solution

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

INTRODUCTION 1
Literature Review 2
Method and Procedure 4

THE SHORTEST DISTANCE DECOMPOSITION ALGORITHM 6
Introduction 6
The Shortest Distance Decomposition Algorithm 8
Comparisons with METIS Using Real-World Transportation Networks 20
Conclusions 30

THE DOMAIN DECOMPOSITION BASED SHORTEST PATH ALGORITHM 32
Introduction 32
Description of a Small-Scale Network to be Analyzed 35
Solution of Small-Scale Network Utilizing the DDSP Algorithm 37
Application of the DDSP Algorithm to Large-Scale Networks 47
Conclusions 50

USING THE SDDA TO PRODUCE FILL REDUCING ORDERINGS 52
Introduction 52
Using the SDDA to Produce Fill Reducing Orderings 52
Multi-Level Reordering Using the SDDA as a Preordering Algorithm 54
Conclusions 56

CONCLUSIONS AND RECOMMENDATIONS 58

REFERENCES 61

APPENDICES 67
APPENDIX A: SDDA PARTITION GRAPHS 68
APPENDIX B: MATLAB Source Code for the SDDA 76
APPENDIX C: MATLAB Source Code for the DDSP Algorithm 85
APPENDIX D: Example Input/Output for the SDDA 96
APPENDIX E: Example Input/Output for the DDSP Algorithm 103

VITA 108

LIST OF TABLES

Table Page

2-1: Shortest Path Information for Source Node 1 12
2-2: Information for Source Nodes 1 and 15 Shortest Path 13
2-3: Populated Sub-Domains 15
2-4: New Node Numbering Scheme 17
2-5: System Boundary Node Comparison between the SDDA and METIS 21
2-6: Partitioning Time Comparison - METIS and The SDDA 22
2-7: Size of each sub-domain provided by METIS 29
2-8: Size of each sub-domain provided by the SDDA 30
3-1: Connectivity Information for Sub-Domains P_0, P_1, and P_2 39
3-2: Shortest Path Solution for Each Sub-Domain - [D] 40
3-3: Shortest Path Solution for Each Sub-Domain - [PRED] 41
3-4: Shortest Path Solution for each Sub-Domain with SBNs added - [D] 42
3-5: Shortest Path Solution for each Sub-Domain with SBNs added - [PRED] 43
3-6: Fully Assembled Shortest Path Solution for Original Network - [D] 46
3-7: Fully Assembled Shortest Path Solution for Original Problem - [PRED] 47
3-8: Networks Tested with Classical Serial Dijkstra Solution Time 48
3-9: SDDA Partitioning Time for Tested Networks 49
3-10: Total Serial Solution Time – SDDA Plus DDSP 49
4-1: Fill-In Term Comparison - METIS and the SDDA 53
4-2: Total Number of Non-Zeros after LU Factorization 55
4-3: Total Number of Non-Zero Terms after LU Factorization - NP = 2 55
4-4: Total Number of Non-Zero Terms after LU Factorization - NP = 3 55
4-5: Total Number of Non-Zero Terms after LU Factorization - NP = 4 56

LIST OF FIGURES

Figure Page

2-1: Network Topology and Connectivity Matrix for 5 Node Network 7
2-2: Network Connectivity with Nodal Ranks.. 10
2-3: Partitioning of the Chicago Network for NP = 4... 19
2-4: METIS Partitioning of the 15 Node Network 24
2-5: SDDA Partitioning of the 15 Node Network 25
2-6: METIS Partitioning of the 11 Node Network 26
2-7: SDDA Partitioning of the 11 Node Network 27
3-1: Network connectivity and associated sub-domain partitioning................................. 36

CHAPTER 1

INTRODUCTION

For numerous large-scale engineering and science problems, domain decomposition (DD) has been recognized as critical to obtain a solution within a reasonable amount of time, as has been demonstrated in the fields of computational fluid dynamics, aerospace engineering, structural engineering, and others. The basic premise of DD is to decompose a large domain or network into smaller sub-domains or sub-networks which are then typically addressed in parallel, i.e. each sub-problem is assigned to an independent computer processor. The solutions to these sub-problems are then integrated in order to recover the solution to the original problem. One of the key determinants for the success of DD is the number of so-called system boundary nodes. Roughly, the number of system boundary nodes determines the degree of interaction between the various sub-domains (a more precise definition of system boundary nodes will be provided in Step 6 of Section 2.2). As the number of system boundary nodes decreases, the more efficiently these sub-domains can be processed. For instance, Nguyen (2006) provides an example which demonstrates that when the number of system boundary nodes increases, the computational time to solve the problem can increase dramatically (from 0.45% to 97% of the overall computation time), clearly showing the need to keep the number of such nodes as small as possible.

In this manuscript, a simple, yet effective, heuristic algorithm is presented. The objective of the algorithm is to decompose a domain into a predefined number of interconnected sub-domains. The network must be divided in such a way that, as its first

priority, the number of system boundary nodes is small; and as its second priority, the number of nodes in each sub-network is of similar size. The first priority is critical to reduce the communication/interaction between the various sub-domains during the solution process. As shown by Nguyen (2006), this communication time typically increases at a much faster rate than the decrease in computation time made possible by a larger number of processors. The second objective is applied such that, in a parallel computational environment, one sub-domain does not dominate the overall solution time. Providing sub-domains of approximately equal size balances the workload among processors. It should be clear that the largest sub-domain will typically determine the computation time in a parallel processing situation. The algorithm presented is based on a simplified version of the well-known Label Correcting Algorithm (LCA) used for the shortest path class of problems, coupled with some simple heuristic rules.

1.1 Literature Review

Given that multiple processors are very common these days (with even simple personal computers having multiple cores), DD is a very timely and relevant technique. Various general heuristic algorithms have been developed by researchers in graph theory (e.g. see Farhat & Lesoinne, 1993; Chen & Taylor, 2002; Simon, 1991; Kernighan & Lin, 1970; and Karypis & Kumar, 1998) that can be used to partition any graph/network. In addition to the complex nature of these heuristics, it is by far not clear how efficient or effective these general graph partitioning algorithms are when applied to common civil engineering problems such as road networks, since these networks are known to possess unique characteristics. For instance, the degree of the nodes is typically in the range of 3

to 4, which does not necessarily hold for networks arising in other fields. It is to be noted that recently, Etemadnia et al. (2014) proposed two heuristic algorithms to decompose a network into multiple domains. Although the authors have presented their work in the context of transportation, the heuristics have only been tested on hypothetical networks. In this sense, their heuristics do not differ from the heuristics previously mentioned.

Various researchers in the transportation field have hinted at the use of DD; for example, it appears in works related to decentralized traffic management (e.g. see Pavlis & Papageorgiou, 1999; Logi & Ritchie, 2002). However, these studies always make the assumption that a transportation network can somehow be decomposed into a user-defined number of sub-networks, i.e. that the partitions are already given. The question then arises as to how these partitions can be obtained in such a way that the interaction between sub-domains is minimal (e.g. see Pavlis & Papageorgiou, 1999; Logi & Ritchie, 2002). This manuscript presents a heuristic that can be used to fill exactly this gap in decentralized traffic management. Other applications in the transportation field exist as well. Generally, DD can be used in transportation problems in which it is beneficial to decompose a large problem into smaller problems to determine its solution, ex. continuum traffic equilibrium problems that are solved via finite element methods (e.g. see Wong et al., 1998; Nguyen, 2006) and certain classes of shortest path problems that can be solved via a series of systems of linear equations (e.g. see Ng & Sathasivan, 2014).

1.2 Method and Procedure

The Shortest Distance Decomposition Algorithm (SDDA) is a multi-step algorithm which will be discussed in detail in Chapter 2. This chapter is based on the recently published Journal of Intelligent Transportation Systems Paper "Large scale network partitioning for decentralized traffic management and other transportation applications" (Johnson et al., 2016). The chapter describes an algorithm which will be used to partition several real-world, large-scale transportation networks. In fact, it will be demonstrated that the current best general purpose network partitioning algorithm is typically not as good as the one proposed, which has been developed with civil engineering applications in mind. To demonstrate this claim, the effectiveness (in terms of minimizing the number of system boundary nodes) of the SDDA is compared to the popular and well respected METIS partitioning algorithm (Karypis & Kumar, 1998). METIS is a serial software package used to partition large irregular graphs and large meshes, and for computing fill-reducing orderings of sparse matrices. It was developed at the Department of Computer Science and Engineering at the University of Minnesota and it is freely distributed. "Metis" was an ancient Greek goddess of wisdom and knowledge in Greek mythology. The goal of their algorithm is "to partition the vertices of a graph in p roughly equal parts, such that the number of edges connecting vertices in different parts is minimized" (Karypis & Kumar, 1998). These two objectives are essentially identical to that of the SDDA. The version of METIS used in the comparisons is Version 5.10.

The manuscript will then demonstrate, in Chapter 3, other advantages of effective domain partitions. Built upon the SDDA is a second algorithm called The Domain Decomposition Based Shortest Path Algorithm, or DDSP. The DDSP algorithm uses

partitions created by the SDDA algorithm to solve the shortest path problem. As a benchmark in this test, the classical Dijkstra method is used to solve the problem and then the results are compared with that of the DDSP algorithm. It is shown that by breaking up the original network into smaller sub-networks and using the information about how each sub-network is connected, one can actually find the correct solution much more efficiently. The algorithm is further explained and results are presented to support this claim.

Finally, Chapter 4 is dedicated to another application of the SDDA. As previously mentioned, one of the uses of METIS is to produce fill reducing orderings. This means that the algorithm reorders a set of data such that, when the network is factorized, it reduces the amount of fill-in terms during Gaussian elimination. SDDA offers this functionality as well, and in fact, again outperforms METIS for the majority of the networks tested. Although the data used to measure the amount of fill-in is from some of the same transportation networks used in other tests presented, these orderings can be exploited in numerous fields of civil engineering where a set of simultaneous linear equations must be solved.

CHAPTER 2

THE SHORTEST DISTANCE DECOMPOSITION ALGORITHM

2.1 Introduction

The solution to the shortest path problem has been well documented in literature, over the years. Several Label Setting (Dijkstra, 1959) or Label Correcting (Bellman, 1956; Glover et al., 1985) Algorithms have been developed. This work utilizes a variant of Glover's polynomial bounded LCA, also referred to as the Polynomial Label Correcting Algorithm (P-LCA) (e.g. see Allen, 2013). This particular algorithm will be reviewed, using a sparse matrix storage scheme to store the network connectivity information. It is to be noted that the proposed decomposition heuristic can be used with any other shortest path algorithm.

Before beginning the P-LCA computation, the network information must be stored in an efficient manner. This is done by using the efficient sparse storage scheme discussed in Lawson et al. (2013) and in Nguyen (2006). By using this method, only the non-zero values/locations in the connectivity matrix are stored, cutting down on both computational time and memory requirements. Suppose a five node/14 link network is to be analyzed. The network contains the following topology and coefficient matrix describing the network connectivity. However, the actual link cost values are not required to be stored by the SDDA algorithm. For the purpose of partitioning, the algorithm assumes that all links have a cost of unity. (Note: Cij in matrix A indicates the cost associated with traveling from "inode" to "jnode", where "inode" represents the source node and "jnode" represents the destination node).

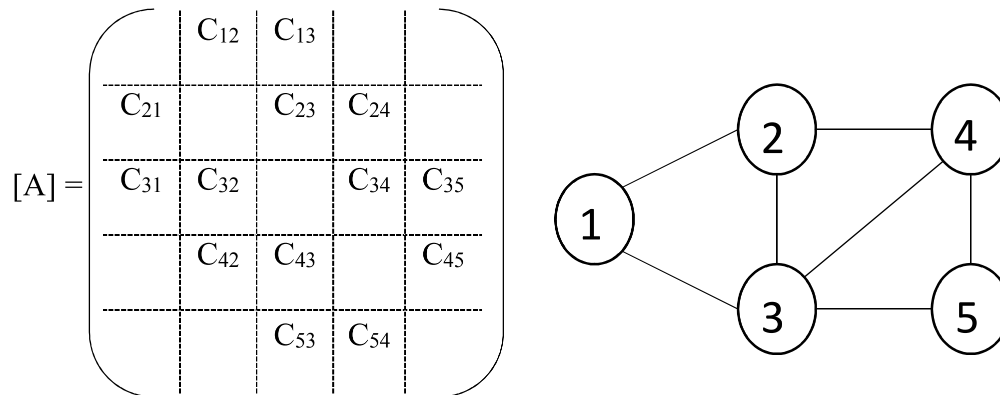


Figure 2-1: Network Topology and Connectivity Matrix for 5 Node Network

As part of our proposed SDDA, a basic Shortest Path (SP) algorithm needs be employed. While any existing SP algorithm can be utilized (such as the LCA, polynomial LCA, Dijkstra, etc.), the polynomial LCA algorithm (Lawson et al., 2013) was selected for this work, since it has been proven to be more efficient than the classical LCA. For the readers' convenience, the main ideas of the Polynomial LCA can be summarized as follows:

1. Use two arrays, {NOW} and {NEXT}, to track and process nodes as they are updated rather than cycling through every node in the network (for each iteration).
2. If the answer for "is d(jnode) > d(inode) + c(inode, jnode)" is YES, and there are more outgoing links from inode, then we must:
    a. Update both array {d} and array {predes}
    b. Include jnode in the list {NEXT}
3. If the answer for "is d(jnode) > d(inode) + c(inode, jnode)" is YES, and there are no more outgoing links from inode, then we must:

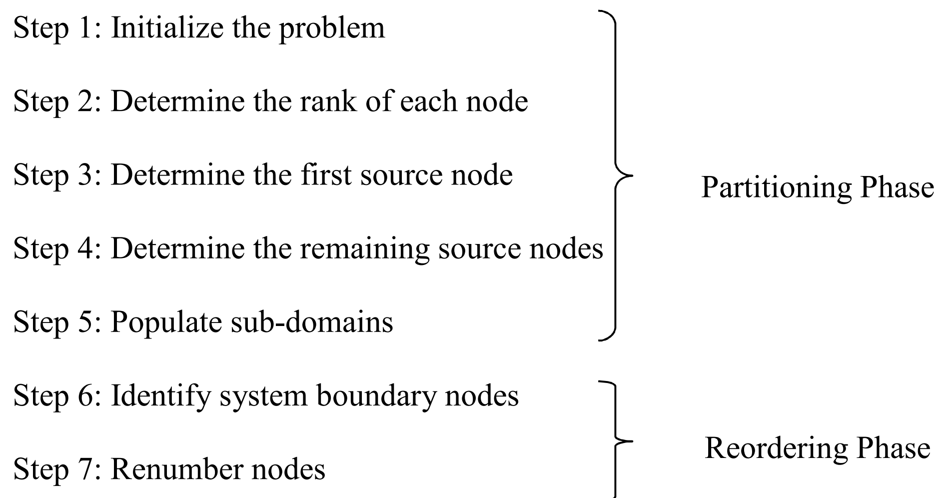
   a. Update both array {d} and array {predes}. These two arrays contain the updated Shortest Time (ST), and its corresponding Shortest Path (SP), respectively.
   b. Include jnode in the list {NEXT}
   c. Remove inode from the list {NOW}

4. If the answer for "is d(jnode) > d(inode) + c(inode, jnode)" is NO, and there are more outgoing links from inode, then we must:
   a. NOT update the arrays {d}, {predes}, {NEXT}, or {NOW}
5. If the answer for "is d(jnode) > d(inode) + c(inode, jnode)" is NO, and there are no more outgoing links from inode, then we must:
   a. NOT update the arrays {d}, {predes}, or {NEXT}
   b. Remove inode from the list {NOW}
6. If the array {NOW} is EMPTY, but the array {NEXT} is NOT empty, we must reset {NOW} = {NEXT} & {NEXT} = {empty}, and repeat the process.
7. If both {NOW} and {NEXT} are EMPTY, then convergence has been achieved by P-LCA

2.2 The Shortest Distance Decomposition Algorithm

The SDDA consists of seven steps. In these seven steps, the algorithm reads and manipulates a user provided set (transportation network) of data and efficiently partitions it into a predefined number of smaller sub-domains, with the goal to minimize the total number of system boundary nodes (for minimizing communication time among different processors in a parallel computer environment) and to ensure that the size of each sub-

domain is approximately equal. The key to the algorithm's success in keeping the number of system boundary nodes small lies in the fact that 1) each sub-domain's source node is selected such that it will be as far as possible from the previously selected source nodes (cf. Step 4), and that 2) nearby nodes are gradually incorporated into each sub-domain (cf. Step 5). Finally, the algorithm renumbers the nodes so that the resulting sub-domains' interior nodes are completely uncoupled. The steps of the algorithm are as follows:



Steps 1 through 5 are used to partition the provided large domain into smaller sub-domains. If subsequent work on the network is required, Steps 6 and 7 can then be used to reorder the nodes of the network into a convenient form in which parallel processing can be exploited. The algorithm begins with simple input from the user: namely, the network topology and the number of sub-domains they wish to partition the network into. From this point, Steps 2, 3, and 4 are used to find the starting source node of each sub-domain. These (starting) source nodes are then used as the basis for the domain population process of Step 5 to populate the remaining nodes for each subdomain. Now,

the partitioning is complete. Steps 6 and 7 are then performed, simply to transform the original network topology into a new reordered topology to utilize parallel processing in order to more efficiently perform future operations on the network. A more detailed, step-by-step discussion is provided as follows, through use of a simple numerical example.

Step 1: Initialize the Problem

To begin, the user must provide the following information:

1.1. Network Connectivity (or topology) Matrix, via text file. The first column of the text file represents the "head" or "source" node, while the second column represents the "tail" or "destination" node.

1.2. The number of Sub-Domains or Number of Processors (NP) that the user wishes to divide the network into. For this example, assume NP = 3. The network to be partitioned is provided by Figure 2-2.

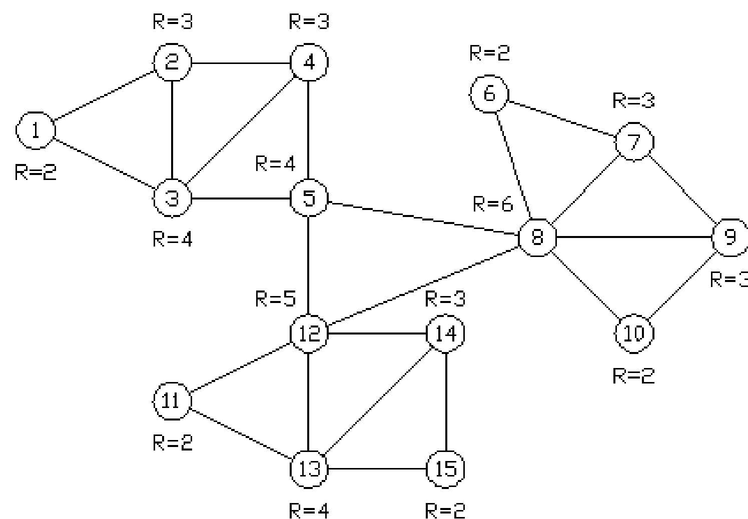


Figure 2-2: Network Connectivity with Nodal Ranks

Step 2: Determine the rank, R, of each Node

In this algorithm, the rank R of each node is defined as the number of links connected to the node in question. By analyzing the manipulated connectivity information produced by Step 1, R can be determined by simply summing the number of occurrences that a given inode occurs in the data set, or by the total number of nodes connected to inode.

Step 3: Determine the First Source Node

Though its calculation is trivial, the first source node is the basis for the remaining steps in the algorithm. The first source node is simply the lowest ranking node in the system. Generally speaking, nodes with few connections often lie on the periphery of a domain. By selecting the lowest ranking node, it is anticipated that a remotely located node along the network's periphery has been identified. If there are multiple nodes of the same rank, the algorithm arbitrarily selects the first lowest ranked node that it encounters. In examining Figure 2-2, nodes 1, 6, 10, 11, and 15 all share the lowest rank of 2. Node 1 is then selected as the first source node, as it is the lowest numbered of this group.

Step 4: Determine the Remaining Source Nodes

The remaining source nodes are found by performing the following steps:

4.1. Refer to the network connectivity information stored in Step 1.

4.2. Perform the Modified P-LCA calculation with the node identified in Step 3 as the source node. In this process, the algorithm computes the distance from this node to all other nodes in the system. Note: For the Modified P-LCA Method, only update the arrays {d}, {NOW}, and {NEXT}; the array {predes}, which stores the SP information, is not needed for the SDDA, and its calculation will

unnecessarily use system memory and therefore should be omitted. The results of this step are provided in the proceeding table, Table 2-1.

Table 2-1: Shortest Path Information for Source Node 1

Node	Distance From 1	Total Distance
1	0	0
2	1	1
3	1	1
4	2	2
5	2	2
6	4	4
7	4	4
8	3	3
9	4	4
10	4	4
11	4	4
12	3	3
13	4	4
14	4	4
15	5	5

4.3. The next source node is found by identifying the node which is farthest from the first source node. Should there be multiple nodes with an identical distance, select the furthest node with the lowest rank. If there is yet again a tie, simply select the lowest numbered node with the furthest distance and the lowest rank. Examining Table 2-1, one can easily see that node 15 should be selected as the second source node.

4.4. Next, perform the P-LCA calculation for the node identified in Step 4.3. This process is identical to what is outlined previously, except for two modifications. Instead of only finding the node with the largest distance from the current source node, sum the distances for all source nodes to find an average distance from all

previous source nodes in an effort to ensure that the next source node to be selected is not adjacent any previously selected source node. If there is a tie among distance and rank, select the node which has the smallest range of the distances from all of the other source nodes. By selecting the node with the smallest range, it is ensured that the node to be selected is more equidistant from the previously selected source nodes. This process should continue until NP source nodes have been identified.

Table 2-2: Information for Source Nodes 1 and 15 Shortest Path

Node	Distance From 1	Distance From 15	Total Distance	Range
1	0	5	5	5
2	1	5	6	4
3	1	4	5	3
4	2	4	6	2
5	2	3	5	1
6	4	4	8	0
7	4	4	8	0
8	3	3	6	0
9	4	4	8	0
10	4	4	8	0
11	4	2	6	2
12	3	2	5	1
13	4	1	5	3
14	4	1	5	3
15	5	0	5	5

Examining Table 2-2, it is easily seen that nodes 6, 7, 9, and 10 each share the largest total distance of 8 and should be considered for the final source node. The arithmetic range for each node (the absolute value of the difference between columns 2 and 3) must be considered in order to break the tie. Since each of these nodes also shares the same value for range, select the first node encountered: namely, node 6.

Step 5: Populate Sub-Domains

Each sub-domain begins with its respective source node. Sub-domains 1, 2, and 3 contain nodes 1, 15, and 6, respectively (as previously found in Steps 3 and 4). Continue by assigning nodes to each sub-domain until all nodes in the system have been allocated. It is important to note that when assigning additional nodes to a particular sub-domain, the algorithm must follow these three rules:

5.1. Each sub-domain must be populated in a simultaneous fashion. However, a given sub-domain may not be able to add a node because of the rule noted in Step 5.3. In this instance, simply use a value of "0" as a placeholder for that iteration's specific sub-domain.

5.2. The process used to add a node to each sub-domain is nearly identical to that of Step 4.3, with one modification. Rather than select the node *farthest* from the source node, add the node which is closest to the original source node. This change is made to ensure that each node to be added to a particular sub-domain is clustered around its source node. The algorithm makes this decision based on the results of the Modified P-LCA computations. For a first tie breaker, select the node which has the lowest nodal rank. If there are multiple nodes which share this value, default to the arithmetic range of nodal distances. However, another modification is made. Select the node with the largest range, with the assumption that it will be close to the source node. If this still results in a tie, arbitrarily select the lower numbered node.

5.3. In order to minimize the number of system boundary nodes generated by the algorithm, it is important to ensure that the domain being built is a continuous

domain. To guarantee this, the algorithm requires the node selected in Step 5 to be directly connected to at least one of the other nodes already in the sub-domain. By following this rule, along with the rule established by Step 5.1, it is guaranteed that each sub-domain will be continuous, which generally results in fewer system boundary nodes. This is due to the fact that the number of exposed edges in the network is minimized (refer to Step 6 for a discussion on system boundary nodes). If there is not a node with a direct connection to the existing nodes available, the algorithm will not add a node to the sub-domain during this iteration. Rather, it simply uses a value of “0” as a placeholder (previously mentioned in Step 5.1). However, a node may be able to be added in subsequent iterations, as nodes are added to other sub-domains. To continue the illustrative example, once Steps 5.1 through 5.3 are complete, the algorithm will provide the partitioned network as given by Table 2-3.

Table 2-3: Populated Sub-Domains

Sub-Domain 1	Sub-Domain 2	Sub-Domain 3
1	15	6
2	14	7
3	13	8
4	11	10
5	12	9

Step 6: Identify System Boundary Nodes

When two nodes belong to different sub-domains, and there is a link connecting them, they are identified as System Boundary Nodes (or SBN). Once identified, these nodes are subsequently used in Step 7. SBN are found as follows:

6.1. The SDDA compares the inode and jnode of each link. If both belong to the same sub-domain, they are considered "interior" nodes. If the nodes are found in different sub-domains, they both must be considered SBN.

6.2. The algorithm records which nodes are found to be boundary nodes and creates a vector containing a list of system boundary nodes which is sorted in ascending order for convenience.

As previously mentioned, the primary goal of the SDDA is to minimize the number of SBN. The authors of the METIS algorithm (Karypis & Kumar, 1998) note they attempt to minimize the number of edges connecting different sub-domains. Rather than count these edges, the SDDA counts the number of nodes which are directly connected to these edges. In short, both (METIS and SDDA) of the algorithms are striving to obtain the same goals: 1) to have partitions approximately equal in size, and 2) to minimize the total number of SBN. The first criterion can be trivially satisfied; if a network has N nodes and is subsequently partitioned into a given number of smaller sub-domains, simply assign each sub-domain approximately N/NP nodes. Although more difficult to achieve, the second criterion yields greater computational efficiency. For the aforementioned reasons, the SDDA puts more emphasis on the second criterion. For the example shown in Figure 2-2, the SBN can be identified as nodes 5, 8, and 12.

Step 7: Reorder Nodes

This step reorders the nodes in such a manner that, when the system's reordered coefficient matrix is plotted, a decoupling of the sub-domains is present. This is a very desirable property; the system is already in a convenient form such that the problem may

be solved via parallel processing techniques through the partitioned sub-domains. This property is achieved via the following steps:

7.1. Each sub-domain's nodes is ordered consecutively, starting with the first sub-domain. This then allows each sub-domain to be completely independent of the others.

7.2. To achieve this decoupled state, remove the boundary nodes and place them at the end of the numbering scheme, making them the highest numbered nodes.

It is noted that Steps 6 and 7 are not requirements to partition the network. Rather, they are used to reorder the network for subsequent operations on the data set. For the network shown in Figure 2-2, the results from Step 7 are shown in Table 2-4.

Table 2-4: New Node Numbering Scheme

New Node Number	Old Node Number	Sub-Domain
1	1	1
2	2	1
3	3	1
4	4	1
5	15	2
6	14	2
7	13	2
8	11	2
9	6	3
10	7	3
11	10	3
12	9	3
13	5	All (SBN)
14	8	All (SBN)
15	12	All (SBN)

After this renumbering has occurred, the decoupling effect becomes very obvious. Figure 2-3 shows this graphically for the much larger Chicago network. This Step is

critical for allowing each sub-domain to be operated on simultaneously. Each sub-domain's interior nodes are completely independent of every other sub-domain's interior nodes, as shown, allowing independent, parallel computations. The connectivity between the sub-domain's boundary nodes then allows this information to be pulled back together and assembled into the complete solution. This image also graphically shows that the number of SBN (lower right diagonal block) is very small when compared to the entire network, thus reducing the overall assembly time of each sub-domain's individual solution back into the total solution for the problem.

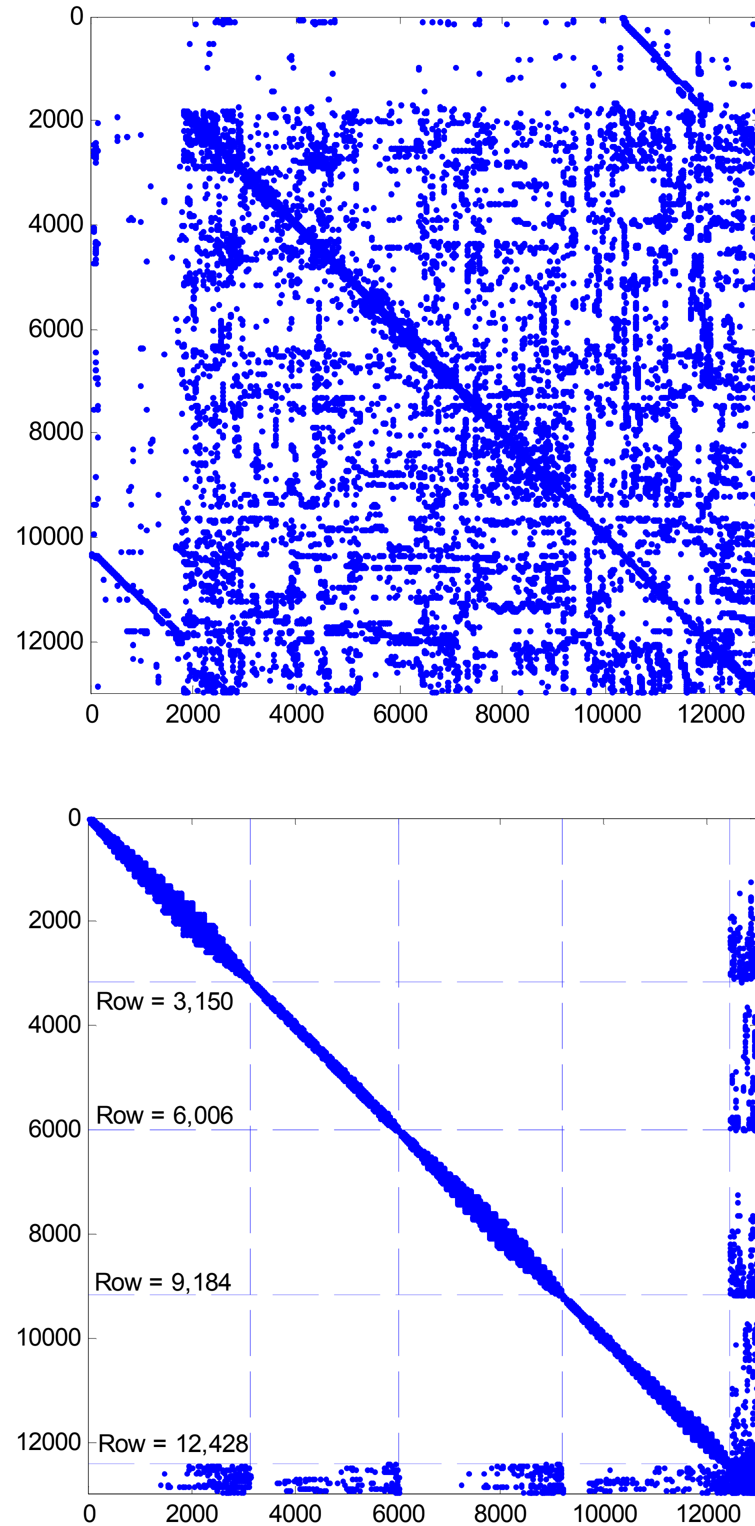


Figure 2-3: Partitioning of the Chicago Network for NP = 4.

Top: Original Network Connectivity; Bottom: SDDA Partitioned Network

2.3 Comparisons with METIS Using Real-World Transportation Networks

In this section, eight additional networks are added (in addition to the 15 node network, yielding nine total networks). Of these, seven are actual road networks which were retrieved from http://www.bgu.ac.il/~bargera/tntp/. These networks were added in order to assess the performance of the SDDA. To this end, the ensuing number of system boundary nodes are compared to the results obtained from METIS, opined by many researchers as the current worldwide standard in network partitioning. Each of the nine networks has been tested for two, three, and four sub-domains, resulting in 27 different examples. The results are summarized in Table 2-5. For example, when investigating the Austin, Texas network (with 7,388 nodes and 18,956 links), when the number of sub-domains is four, the number of system boundary nodes resulting from the SDDA is only 305. Conversely, METIS yields 1,221 boundary nodes, which represents an increase of 400%. As can be seen, out of these 27 cases, the SDDA outperforms METIS 21 times (78%). Moreover, it should be noted that, on average, METIS results in approximately 215% more system boundary nodes than the number returned by the SDDA. When only the hypothetical and smaller networks are considered (limiting the study to only the Austin, Chicago, and Philadelphia networks), this reduction becomes even more dramatic: METIS partitions result in 238% more boundary nodes than the SDDA, on average.

METIS outperformed our algorithm in six of the 27 (22%) test cases. When METIS provided better results, it yielded nearly half the number of boundary nodes obtained from SDDA. It should be noted that three of these six occurrences were found for the Winnipeg network, with the other occurrences resulting in virtually a tie between

the SDDA and METIS algorithms. It can be assumed that the disparity between the SDDA and METIS in the Winnipeg network is purely due to the specific network topology of that network. Indeed, being a heuristic algorithm, it cannot be guaranteed that SDDA will outperform METIS for a specific network. The main conclusion here is that SDDA outperforms METIS more often than not, when applied to some of the most common networks appearing in the transportation literature. The 21 instances in which the SDDA outperformed METIS are highlighted in Table 2-5.

Table 2-5: System Boundary Node Comparison between the SDDA and METIS

	Number of System Boundary Nodes						
	# Nodes/Links	NP = 2		NP = 3		NP = 4	
		SDDA	METIS	SDDA	METIS	SDDA	METIS
11 Node	11/32	**6**	8	**7**	8	**7**	9
15 Node	15/48	**3**	7	**3**	3	**8**	12
Sioux Falls	24/76	**10**	10	17	13	20	19
Anaheim	416/914	**45**	81	**57**	162	**66**	160
Barcelona	930/2,522	**92**	206	**152**	243	**189**	303
Winnipeg	1,040/2,836	81	51	133	63	156	108
Austin	7,388/18,956	**265**	878	**276**	872	**305**	1,221
Chicago	12,979/39,018	**240**	626	**407**	794	**551**	1,347
Philadelphia	13,389/40,003	**370**	773	413	393	**523**	1,080

It would be an incomplete analysis if the computational times of both algorithms were not compared. To compare computational time, Old Dominion University's High Performance Computing Center computer hardware TURING cluster (node #064) was used to run METIS (FORTRAN shell program which calls METIS, written in C) and the SDDA (written in the MATLAB environment), with the following characteristics:

Number of Cores = 20

RAM = 128 GB

CPU = Intel Xeon E5-2670 v2 @2.50 GHz

Table 2-6 provides a comparison of the computational time required to perform the partitioning of each network tested.

Table 2-6: Partitioning Time Comparison - METIS and The SDDA

	Solution Time (s)						
	# Nodes/Links	NP = 2		NP = 3		NP = 4	
		SDDA	METIS	SDDA	METIS	SDDA	METIS
11 Node Network	11/32	0.050	0.002	0.050	0.001	0.050	0.001
15 Node Network	15/48	0.051	0.002	0.050	0.001	0.051	0.001
Sioux Falls Network	24/76	0.051	0.002	0.051	0.001	0.051	0.002
Anaheim Network	416/914	0.076	0.003	0.078	0.003	0.077	0.004
Barcelona Network	930/2,522	0.109	0.002	0.111	0.003	0.117	0.004
Winnipeg Network	1,040/2,836	0.118	0.002	0.119	0.002	0.125	0.003
Austin Network	7,388/18,956	0.846	0.006	0.934	0.009	0.926	0.010
Chicago Network	12,979/39,018	1.864	0.008	1.729	0.012	1.779	0.013
Philadelphia Network	13,389/40,003	2.179	0.080	1.917	0.026	1.933	0.013

From Table 2-6, it can be seen that METIS outperforms the SDDA from a computational time perspective. However, this small difference is not an issue for the following reasons:

1. The computational requirement for SDDA remains very low, around 2 seconds in Table 2-6, and
2. Partitioning a network is typically a pre-processing step, after which more serious computations are carried out that consume much longer computation times (e.g. see Nguyen, 2006). It should be noted the comparisons in Table 2-6 might be skewed in METIS' favor for the following reasons:

a. It is generally accepted that MATLAB (in which SDDA has been written in) is significantly slower than compiled languages, such as FORTRAN or C, which METIS is written in (e.g. see Kouatchou, 2009).
b. The built-in FORTRAN time measurement function (etime) for collecting METIS computation time fluctuated very significantly during our tests, and might therefore not be very reliable (when running the SDDA in the MATLAB environment no such observation was made).

In light of these observations, it is fair to conclude that SDDA is an efficient algorithm.

In addition to the proceeding information, Figure 2-4 (METIS partitioning results) and Figure 2-5 (SDDA partitioning results) are provided to afford the reader graphical partitions of the 15 node network. Similarly, Figure 2-6 and Figure 2-7 show the same partitioning results for the 11 node network. This helps to graphically depict the difference between the partitions provided by each of the two algorithms.

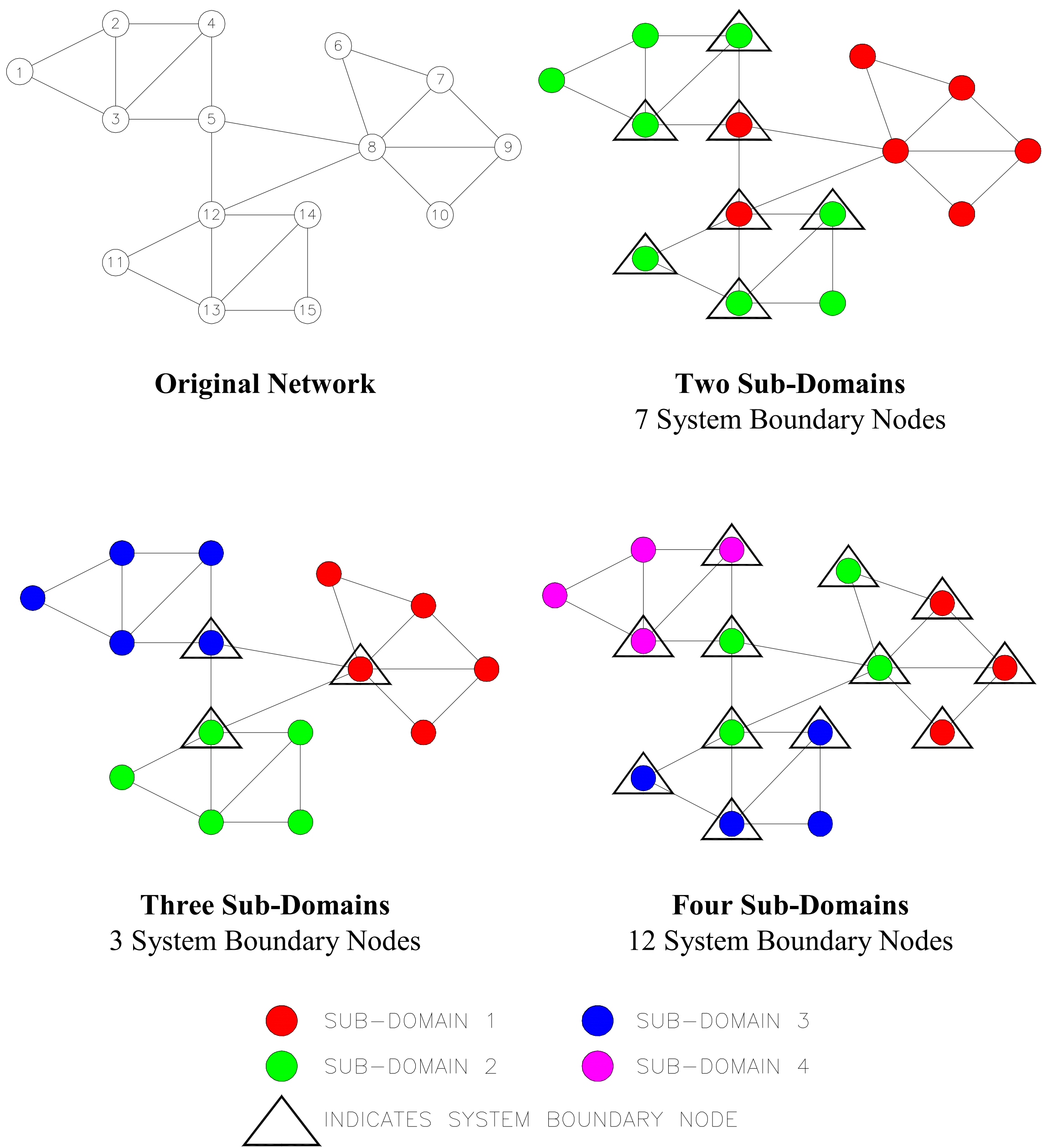
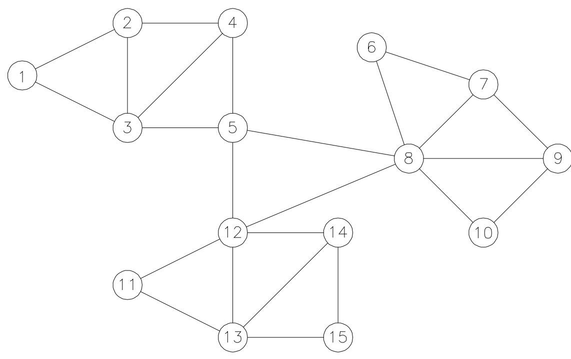
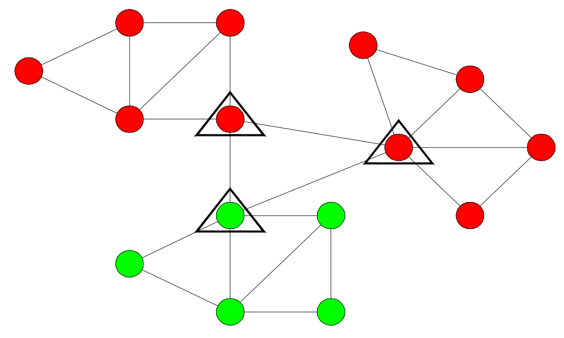


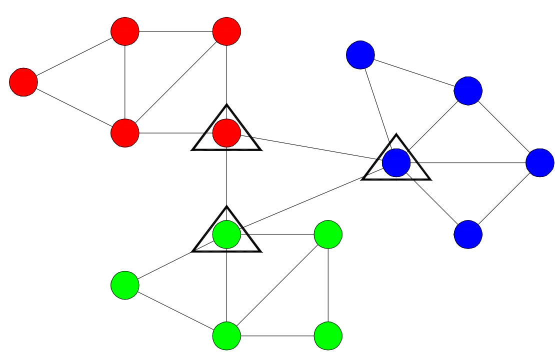
Figure 2-4: METIS Partitioning of the 15 Node Network



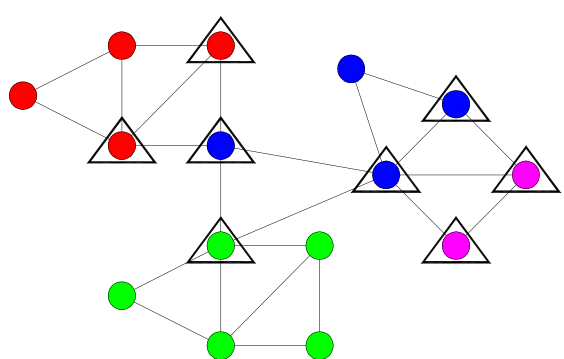
Original Network



Two Sub-Domains
3 System Boundary Nodes



Three Sub-Domains
3 System Boundary Nodes



Four Sub-Domains
8 System Boundary Nodes

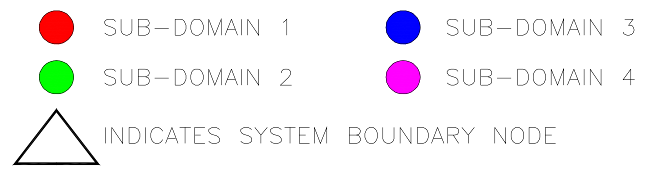
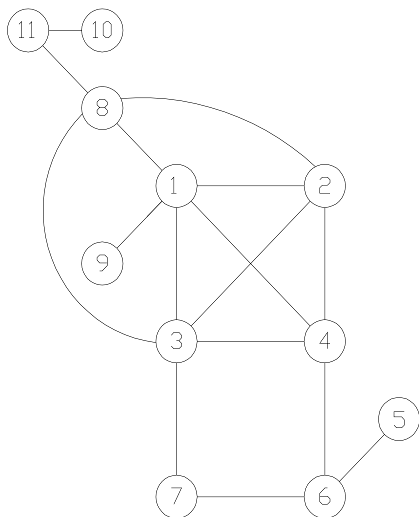
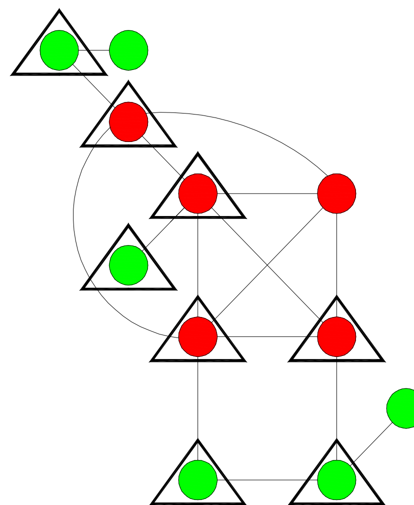


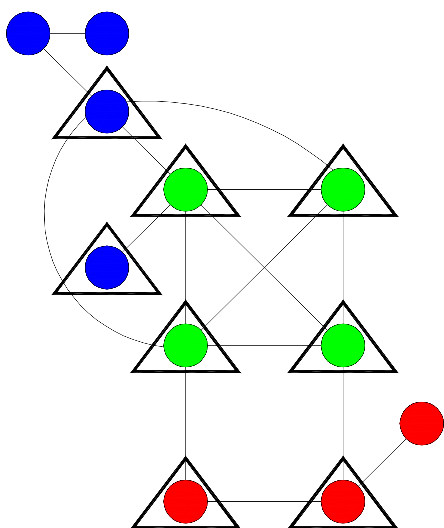
Figure 2-5: SDDA Partitioning of the 15 Node Network



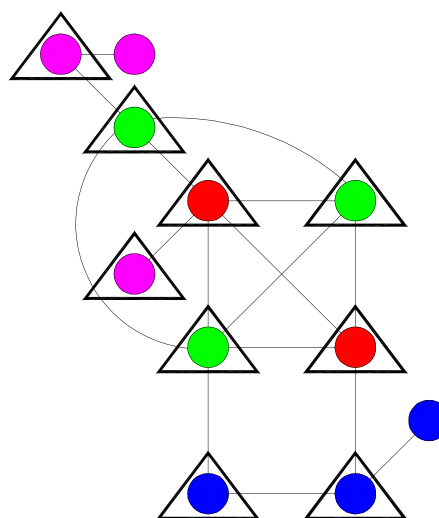
Original Network



Two Sub-Domains
8 System Boundary Nodes



Three Sub-Domains
8 System Boundary Nodes



Four Sub-Domains
9 System Boundary Nodes

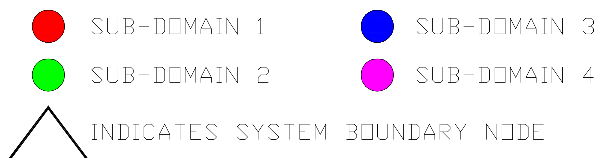
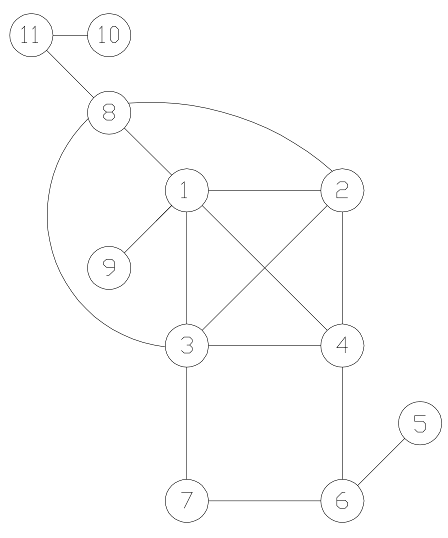
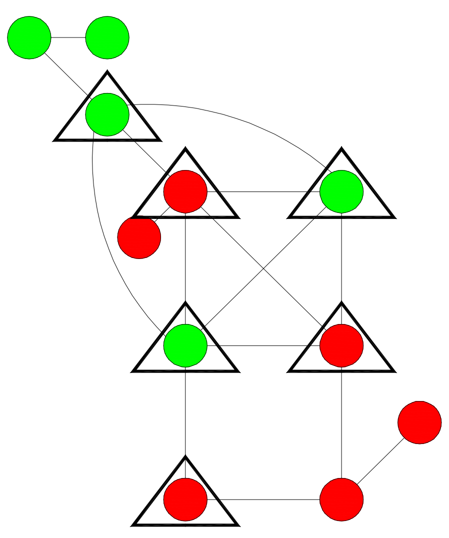


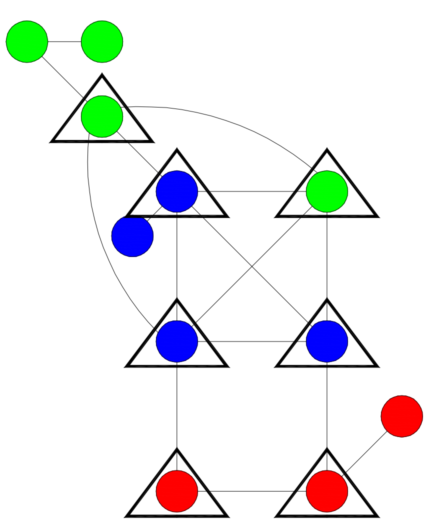
Figure 2-6: METIS Partitioning of the 11 Node Network



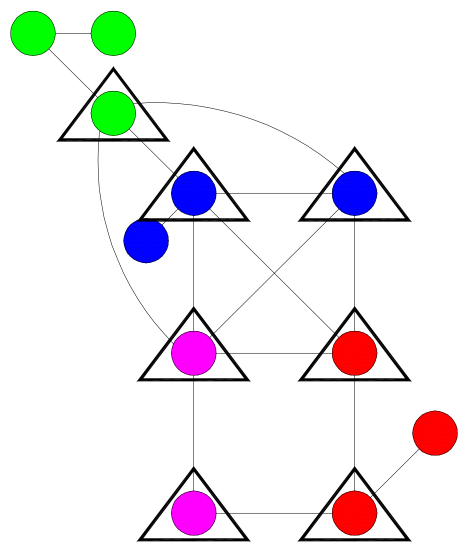
Original Network



Two Sub-Domains
6 System Boundary Nodes



Three Sub-Domains
7 System Boundary Nodes



Four Sub-Domains
9 System Boundary Nodes

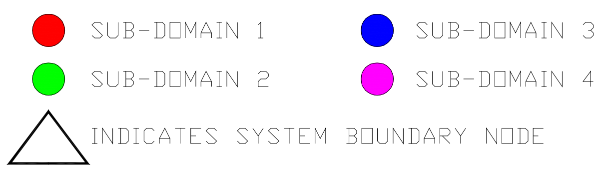


Figure 2-7: SDDA Partitioning of the 11 Node Network

As previously indicated, the secondary objective of the SDDA is to balance the number of nodes per sub-domain. Results have been prepared to demonstrate the uniformity of the sub-domain sizes achieved. Clearly, the target percentage of nodes to be assigned to each sub-domain is 50%, 33%, and 25% when NP = 2, 3 and 4, respectively. Comparing Table 2-7 and Table 2-8, both METIS and the SDDA result in sub-domains of approximately equal size, when considering large-scale transportation networks.

Table 2-7: Size of each sub-domain provided by METIS

	Size of Each Sub-Domain (Number of Nodes) – METIS Partition					
	Number of Nodes	**NP**	**P1 Nodes/% of Total**	**P2 Nodes/% of Total**	**P3 Nodes/% of Total**	**P4 Nodes/% of Total**
11 Node/32 Link Example	11	NP = 2	5 / 45	6 / 54	N/A	N/A
		NP = 3	3 / 27	4 / 36	4 / 36	N/A
		NP = 4	2 / 18	3 / 27	3 / 27	3 / 27
15 Node/48 Link Example	15	NP = 2	7 / 47	8 / 53	N/A	N/A
		NP = 3	5 / 33	5 / 33	5 / 33	N/A
		NP = 4	3 / 20	4 / 27	4 / 27	4 / 27
Sioux Falls Network	24	NP = 2	12 / 50	12 / 50	N/A	N/A
		NP = 3	8 / 33	8 / 33	8 / 33	N/A
		NP = 4	6 / 25	6 / 25	6 / 25	6 / 25
Anaheim Network	416	NP = 2	208 / 50	208 / 50	N/A	N/A
		NP = 3	138 / 33	139 / 33	139 / 33	N/A
		NP = 4	104 / 25	104 /25	104 /25	104 /25
Barcelona Network	930	NP = 2	465 / 50	465 / 50	N/A	N/A
		NP = 3	310 / 33	311 / 33	309 / 33	N/A
		NP = 4	232 / 25	233 / 25	232 / 25	233 / 25
Winnipeg Network	1,040	NP = 2	520 / 50	520 / 50	N/A	N/A
		NP = 3	346 / 33	347 / 33	347 / 33	N/A
		NP = 4	260 / 25	260 / 25	260 / 25	260 / 25
Austin Network	7,388	NP = 2	3694 / 50	3694 / 50	N/A	N/A
		NP = 3	2462 / 33	2463 / 33	2463 / 33	N/A
		NP = 4	1847 / 25	1847 / 25	1848 / 25	1846 / 25
Chicago Network	12,979	NP = 2	6488 / 50	6491 / 50	N/A	N/A
		NP = 3	4327 / 33	4325 / 33	4327 / 33	N/A
		NP = 4	3244 / 25	3246 / 25	3244 / 25	3245 / 25
Philadelphia Network	13,389	NP = 2	6694 / 50	6695 / 50	N/A	N/A
		NP = 3	4463 / 33	4463 / 33	4463 / 33	N/A
		NP = 4	3348 / 25	3346 / 25	3348 / 25	3347 / 25

Table 2-8: Size of each sub-domain provided by the SDDA

	Size of Each Sub-Domain (Number of Nodes)					
	Number of Nodes	**NP**	**P1 Nodes/% of Total**	**P2 Nodes/% of Total**	**P3 Nodes/% of Total**	**P4 Nodes/% of Total**
11 Node	11	NP = 2	6 / 55	5 / 45	N/A	N/A
		NP = 3	3 / 27	4 / 36	4 / 36	N/A
		NP = 4	3 / 27	3 / 27	3 / 27	2 / 18
15 Node	15	NP = 2	10 / 67	5 / 33	N/A	N/A
		NP = 3	5 / 33	5 / 33	5 / 33	N/A
		NP = 4	4 / 27	5 / 33	4 / 27	2 / 13
Sioux Falls Network	24	NP = 2	12 / 50	12 / 50	N/A	N/A
		NP = 3	8 / 33	8 / 33	8 / 33	N/A
		NP = 4	5 / 21	7 / 29	6 / 25	6 / 25
Anaheim Network	416	NP = 2	212 / 51	204 / 49	N/A	N/A
		NP = 3	153 / 37	149 / 36	114 / 27	N/A
		NP = 4	106 / 25	107 / 26	104 / 25	99 / 24
Barcelona Network	930	NP = 2	467 / 50	463 / 50	N/A	N/A
		NP = 3	310 / 33	315 / 34	305 / 33	N/A
		NP = 4	249 / 27	235 / 25	240 / 26	206 / 22
Winnipeg Network	1,040	NP = 2	508 / 49	532 / 51	N/A	N/A
		NP = 3	351 / 34	358 / 34	331 / 32	N/A
		NP = 4	289 / 28	268 / 26	273 / 26	210 / 20
Austin Network	7,388	NP = 2	3,788 / 51	3,600 / 49	N/A	N/A
		NP = 3	2,812 / 38	2,651 / 36	1,925 / 26	N/A
		NP = 4	1,690 / 23	2,077 / 28	1,602 / 22	2,019 / 27
Chicago Network	12,979	NP = 2	6,743 / 52	6,236 / 48	N/A	N/A
		NP = 3	4,312 / 33	4,409 / 34	4,258 / 33	N/A
		NP = 4	3,301 / 25	2,954 / 23	3,321 / 26	3,403 / 26
Philadelphia Network	13,389	NP = 2	7,502 / 56	5,887 / 44	N/A	N/A
		NP = 3	4,341 / 32	4,796 / 36	4,252 / 32	N/A
		NP = 4	3,663 / 27	3,042 / 23	3,171 / 24	3,513 / 26

2.4 Conclusions

Domain decomposition is used frequently for solving numerous large-scale engineering and science problems in an efficient manner. To make the most out of this process, it has

been shown that one should aim to minimize the number of system boundary nodes. By doing so, one can more efficiently process the sub-domains through the use of parallel processing. Although various transportation researchers have hinted at the use of DD (for example, in decentralized traffic management), the assumption is always made that the partition is given. This manuscript presents a simple, efficient and effective heuristic to decompose a network into a predefined number of interconnected sub-domains. The algorithm partitions in such a way that the number of system boundary nodes is minimized (first priority), and the size of each sub-network is similar (second priority). The proposed method has been compared with the METIS algorithm, which is believed by many to be the most widely used algorithm in graph partitioning worldwide. It should be noted that incorporating METIS into users' application codes will require the users to download and install many subroutines/functions. The developed SDDA (written in the popular MATLAB computer environment), on the other hand, can be easily incorporated into general users' application codes, as it is a simple and short MATLAB script. Using large-scale, real-world transportation test networks, it was found that the proposed algorithm performed significantly better than METIS. The SDDA outperformed METIS in 21 of 27 tested examples. On average, the SDDA provided (approximately) 42% of the total number of system boundary nodes provided by METIS, when considering large-scale networks.

CHAPTER 3

THE DOMAIN DECOMPOSITION BASED SHORTEST PATH ALGORITHM

3.1 Introduction

The shortest path (SP) problem in transportation applications has been the subject of extensive research, resulting in a large number of scientific publications. Dealing with real-world, large-scale networks, various parallel procedures (based on label correction, Dijkstra, bi-direction, A*, etc.) have been proposed (Foster, 2003; Chabini & Ganugapati, 2002; Habbal et al., 1994; Allen, 2013; Nguyen, 2006). In general, most of the existing parallel procedures for the SP problems were based on either of the following ideas:

1. **Destination-based Decomposition:** In this strategy, one assigns each processor to handle blocks of rows (or columns) of the given network's topology (Foster, 2003; Habbal et al., 1994; Allen, 2013). This parallel implementation is trivial and has been adopted in earlier works (Foster, 2013; Chabini, 1998; Ziliaskopoulos et al., 1997). For most (major) transportation applications, one is required to find the SP (and its corresponding shortest time, ST) from all source nodes to multiple destination nodes. While reasonably good speed-up can be achieved, this approach may not be desirable when:
    a. many processors are available for just a few destination nodes, and/or
    b. computer memory needs to be managed conveniently and efficiently
2. **Network-topology-based Decomposition:** In this strategy, one breaks the original problem into a series of smaller sub-domains (Chabini & Ganugapati, 2002; Nguyen, 2006). Each sub-domain is then independently analyzed by its

assigned processor, and finally each sub-domains' results have to be integrated in order to produce the final solution to the original network.

The idea of breaking up a large problem into smaller sub-problems (or sub-domains) is not new. This concept of sub-structuring originated, and was subsequently applied, in the aerospace and structural engineering arenas several decades ago (Przemieniecki, 1985). Since, sub-structuring methods have been studied, refined, and extended to many other fields of research, such as structural dynamics, generalized Eigen-value problems, etc. (Nguyen, 2006). In these earlier applications, however, the purpose of sub-structuring for computer implementation was related to solving systems of simultaneous linear equations. More recently, the sub-structuring formulation (or domain decomposition formulation) has been applied to real-world, large-scale transportation networks where the objective is to find the SP from some number of source nodes to some number of destination nodes (Chabini, 2002).

In order to design and to implement the proposed Domain Decomposition based Shortest Path (DDSP) algorithm efficiently on large-scale problems, one needs to first divide the given network into sub-domains. For this purpose, any partitioning algorithm, such as the well-known and popular METIS algorithm (Karypis & Kumar, 1998) has been used by transportation researchers (Chabini & Canguapati, 2002). However, the recent development of the SDDA (Johnson et al., 2014; Johnson et al., 2016) has been found to be preferable in terms of its ability to reduce the number of SBN. Regardless of the algorithm, the network's topology and the desired number of sub-domains needs to be specified. After this, decomposition algorithms will automatically divide the topology into NP sub-domains in such a way that:

1. The workload assigned to each processor will be roughly balanced, and more importantly,
2. The degree of connection between each sub-domain (as measured by the number of SBN) is minimized.

Remarks:

1. METIS will only provide the assignment of each node to a particular processor, P_i. Therefore, it is the responsibility of the user to write their own subroutine(s) to identify:
   a. Nodes which are classified as interior node(s) of a particular processor, P_i, and
   b. Nodes which are classified as boundary node(s) of a particular processor, P_i, and
   c. Nodes which are classified as SBN for the entire network.

   However, use of the SDDA will not require this, as it provides all the above mentioned information (Johnson et al., 2014; Johnson et al., 2016).
2. There are several reasons to focus on the theoretical development of the DDSP algorithm:
   a. Numerical results indicate that, as the number of nodes increases, the wall-time to find the SP from all-to-all increases exponentially. When solving all-to-all for a network of 7,388 nodes, the solution time is 3,047 seconds, whereas a network of 1,040 nodes has a solution time of only 15.58 seconds (Allen, 2013 and Lawson et al., 2013). Thus, an increase in network size of approximately seven times results in an increased solution time of nearly 200 times.

b. The DDSP uses computer memory more efficiently since each processor will essentially store a much smaller data set in its own local memory.

c. Through the flexibility of the DDSP algorithm, any classical SP problem algorithm can be employed to analyze/solve each sub-domain.

d. The DDSP algorithm is optimal to achieve a two-level parallel computation:

   i. At the top level, parallel computation can be realized by assigning a group of processors to each sub-domain. Then,

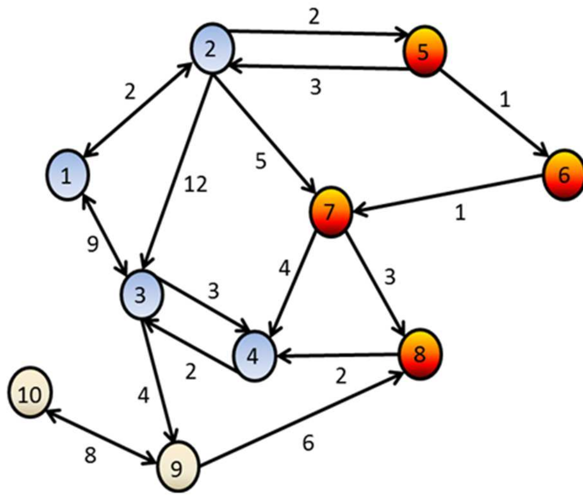
   ii. At the bottom level, any classical parallel SP algorithm can be applied to each sub-domain.

   Thus, the user will have more flexibility to optimally utilize the available number of processors.

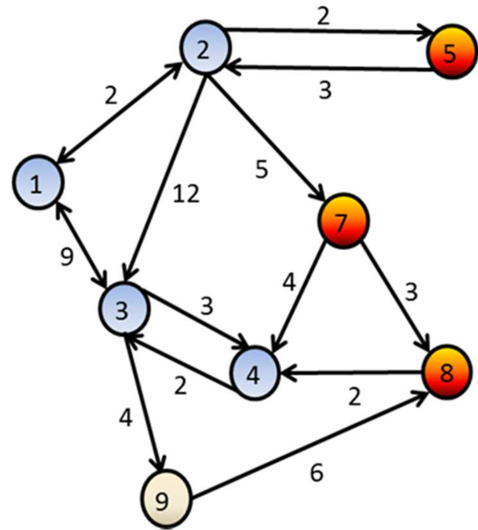
3. In earlier research works (Chabini & Ganugapati, 2002), at every time interval, a slave processor will need to send (receive) information about the boundary links to (from) the other slave processors. Communication is also required between the master processor and slave processors. Thus, distributed memory implementation can become slower, even, than serial implementation. In the proposed DDSP algorithm, only communication between boundary nodes belonging to the sub-domain in question and that same sub-domain's interior nodes are required. Thus, the proposed DDSP is expected to require less communication time.

3.2 Description of a Small-Scale Network to be Analyzed

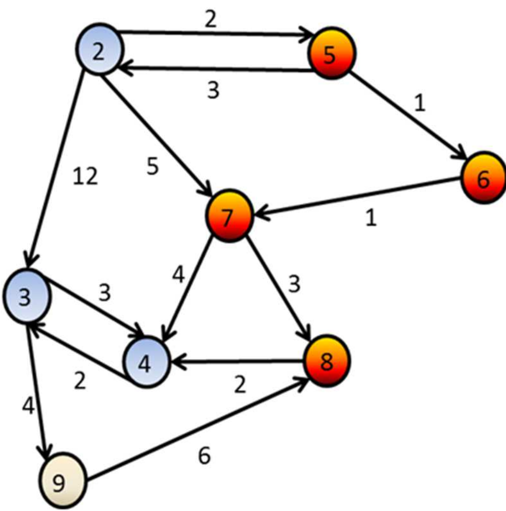
Figure 3-1 represents the original network and its respective sub-domains after partitioning.



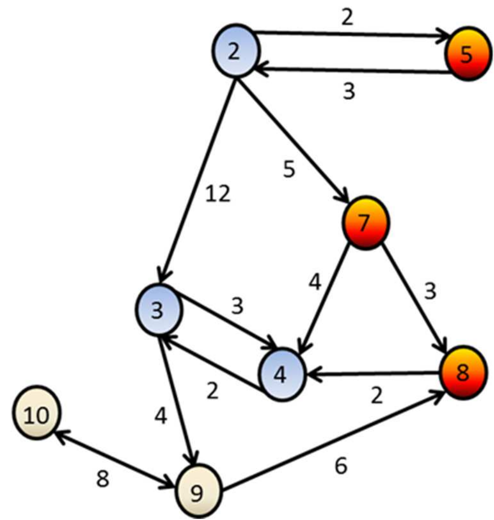
(a) Complete (un-partitioned) Network



(b) Sub-Domain P_0 (Interior node = 1)



(c) Sub-Domain P_1 (Interior node = 6)



(d) Sub-Domain P_2 (Interior node = 10)

Figure 3-1: Network connectivity and associated sub-domain partitioning

It is noteworthy that, while the example considered in this work is a small-scale transportation network, there are no size restrictions imposed on a network to be analyzed by the DDSP algorithm. This figure has been specifically designed, as it has certain properties which cause unique challenges during the DD solution process. The network has been partitioned such that nodes 1-4, 5-8, and 9-10 have been assigned to processors P_0, P_1, and P_2, respectively. Further, any link in the network connected by two nodes which belong to different sub-domains are defined as SBN. Thus, the SBN for this example are 2, 3, 4, 5, 7, 8, and 9. This leaves nodes 1, 6, and 10, which belong to processor P_0, P_1, and P_2, respectively, as each sub-domain's interior nodes. This network was not partitioned using METIS or SDDA (due to its small size); however, it was arbitrarily partitioned by hand to create sub-domains which help explain the algorithm.

By simple observation, the SP from node 2 to node 3 can be found as 2-5-6-7-4-3, for a total ST = 2+1+1+4+2 = 10 units (based on the complete/un-partitioned network). This must mean that to find the correct SP, one must travel from one sub-domain to another through a boundary node. If one were to simply consider sub-domain P_0's topology, one would arrive at the incorrect solution of either 2-1-3 or 2-7-4-3, both of which yield a time of 11 units.

3.3 Solution of Small-Scale Network Utilizing the DDSP Algorithm

The DDSP algorithm consists of just a few simple steps. First, the network must be partitioned, as is shown in Figure 3-1. Once it is partitioned, one can solve the SP problem for each sub-domain independently from all source nodes to all destination nodes in the sub-domain. Step 3 requires one to compute the shortest path for each SBN

to all other nodes in the network. Finally, in Step 4, the algorithm checks the values computed by seeing if the correct shortest path actually occurs over multiple sub-domains. To do so, it simply checks the cost to travel from a source node to an SBN (and this cost is now known), and then adds this cost to the cost to travel from the same SBN to the destination node. If the computed cost is determined to be cheaper than what was previously recorded, both the shortest path cost and predecessor values must be updated. A detailed explanation of each step for the figure provided follows:

Step 1: Partition the network

This step is assumed to have been completed prior to starting the DDSP algorithm. For a recommended partitioning algorithm, readers are referred to Chapter 2 and to Johnson et al. (2014 and 2016).

Step 2: Solve the SP problem for each sub-domain from all-to-all

As mentioned, this network has been partitioned into three sub-domains, resulting in nodes 1-4, 5-8, and 9-10 belonging to sub-domains P_0, P_1, and P_2 respectively. Each of these sub-domains can now be treated as its own independent network, with the topology given by Table 3-1.

Table 3-1: Connectivity Information for Sub-Domains P_0, P_1, and P_2

Sub-Domain P_0			**Sub-Domain P_1**			**Sub-Domain P_2**		
Source	**Dest**	**Cost**	**Source**	**Dest**	**Cost**	**Source**	**Dest**	**Cost**
1	2	2	5	6	1	9	10	8
1	3	9	6	7	1	10	9	8
2	1	2	7	8	3			
2	3	12						
3	1	9						
3	4	3						
4	3	2						

It is now easily seen that each sub-domain is in fact independent of the others, as no sub-domain contains any nodes found in any other sub-domain. Using any classical (serial or parallel) SP algorithm, the first step would be to pre-allocate two matrices: one matrix will eventually contain the shortest path cost information, while the other will contain the predecessor information for each node. References to these two matrices will be made as [D] and [PRED], respectively. Generally, both matrices would be initiated as square matrices with dimensions M by N, which are both equal to the number of nodes in the network. Matrix [D] would be fully populated with the value of infinity for each distance, while [PRED] would be fully populated as a zero matrix. With this known, the following tables are provided, with the values of infinity and zero intentionally left out to make the solution steps appear clearer to the reader, thus leaving empty cells which simply indicate a value for that cell has yet to be computed. Continuing by computing the

all-to-all solution for each independent sub-domain (as shown in Table 1), the values shown in Table 3-2 and Table 3-3 for [D] and [PRED], respectively, are obtained. Note that, in Table 2, the shortest time (ST) to travel from node 6 to node 5 is INF (infinity), because no such path exists. Similar logic is applied to other source-to-destination pairs where no path exists (7-5, 7-6, 8-5, 8-6, and 8-7).

Table 3-2: Shortest Path Solution for Each Sub-Domain - [D]

		Source Nodes (From)									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Destination Nodes (To)	**1**	0	2	9	11						
	2	2	0	11	13						
	3	9	11	0	2						
	4	12	9	3	0						
	5					0	INF	INF	INF		
	6					1	0	INF	INF		
	7					2	1	0	INF		
	8					5	4	3	0		
	9									0	8
	10									8	0

Table 3-3: Shortest Path Solution for Each Sub-Domain - [PRED]

		Source Nodes (From)									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Destination Nodes (To)	**1**	0	2	3	3						
	2	1	0	1	1						
	3	1	1	0	4						
	4	3	3	3	0						
	5					0	0	0	0		
	6					5	0	0	0		
	7					6	6	0	0		
	8					7	7	7	0		
	9									0	10
	10									9	0

Step 3: Solve the SP problem for each SBN to all other nodes

From Figure 3-1, the nodes defined as SBN are easily identified. For example, Sub-Domain P_0 is made up of nodes 1 through 4. Figure 3-1 shows these four nodes along with their connections to the other sub-domains (via nodes 5, 7, 8, and 9). By definition, any node which connects two sub-domains is an SBN. All other nodes are defined as interior nodes belonging to each independent sub-domain. For this example, as previously stated, the SBNs are found to be 2, 3, 4, 5, 7, 8, and 9. Treating each SBN as a source node, one can complete the SP problem to all other nodes in the network.

At this point, it is worth noting the benefit of minimizing the number of SBN during the partitioning process. The fewer SBN there are in a given network, the faster this step may be computed, as it requires fewer nodes to be cycled through the SP algorithm. Completing the one-to-all solution for each SBN allows the previously computed Table 3-2 and Table 3-3 to be updated. The updated results are provided in Table 3-4 and Table 3-5. Any value which changed during the update process is shown in boldface text.

Table 3-4: Shortest Path Solution for each Sub-Domain with SBNs added - [D]

		Source Nodes (From)									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Destination Nodes (To)	**1**	0	2	9	11	5		15	13	19	
	2	2	0	11	13	3		17	15	21	
	3	9	**10<11**	0	2	8		6	4	10	
	4	12	**8<14**	3	0	6		4	2	8	
	5		2	13	15	0	INF	**19<INF**	**17<INF**	23	
	6		3	14	16	1	0	**20<INF**	**18<INF**	24	
	7		4	15	17	2	1	0	**19<INF**	25	
	8		7	10	12	5	4	3	0	6	
	9		14	4	6	12		10	8	0	8
	10		22	12	14	20		18	16	8	0

Table 3-5: Shortest Path Solution for each Sub-Domain with SBNs added - [PRED]

		Source Nodes (From)									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Destination Nodes (To)	**1**	0	2	3	3	2		3	3	3	
	2	1	0	1	1	5		1	1	1	
	3	1	4	0	4	4		4	4	4	
	4	3	7	3	0	7		7	8	8	
	5		2	2	2	0	0	2	2	2	
	6		5	5	5	5	0	5	5	5	
	7		6	6	6	6	6	0	6	6	
	8		7	9	9	7	7	7	0	9	
	9		3	3	3	3		3	3	0	10
	10		9	9	9	9		9	9	9	0

In actual computer implementation, it is clearly more efficient to perform Step 3 before Step 2, so that redundant work can be avoided. For example, the cost from node 2 to nodes 1, 2, 3, and 4 was computed in Step 2. Step 3 subsequently computed this information again by virtue of node 2 being an SBN. Because Step 2 is performed on the partitioned network and Step 3 is then performed on the un-partitioned network, the results of Step 3 are guaranteed to be correct. However, the information has been presented in this order to provide the reader a clear understanding of each step. Step 3 should be performed before Step 2, and in Step 2, any SBN in the sub-domain may be

ignored to ensure that unnecessary computations are not being carried out. Note that several values may have been updated in Table 3-4 and Table 3-5 from the original values presented in Table 3-2 and Table 3-3. With this, Table 3-4 and Table 3-5 represent all of the information required to compute the correct solutions for both matrices [D] and [PRED], which will be assembled in Step 4.

Step 4: Assemble the Matrices [D] and [PRED] for the Original Problem

Utilizing Table 3-4 and Table 3-5, the correct solution for both matrices [D] and [PRED] can be assembled. There are only two possible paths when considering the solution to the shortest path problem based on decomposed networks. First, the correct path lies entirely within a sub-domain. In this instance, the path never travels through a SBN. The second possibility exists when the path crosses a SBN, and the correct path includes nodes from multiple sub-domains. Because the all-to-all solution for each sub-domain has already been computed, all possible answers for the first scenario have also been computed. Now, the information from each SBN is used to check the values previously recorded and to finish filling in any blanks in the tables.

To verify that the values obtained are correct, simply travel from a single source node to an SBN, then from the same SBN to a single destination node, and record the aggregate distance. For example, check the solution for the shortest path from 1 to 4. Referring to Table 3-4, this path has a distance of 12 units. Table 3-5 shows the predecessor node for node 4 as being node 3. By Figure 3-1, node 1 belongs in sub-domain P_0, which has boundary nodes of 2, 3, and 4. So, if the correct path crosses into

another sub-domain, its path is guaranteed to travel through one of these nodes. This is checked as follows:

Path 1: 1 – 2 – 4; looking at Table 3-4, the path from 1 to 2 costs 2 units. The path from 2 to 4 costs 8 units, resulting in a total of 2 units + 8 units = 10 units. Because 10 units is less than the previously computed 12 units, update the table to reflect a cost of 10 units. Because matrix [D] is updated, the corresponding location in the matrix [PRED] must also be updated. The predecessor node of 2-4 is 7. Therefore, update the predecessor node of 1-4 to also be 7.

Path 2: 1 – 3 – 4; looking at Table 3-4, the path from 1 to 3 costs 9 units. The path from 3 to 4 costs 3 units, resulting in a total of 9 units + 3 units = 12 units. Because 12 units is greater than the cost of 10 units computed in Path 1, do not update either matrix [D] or [PRED].

Path 3: 1 – 4 – 4; looking at Table 3-4, the path from 1 to 4 costs 12 units. The path from 4 to 4 costs 0 units, resulting in a total of 12 units + 0 units = 12 units. Because 12 units is greater than the cost of 10 units computed in Path 1, do not update either matrix [D] or [PRED].

Using this methodology and checking every node through each of its respective sub-domains' boundary nodes, the two matrices can be fully updated. The final results are shown in Table 3-6 and in Table 3-7. It should be noted that, when compared to

classical solutions to the SP problem, these updated tables do, in fact, result in the correct solution when compared to the original un-partitioned network. To perform this check, one can simply compare the fully assembled matrices from the partitioned solution to the original un-partitioned network.

Table 3-6: Fully Assembled Shortest Path Solution for Original Network - [D]

		Source Nodes (From)									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Destination Nodes (To)	**1**	0	2	9	11	5	16	15	13	19	27
	2	2	0	11	13	3	18	17	15	21	29
	3	9	10	0	2	8	7	6	4	10	18
	4	10	8	3	0	6	5	4	2	8	16
	5	4	2	13	15	0	20	19	17	23	31
	6	5	3	14	16	1	0	20	18	24	32
	7	6	4	15	17	2	1	0	19	25	33
	8	9	7	10	12	5	4	3	0	6	14
	9	13	14	4	6	12	11	10	8	0	8
	10	21	22	12	14	20	19	18	16	8	0

Table 3-7: Fully Assembled Shortest Path Solution for Original Problem - [PRED]

		Source Nodes (From)									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Destination Nodes (To)	**1**	0	2	3	3	2	3	3	3	3	3
	2	1	0	1	1	5	1	1	1	1	1
	3	1	4	0	4	4	4	4	4	4	4
	4	7	7	3	0	7	7	7	8	8	8
	5	2	2	2	2	0	2	2	2	2	2
	6	5	5	5	5	5	0	5	5	5	5
	7	6	6	6	6	6	6	0	6	6	6
	8	7	7	9	9	7	7	7	0	9	9
	9	3	3	3	3	3	3	3	3	0	10
	10	9	9	9	9	9	9	9	9	9	0

3.4 Application of the DDSP Algorithm to Large-Scale Networks

To test the correctness and overall performance of the DDSP algorithm described in the previous section, this algorithm was employed to solve the SP problem on two small-scale test networks and four real-world transportation networks. (The actual road network connectivity information was retrieved from http://www.bgu.ac.il/~bargera/tntp). The size of the four road networks varied from 24 nodes to 1,040 nodes in an effort to determine how size effects the performance of the algorithm. To provide a metric as to how efficiently the DDSP algorithm performed, some of the same networks used in Chapter 2

were referenced to solve the shortest path problem using the classical Dijkstra method. The networks tested, and the results of the all-to-all classical SP solution, are given by Table 3-8.

Table 3-8: Networks Tested with Classical Serial Dijkstra Solution Time

Network Name	Number of Nodes	Number of Links	Dijkstra All-to-All Calculation Time (s)
10 Node	10	19	0.004
15 Node	15	48	0.006
Sioux Falls	24	76	0.029
Anaheim	416	914	472.4
Barcelona	930	2,522	10,615
Winnipeg	1,040	2,836	16,435

All six networks were partitioned into two, three, and four sub-domains. The partitions were obtained using the SDDA algorithm. The DDSP algorithm was then employed to solve the SP problem. To provide a meaningful comparison, the algorithm utilized the same Dijkstra algorithm that had been used in the generation of the data shown in Table 3-8 (in a serial computational environment). For all networks tested, the DDSP algorithm obtained the same shortest distance, as compared to the classical Dijkstra algorithm for all origin-to-destination pairs. To show a complete solution, the partitioning times are provided in Table 3-9. To this table, the computational time of the DDSP was added to arrive at the values in Table 3-10. Several values are highlighted in this table, which show the most efficient partition size for the given network. For

example, consider the Anaheim network. It can be seen the most efficient partition occurs when the network is partitioned into four sub-domains.

Table 3-9: SDDA Partitioning Time for Tested Networks

Network Name	SDDA Partition Time (s)				
	NP = 2	NP = 3	NP = 4	NP = 5	NP = 6
10 Node	0.001	0.001	0.001	0.002	0.002
15 Node	0.001	0.001	0.002	0.002	0.002
Sioux Falls	0.002	0.002	0.002	0.003	0.003
Anaheim	0.022	0.023	0.021	0.034	0.041
Barcelona	0.067	0.068	0.073	0.099	0.099
Winnipeg	0.080	0.082	0.090	0.112	0.121

Table 3-10: Total Serial Solution Time – SDDA Plus DDSP

DDSP Shortest Path and Total (Partition Included) Serial Calculation Time (s)										
Network Name	**NP = 2**		**NP = 3**		**NP = 4**		**NP = 5**		**NP = 6**	
	DDSP	**Total**	**DDSP**	**Total**	**DDSP**	**Total**	**DDSP**	**Total**	**DDSP**	**Total**
10 Node	0.002	0.003	0.002	**0.003**	0.002	0.003	0.002	0.004	0.002	0.004
15 Node	0.004	0.005	0.003	**0.004**	0.004	0.006	0.006	0.008	0.006	0.008
Sioux Falls	0.018	**0.020**	0.021	0.023	0.022	0.024	0.021	0.024	0.022	0.025
Anaheim	216.7	216.7	129.0	129.0	109.6	**109.6**	132.3	132.3	146.0	146.0
Barcelona	4,801	4,801	3,657	3,657	3,347	**3,347**	3,403	3,403	3,809	3,809
Winnipeg	7,581	7,581	5,533	5,533	4,536	4,536	3,807	**3,807**	4,001	4,001

As can be seen, the DDSP algorithm efficiently completed the SP problem for the large-scale transportation networks. For example, for the Winnipeg network, when using four partitions, the total time to find the all-to-all shortest paths was only 4,536 seconds, as compared to 16,435 seconds when no decomposition was used (cf. Table 3-8). With the partitioning time considered in the total computational time, a speedup factor of several times is obtained. As mentioned, one can further enhance the performance of the DDSP algorithm if parallel processing is exploited. In fact, the DDSP algorithm offers the opportunity for a two level parallel computation. The top level of parallel computation is realized by assigning a group of processors to each sub-domain. The bottom level of parallel computation is then achieved by conventional parallel strategies, such as assigning each processor to handle a group of source nodes.

Examining the larger networks, as the number of sub-domains increase, the computational time required by the DDSP algorithm is decreased. There will, however, exist a point of diminishing return when the number of sub-domains is too large. This is due to an increased solution time for a fixed problem size, as more partitions generally result in an increased number of SBNs and an increased amount of communication time.

3.5 Conclusions

In this work, a new and general DDSP algorithm is presented. The algorithm uses DD to solve the SP problem in an efficient manner. As the computational effort for the SP problem increases exponentially with the number of nodes in the network, the proposed DDSP algorithm is proven to be effective in reducing the computational time, even in the

serial computer environment. It is expected that additional computational efficiencies may be observed in parallel environments, since each sub-domain can be processed simultaneously. To the best of the author's knowledge, the proposed algorithm is not only novel, but also represents the first attempt to clearly explain the detailed steps for coupling domain decomposition concepts with existing SP algorithms, while including numerical comparisons between the DDSP algorithm and an existing SP algorithm.

CHAPTER 4

USING THE SDDA TO PRODUCE FILL REDUCING ORDERINGS

4.1 Introduction

When using direct methods (such as Gaussian elimination, Cholesky factorization, LU decomposition, LDL^T decomposition, etc.) to solve systems of linear equations, fill-in terms occur where the coefficient matrix factor changes from what is initially zero to a non-zero value. Reducing the number of fill-in terms reduces memory requirements, as there are fewer terms to store. It also reduces the number of operations required to solve the system of equations, which results in a reduced computational time. The method of reordering used in the SDDA algorithm was thought to exhibit properties beneficial to reducing fill. To compare the performance of the SDDA, it was once again compared with the popular METIS algorithm. The authors of METIS (Karypis & Kumar, 1998) claim, "The fill-reducing orderings produced by METIS are significantly better than those produced by other widely used algorithms including multiple minimum degree. For many classes of problems arising in scientific computations and linear programming, METIS is able to reduce the storage and computational requirements of sparse matrix factorization, by up to an order of magnitude." As such, it was determined METIS would provide an adequate benchmark to compare results to.

4.2 Using the SDDA to Produce Fill Reducing Orderings

The same seven networks used in Chapter 2 are used in this chapter to compute the number of fill-in terms which occur after LU factorization of the coefficient matrix.

Because these networks are transportation networks, the value on the diagonal of the coefficient matrix is zero, as the cost to travel to a given node from that same node is zero. To represent a more realistic application of a set of simultaneous linear equations which may be seen in other fields (structural engineering, aerospace engineering, computational fluid dynamics, and engineering mechanics, to name a few), an artificial diagonal was added to each coefficient matrix. In these types of applications, the diagonal term is nearly always positive. As such, the artificial diagonal was set equal to the absolute value of the sum of the off-diagonal term for each row, and was multiplied by 10.

Each system, with its modified diagonal, was sub-structured using METIS and the SDDA. The total number of non-zero terms (original non-zeros plus the amount of fill-in terms) were recorded after LU factorization for partition sizes of 2, 3, and 4 sub-domains, respectively. The results can be found in Table 4-1.

Table 4-1: Fill-In Term Comparison - METIS and the SDDA

Total Number of Non-Zero Terms After LU Factorization						
Network	METIS NP = 2	METIS NP = 3	METIS NP = 4	SDDA NP = 2	SDDA NP = 3	SDDA NP = 4
Sioux Falls	240	202	236	**208**	**202**	**210**
Anaheim	11,456	11,960	10,342	**9,339**	**7,667**	**7,190**
Barcelona	63,447	55,512	56,742	**54,578**	**49,717**	**49,973**
Winnipeg	46,744	34,700	40,281	65,253	53,094	44,074
Austin	1,603,263	1,310,687	1,378,730	**1,144,094**	**834,397**	**741,683**
Chicago	5,388,605	3,741,665	3,821,323	**2,599,459**	**2,145,965**	**2,024,711**
Philadelphia	4,442,922	3,002,843	3,390,924	**2,183,169**	**1,674,688**	**1,470,988**

As can be seen by the highlighted values in the table, the SDDA outperformed METIS in every test, except for the Winnipeg network (or in 18 of the 21 tests = 86%). It is noteworthy to point out that the SDDA also performed poorly in partitioning the

Winnipeg network, in terms of minimizing the number of SBN, when compared to METIS (refer to Chapter 2). Since the SDDA performed so well, could a multi-level reordering be possible, to improve other existing reordering algorithms currently used?

4.3 Multi-Level Reordering Using the SDDA as a Preordering Algorithm

There are several popular reordering algorithms available in a number of commercial software programs as built-in code. For example, MATLAB offers the following reordering algorithms:

AMD	Approximate minimum degree permutation
COLAMD	Column approximate minimum degree permutation
COLPERM	Sparse column permutation based on nonzero count
DMPERM	Dulmage-Mendelsohn decomposition
RANDPERM	Random permutation
SYMAMD	Symmetric approximate minimum degree permutation
SYMRCM	Sparse reverse Cuthill-McKee ordering

Many of these algorithms perform very efficiently, and outright outperformed METIS and the SDDA in the networks tested, however, could a preordering phase be beneficial prior to reordering using one of these schemes? To answer this question, the same networks were tested, again with 2, 3, and 4 partitions, with the SDDA as a preordering function to several of these built in functions. The functions tested were AMD, COLAMD, SYMRCM, and COLPERM. First, the built-in functions were tested alone, and yielded the following results:

Table 4-2: Total Number of Non-Zeros after LU Factorization

Network	AMD	COLAMD	SYMRCM	COLPERM
Sioux Falls	170	194	202	180
Anaheim	3,309	3,727	6,433	10,813
Barcelona	12,436	16,907	34,464	39,619
Winnipeg	10,093	13,391	40,726	44,973
Austin	82,818	122,659	756,600	1,353,678
Chicago	229,727	396,563	1,676,781	6,357,352
Philadelphia	128,415	212,808	1,302,577	1,234,380

With a baseline now established, the networks were preordered using the SDDA ordering and then were reordered, to see if a difference was realized. The following tables show these results for NP = 2, 3, and 4 respectively. After analyzing, it can be seen that preordering produces better results 50% of the time (54%, 46%, and 50% for NP = 2, 3, and 4 respectively).

Table 4-3: Total Number of Non-Zero Terms after LU Factorization - NP = 2

Network	AMD	COLAMD	SYMRCM	COLPERM
Sioux Falls	**170**	**194**	**200**	**180**
Anaheim	3,360	3,807	6,596	**8,925**
Barcelona	13,091	17,940	38,168	**36,341**
Winnipeg	10,153	**13,218**	40,806	**42,977**
Austin	83,451	**119,910**	761,356	1,429,554
Chicago	**225,886**	**393,025**	**1,670,110**	**3,650,755**
Philadelphia	**128,380**	215,224	1,385,784	**898,912**

Table 4-4: Total Number of Non-Zero Terms after LU Factorization - NP = 3

Network	AMD	COLAMD	SYMRCM	COLPERM
Sioux Falls	**170**	**194**	**202**	**180**
Anaheim	3,382	3,820	6,535	**8,829**
Barcelona	12,700	17,216	38,147	40,179
Winnipeg	10,408	13,565	**40,507**	**43,654**
Austin	**82,394**	122,710	763,509	1,674,602
Chicago	232,422	**392,233**	**1,671,790**	**3,828,680**
Philadelphia	128,450	**207,257**	1,376,980	**812,430**

Table 4-5: Total Number of Non-Zero Terms after LU Factorization - NP = 4

Network	AMD	COLAMD	SYMRCM	COLPERM
Sioux Falls	**170**	**194**	216	**180**
Anaheim	3,339	3,938	6,519	**8,958**
Barcelona	12,653	17,211	38,279	**38,353**
Winnipeg	10,377	**13,379**	**40,633**	**42,326**
Austin	**81,603**	123,700	761,455	1,599,720
Chicago	230,715	**386,719**	**1,672,509**	**3,973,383**
Philadelphia	129,106	**209,462**	1,386,747	**812,552**

4.4 Conclusions

Examining the results presented in Table 4-1 through Table 4-5, several conclusions can be drawn. First, the SDDA outperforms METIS as a reordering algorithm when minimizing the number of fill-in terms is the desired goal. Second, if the SDDA is used as a preordering algorithm, it only produces better results 50% of the time. Based on this, it cannot be said that preordering with the SDDA will always offer better orderings, however, it can be said that:

1. Preordering with SDDA may be beneficial
   a. SDDA provided same or better results in 50% of tested cases
   b. 81% of tests were better than Sparse Column Permutation
   c. 52% of tests were better than Column Approximate Minimum Degree Permutation
   d. 33% of tests were better than Sparse Reverse Cuthill-McKee Ordering
   e. 33% of tests were better than Approximate Minimum Degree
2. As network size increased, results appear better

So, if the COLPERM or COLAMD algorithms are to be used, SDDA preordering improves the results more often than not. However, preordering with SDDA prior to using SYMRCM or AMD often produces less favorable results.

CHAPTER 5

CONCLUSIONS AND RECOMMENDATIONS

This manuscript has presented a new and novel heuristic algorithm with the purpose of partitioning a set of data into a predefined number of sub-domains. From here, two applications of the algorithm are presented with results and conclusions are drawn. The SDDA was compared to what is believed by many to be the standard in general graph partitioning, METIS. When comparing general road networks, the SDDA showed a reduction in system boundary nodes of up to 400% (refer to the Austin network with its three partitions). Nguyen (2006) made it very clear in an example which demonstrates that when the number of system boundary nodes increases, the computational time to solve the domain decomposition problem can increase dramatically (from 0.45% to 97% of the overall computation time). Therefore, in domain decomposition applications, it is of paramount importance to reduce the communication time required by each sub-domain (by minimizing the number of system boundary nodes), since communication time increases exponentially as the degree of connection is increased. While this time may be offset by introducing multiple processors and by solving the problem via parallel computation techniques, the exponential increase in communication time outweighs the benefit realized by multiple processors (Nguyen, 2006).

The first major application of the SDDA was the so-called DDSP. The DDSP algorithm utilizes the SDDA to provide efficient partitions when computing the solution to the shortest path problem. As previously mentioned, as the network size increases, the computational effort for the SP problem increases exponentially. Providing an algorithm

which allows the original network to be solved by solving smaller pieces at a time offers significant computational efficiencies, as can be seen by simply comparing Table 3-8 and Table 3-10. This comparison is made in the serial computing environment. A potential major improvement to the DDSP algorithm would be to solve the individual sub-domains in parallel, which would significantly increase computational efficiency.

Finally, the SDDA algorithm has been used to produce fill-reducing orderings. In the reordering process of the algorithm (reference Steps 6 and 7), the nodes are situated in such a way that the non-zero terms are clustered tightly around the diagonal. By doing this, opportunity for fill-in terms is reduced because there are significantly fewer zeros between the diagonal and the non-zero term furthest from the diagonal in any given row or column. Again, comparing the SDDA to METIS, one can see that the SDDA outperforms METIS in 86% of the tests conducted. Reducing the number of fill-in terms is beneficial to many engineering and computational science fields: computational fluid dynamics, aerospace engineering, structural engineering, and general finite element analysis models used to solve a set of simultaneous linear equations are a few examples. The utility is beneficial, since when fill occurs during the factorization process, the problem size increases. Providing an efficient preordering or reordering of the network reduces the amount of fill, which reduces the problem size, which increases computational efficiency.

There are potentially several other applications which can be exploited in future work. For example, utilizing domain decomposition can prove useful for the class of continuum traffic equilibrium problems (e.g. see Wong et al., 1998), network wide incident management planning (Ng et al., 2013), and certain classes of shortest path

problems (Ng & Sathasivan, 2014). Other applications might include transportation problems that are known to be computationally challenging, including dynamic traffic assignment, stochastic and dynamic routing problems, and problems currently addressed using metaheuristic approaches (Chen et al., 2014; Flötteröd & Liu, 2014; Kurauchi & Yoshii, 2014, Szeto, 2014; Tian and Chiu, 2014; Ghanim & Abu-Lebdeh, 2015). In fact, for this last application, it was found that significant speed-ups can be made possible by SDDA. For example, for the Austin road network, speed-ups of 2.7 times were found within one iteration of the policy iteration algorithm, when decomposing the network in four sub-domains. Recall, from Table 2-5, that SDDA gives 305 system boundary nodes, whereas METIS gives 1221 boundary nodes in this case. One last future research effort might be trying to further reduce (e.g. with other data structures) the computational time of SDDA, despite its being very minimal already.

REFERENCES

Allen, Shawn. Parallel Domain Decomposition Polynomial LCA (Label Correction Algorithm) Solution For DUE (Deterministic User Equilibrium) Problems. M.S. thesis. Old Dominion University. Department of Civil and Environmental Engineering, 2013

Bellman, R. (1956). *On a routing problem* (No. RAND-P-1000). RAND CORP SANTA MONICA CA.

Chabini, I., & Ganugapati, S. (2002). Parallel algorithms for dynamic shortest path problems. *International Transactions in Operational Research*, *9*(3), 279-302.

Chabini, I. (1998). Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time. *Transportation Research Record: Journal of the Transportation Research Board*, (1645), 170-175.

Chen, B. Y., Lam, W. H., Sumalee, A., Li, Q., & Tam, M. L. (2014). Reliable shortest path problems in stochastic time-dependent networks. *Journal of Intelligent Transportation Systems*, *18*(2), 177-189.

Chen, J., & Taylor, V. E. (2002). Mesh partitioning for efficient use of distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, *13*(1), 67-79.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*(1), 269-271.

Etemadnia, H., Abdelghany, K., & Hassan, A. (2014). A network partitioning methodology for distributed traffic management applications. *Transportmetrica A: Transport Science*, *10*(6), 518-532.

Farhat, C., & Lesoinne, M. (1993). Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *International Journal for Numerical Methods in Engineering*, *36*(5), 745-764.

Foster, Ian. *Designing and Building Parallel Programs.* 3.9.1 Parallel row-wise Domain Decomposition. http://www-unix.mcs.anl.gov/dbpp/text/book.html. November 2003.

Flötteröd, G., & Liu, R. (2014). Disaggregate path flow estimation in an iterated dynamic traffic assignment microsimulation. *Journal of Intelligent Transportation Systems*, *18*(2), 204-214.

Ghanim, M. S., & Abu-Lebdeh, G. (2015). Real-Time Dynamic Transit Signal Priority Optimization for Coordinated Traffic Networks Using Genetic Algorithms and Artificial Neural Networks. *Journal of Intelligent Transportation Systems*, 19(4), 327-338.

Glover, F., Klingman, D., & Phillips, N. (1985). A new polynomially bounded shortest path algorithm. *Operations Research*, *33*(1), 65-73.

Habbal, M. B., Koutsopoulos, H. N., & Lerman, S. R. (1994). A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures. *Transportation Science*, *28*(4), 292-308.

Johnson III, P. W., Nguyen, D., & Ng, M. (2014). An Efficient Shortest Distance Decomposition Algorithm for Large-Scale Transportation Network Problems. In *Transportation Research Board 93rd Annual Meeting* (No. 14-2921).

Johnson, P., Nguyen, D., & Ng, M. (2016). Large-scale network partitioning for decentralized traffic management and other transportation applications. *Journal of Intelligent Transportation Systems*, 20(5), 461-473.

Karypis, G., & Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, *20*(1), 359-392.

Karypis, G., & Kumar, V. (1998). A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*.

Kernighan, B. W., & Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, *49*(2), 291-307.

Kouatchou, J. (2009, November 2). Comparing Python, NumPy, Matlab, Fortran, etc. Retrieved August 8, 2014, from https://modelingguru.nasa.gov/docs/DOC-1762.

Kurauchi, F., & Yoshii, T. (2014). Special Section on Dynamic traffic assignment: A tool for evaluating and assessing dynamic traffic management schemes. *Journal of Intelligent Transportation Systems*, *18*(2), 175-176.

Lawson, G., Allen, S., Rose, G., Nguyen, D., & Ng, M. (2013). Parallel label correcting algorithms for large-scale static and dynamic transportation networks on laptop personal computers. *TRB 92nd Annual Meeting Compendium of Papers*, (13-2103).

Logi, F., & Ritchie, S. G. (2002). A multi-agent architecture for cooperative inter-jurisdictional traffic congestion management. *Transportation Research Part C: Emerging Technologies*, *10*(5), 507-527.

Ng, M., Khattak, A., & Talley, W. K. (2013). Modeling the time to the next primary and secondary incident: A semi-Markov stochastic process approach. *Transportation Research Part B: Methodological*, *58*, 44-57.

Ng, M., & Sathasivan, K. (2014). Probabilistic Modeling of Erroneous Human Response to In-Vehicle Route Guidance Systems: A First Look. *Journal of Intelligent Transportation Systems*, *18*(2), 131-137.

Nguyen, D. T. (2006). *Finite Element Methods: Parallel-Sparse Statics and Eigen-Solutions*. Springer Science & Business Media.

Pavlis, Y., & Papageorgiou, M. (1999). Simple decentralized feedback strategies for route guidance in traffic networks. *Transportation science*, *33*(3), 264-278.

Przemieniecki, Janusz S. *Theory of matrix structural analysis*. Courier Corporation, 1985.

Simon, H. D. (1991). Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, *2*(2), 135-148.

Szeto, W. Y. (2014). Dynamic Modeling for Intelligent Transportation System Applications. *Journal of Intelligent Transportation Systems*, *18*(4), 323-326.

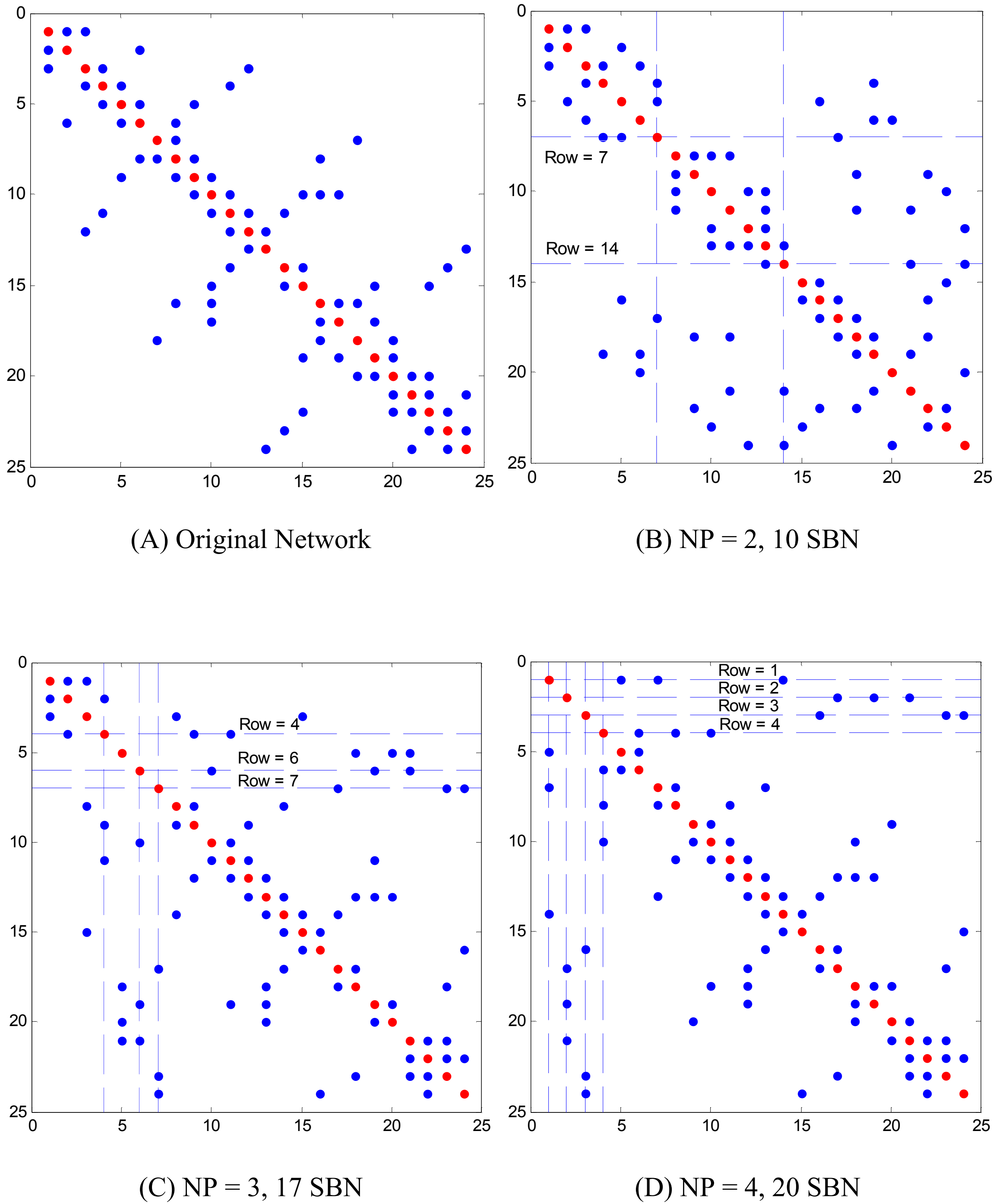
Tian, Y., & Chiu, Y. C. (2014). A variable time-discretization strategies-based, time-dependent shortest path algorithm for dynamic traffic assignment. *Journal of Intelligent Transportation Systems*, *18*(4), 339-351.

Wong, S. C., Lee, C. K., & Tong, C. O. (1998). Finite element solution for the continuum traffic equilibrium problems. *International Journal for Numerical Methods in Engineering*, *43*(7), 1253-1273.

Ziliaskopoulos, A., Kotzinos, D., & Mahmassani, H. S. (1997). Design and implementation of parallel time-dependent least time path algorithms for intelligent transportation systems applications. *Transportation Research Part C: Emerging Technologies*, *5*(2), 95-107.

APPENDICES

APPENDIX A: SDDA PARTITION GRAPHS



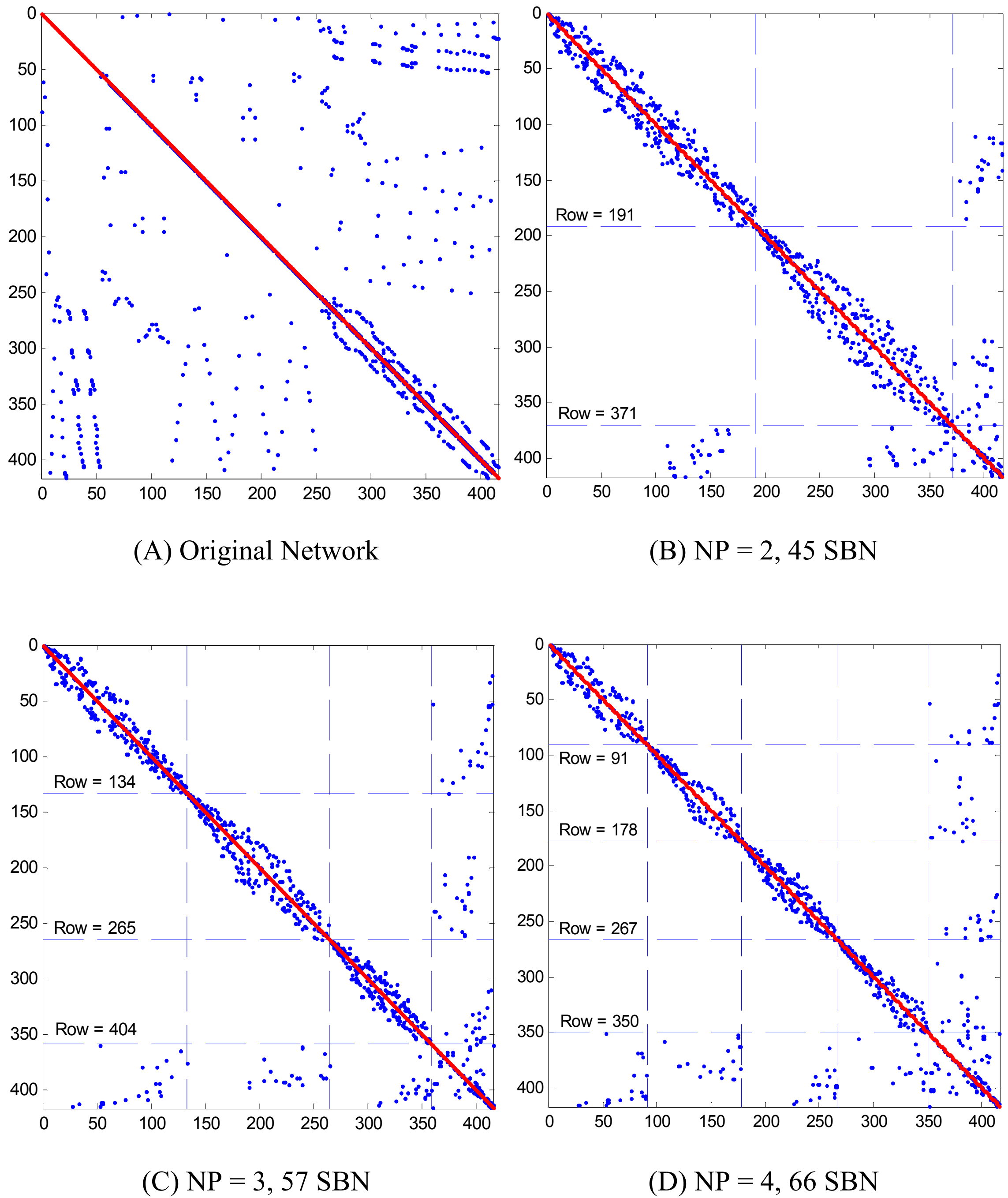
(A) Original Network

(B) NP = 2, 10 SBN

(C) NP = 3, 17 SBN

(D) NP = 4, 20 SBN

Figure A-1: Partitioning of the Sioux Falls Network



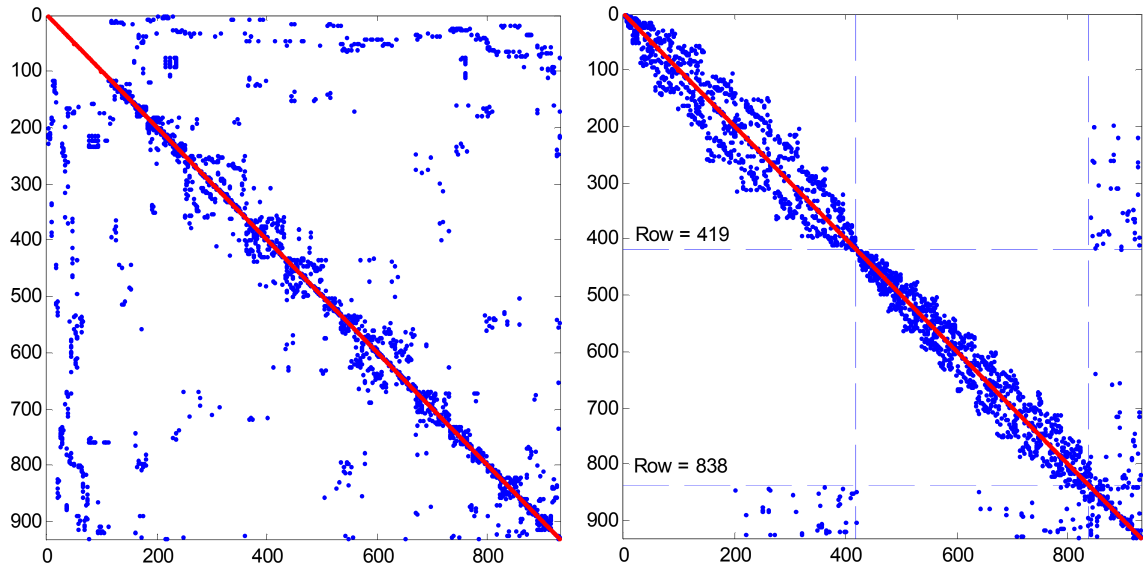
(A) Original Network

(B) NP = 2, 45 SBN

(C) NP = 3, 57 SBN

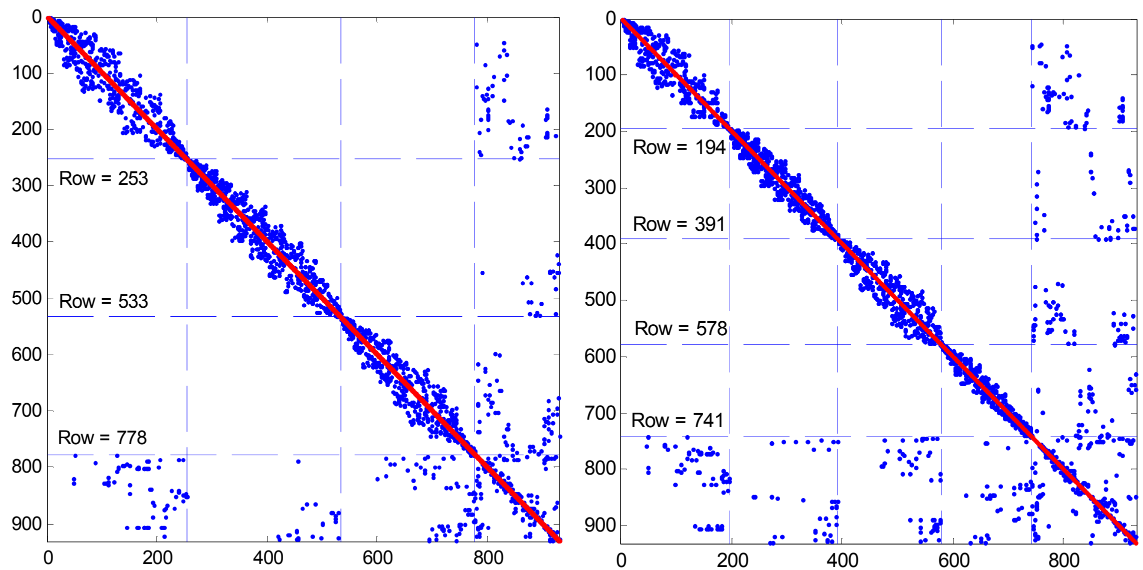
(D) NP = 4, 66 SBN

Figure A2: Partitioning of the Anaheim Network



(A) Original Network

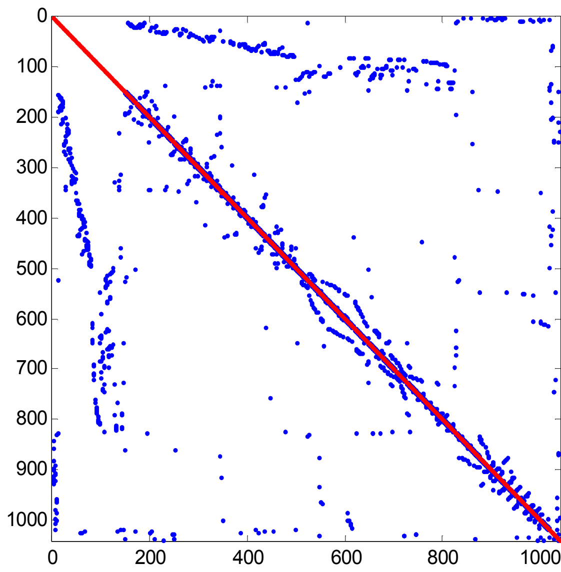
(B) NP = 2, 92 SBN



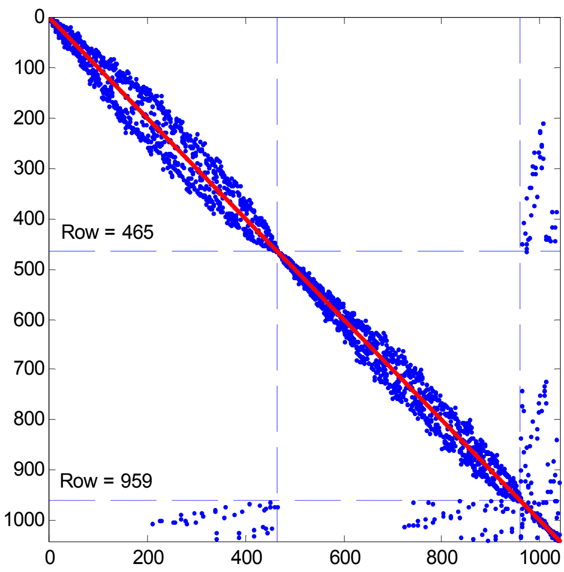
(C) NP = 3, 152 SBN

(D) NP = 4, 189 SBN

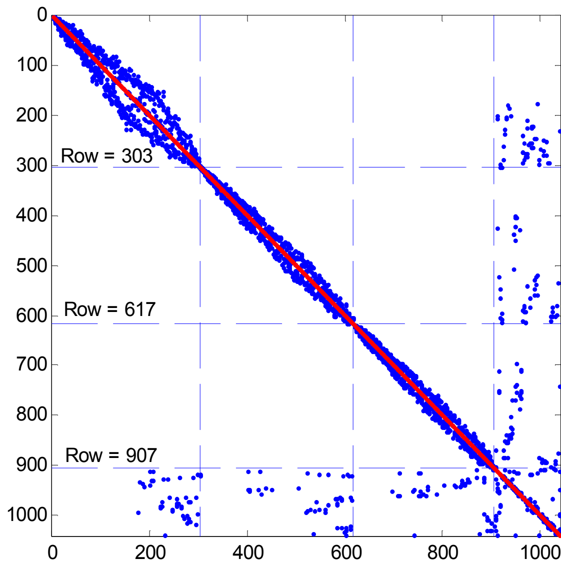
Figure A3: Partitioning of the Barcelona Network



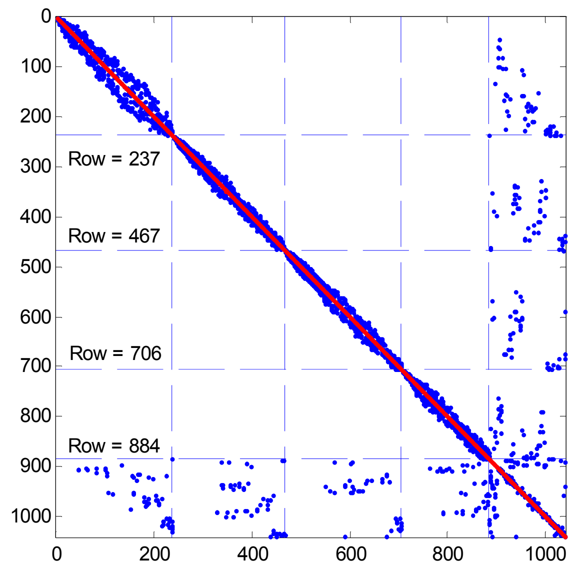
(A) Original Network



(B) NP = 2, 81 SBN

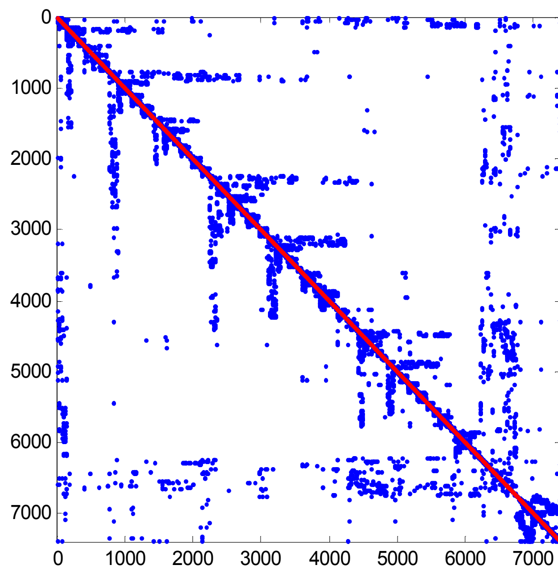


(C) NP = 3, 133 SBN

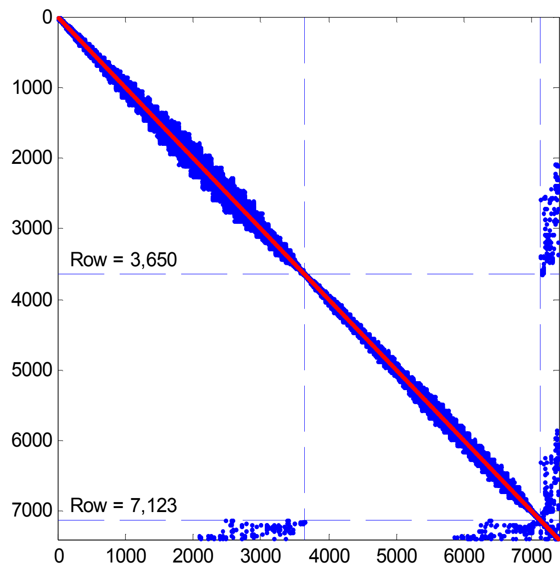


(D) NP = 4, 156 SBN

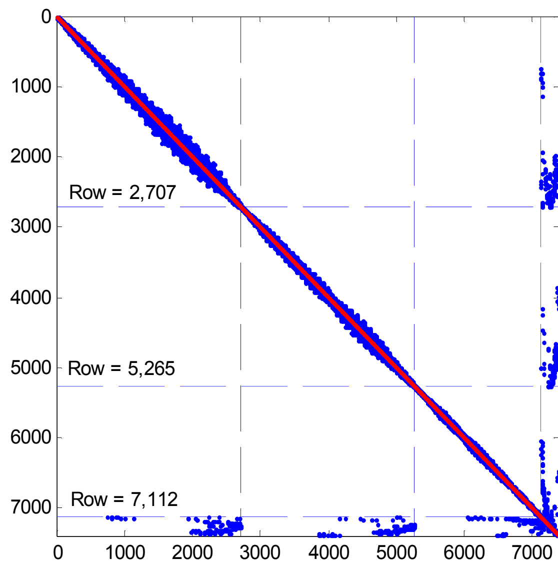
Figure A4: Partitioning of the Winnipeg Network



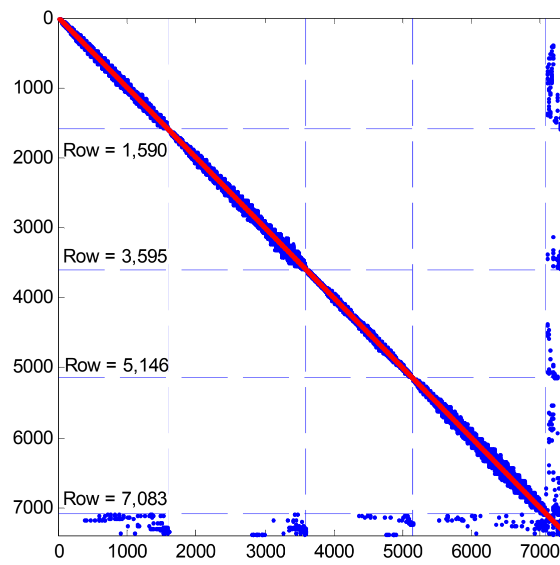
(A) Original Network



(B) NP = 2, 265 SBN

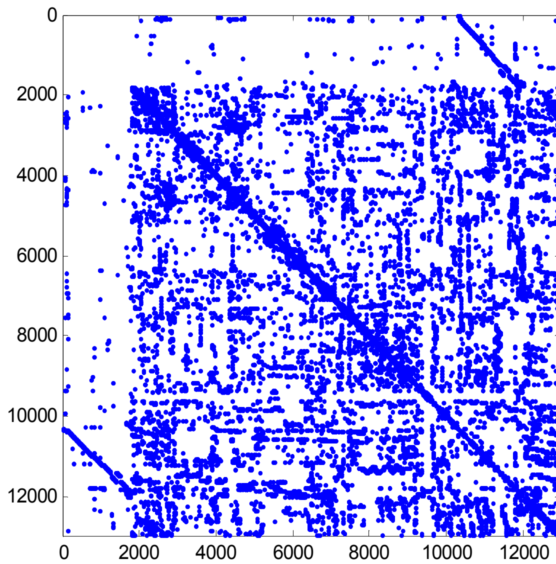


(C) NP = 3, 276 SBN

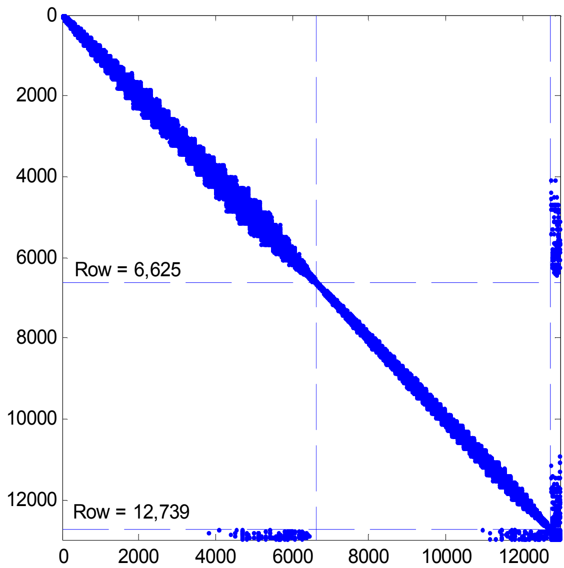


(D) NP = 4, 305 SBN

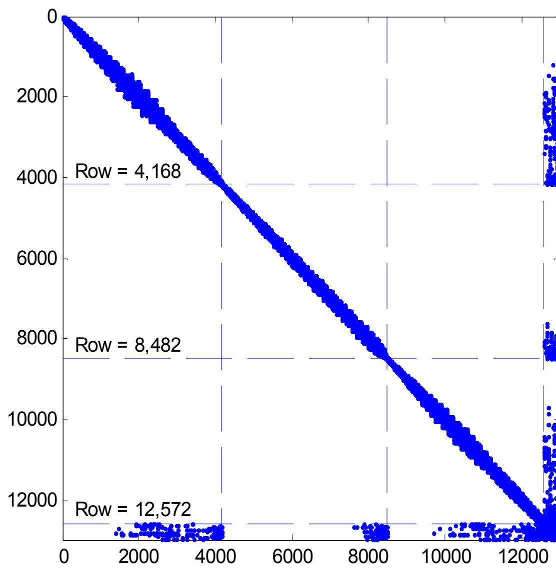
Figure A5: Partitioning of the Austin Network



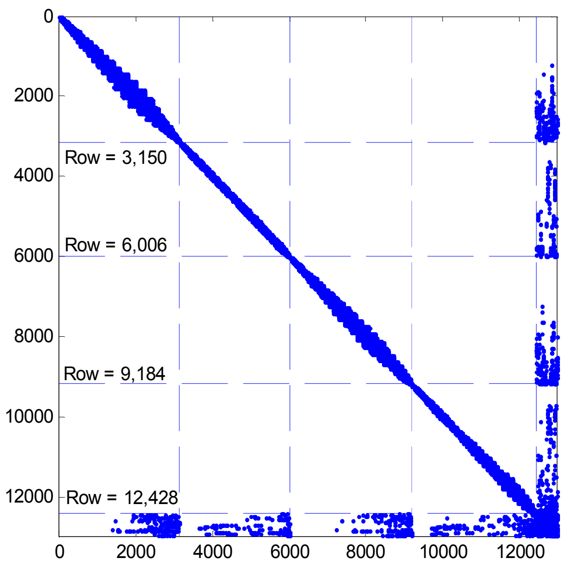
(A) Original Network



(B) NP = 2, 240 SBN

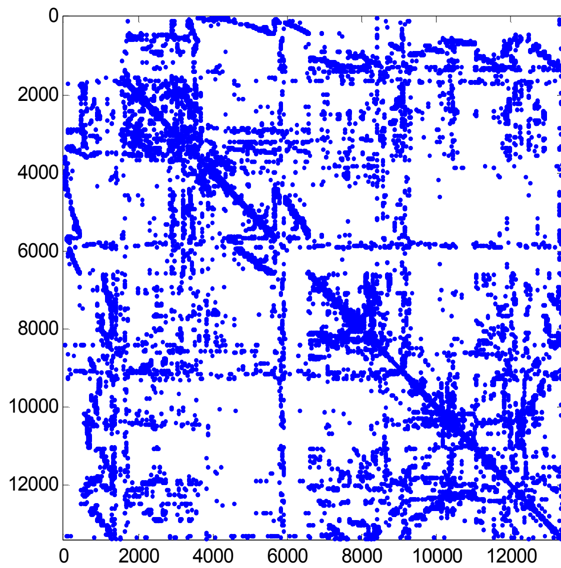


(C) NP = 3, 407 SBN

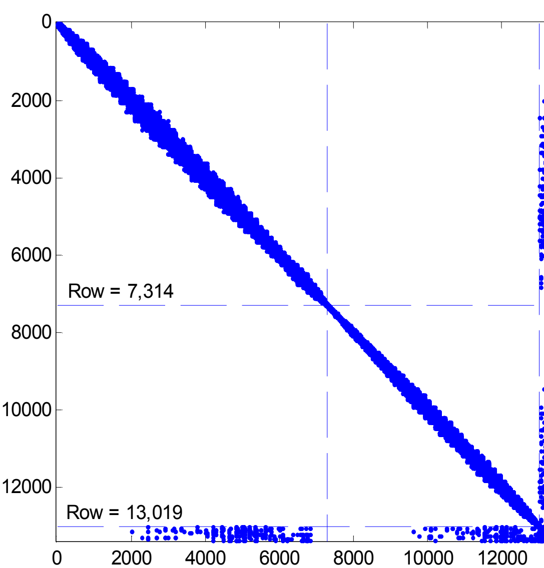


(D) NP = 4, 551 SBN

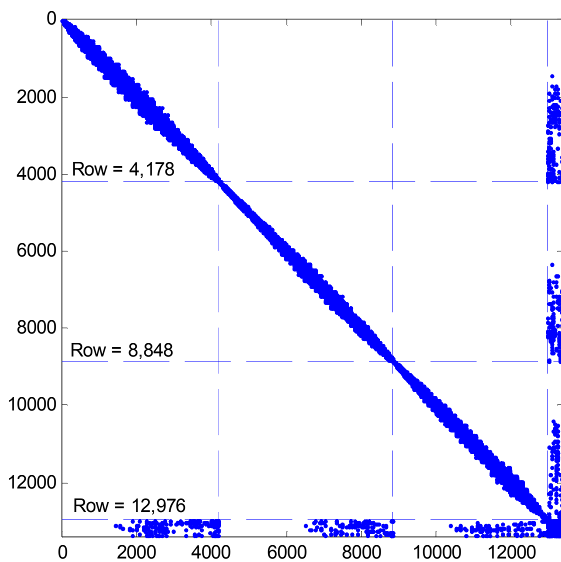
Figure A6: Partitioning of the Chicago Network



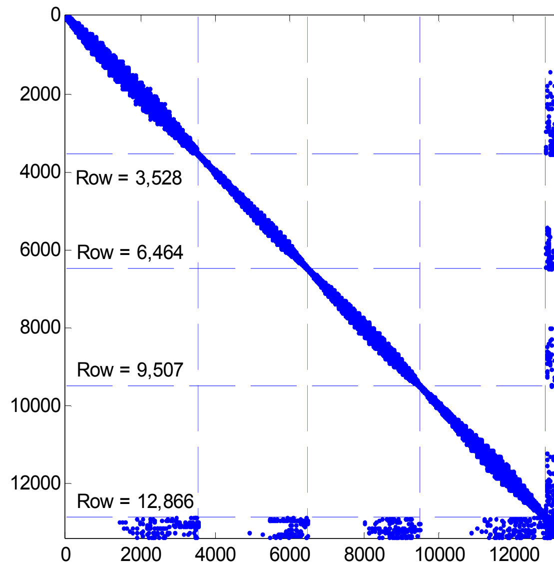
(A) Original Network



(B) NP = 2, 370 SBN



(C) NP = 3, 413 SBN



(D) NP = 4, 523 SBN

Figure A7: Partitioning of the Philadelphia Network

APPENDIX B: MATLAB Source Code for the SDDA

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%File name = SDDA.m
%Written by Paul W. Johnson
%Old Dominion University
%Code Date 2016/10/31

%The SDDA is a seven step algorithm. In these seven steps, the
%algorithm reads and manipulates a user provided set of data and
%efficiently sub-structures the data set into a user defined number
%(NP) of sub-domains It is the goal of the algorithm that the resulting
%number of system boundary nodes is small, and each sub-domain is of
%similar size. Ultimately the algorithm re-numbers the nodes in such a
%manner that the interior nodes of each sub-domain are completely
%independent from every other sub-domain.

%Some command definitions are per MathWorks.com

%Refer to Johnson, P., Nguyen, D., & Ng, M. (2016). "Large-scale
%network partitioning for decentralized traffic management and other
%transportation applications." Journal of Intelligent Transportation
%Systems, 1-13. for further definition of the SDDA algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear   %Removes all variables from the current workspace,
        %releasing them from system memory.

NP = input('Please input number of sub-domains.');  %Defines number of
                                                     %partitions desired

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 1: INPUT PROBLEM TOPOLOGY
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic     %Starts a stopwatch timer to measure performance. The function
        %records the internal time at execution of the tic command.
        %This instance begins recording the time required for STEP 1.

        %Below represents the network connectivity information for the
        %netwokrs tested when developing the SDDA and DDSP algorithms.
        %Simply save any .txt file to the same directory as this .m
        %file and define it below, and the program will read the
        %appropriate (uncommented) file.

Element_Connectivity = load('15_Node.txt');
%Element_Connectivity = load('Sioux_Falls.txt');
%Element_Connectivity = load('Anaheim.txt');
%Element_Connectivity = load('Barcelona.txt');
%Element_Connectivity = load('Winnipeg.txt');
%Element_Connectivity = load('Austin.txt');
%Element_Connectivity = load('Chicago.txt');
%Element_Connectivity = load('Philadelphia.txt');
```

```
A = Element_Connectivity; %Stores network information as a variable
A = sortrows(A); %Sorts rows of network connectivity for convenience
An = A(:,3); %Extracts the cost information and stores as a variable
A = A(:,[1 2]); %Delets the cost information, leaving source/dest info

%Because the SDDA algorithm assumes each link is bi-directional the
%algorithm must ensure each origin/destination (O-D) pair has a reverse
%direction defined solely for partitioning purposes

B=flipdim(A,2); %Creates a reverse link for every pair defined in 'A'
C = [A;B]; %Vertically concatenates the two variables 'A' and 'B'
C = unique(C,'rows'); %Deletes any redudant links created by 'C'
clear B; %Deletes variable 'B' as it is no longer required to be stored

nelements = length(C);  %Computes the number of links in modified data
                        %set
nnodes = length(unique(C)); %Computes the number of nodes in network

Step_1 = toc;   %reads the elapsed time from the stopwatch timer
                %started by the tic function. The function reads the
                %internal time at the execution of the toc command, and
                %displays the elapsed time since the most recent call
                %to the tic function that had no output, in seconds. At
                %this point, the time for STEP 1 is recorded.

%NP = input('Please input number of sub-domains.');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 2: DETERMINE THE RANK, R, OF EACH NODE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic %Resets and starts the stopwatch for STEP 2

R = accumarray(C(:,1),1);   %Calculates the Rank, R, of each node and
                            %stores in variable form

Step_2 = toc; %Stops the stopwatch and records the time for STEP 2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 3: DETERMINE THE FIRST SOURCE NODE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic %Resets and starts the stopwatch for STEP 3
first_node = find(R==min(R),1); %Determines first source node
Step_3 = toc; %Stops the stopwatch and records the time for STEP 3

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 4: DETERMINE OTHER SOURCE NODES USING THE MODIFIED PLCA METHOD
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic %Resets and starts the stopwatch for STEP 4

IA = zeros(nnodes + 1,1); %Computes IA vector for sparse storage scheme
IA(1)=1;
```

```
for i = 2:nnodes+1
    IA(i) = IA(i-1) + R(i-1);
end

JA = C(:,2);        %Computes JA vector for sparse storage scheme. NOTE:
                    %both IA and JA are based on the modified link
                    %connectivity created in Step 1 for bi-directional
                    %links. These vectors do not represent the original
                    %link connectivity

%The following lines of code use the PLCA shortest path solution to
%identify which nodes should be used as source nodes for the SDDA
solution
%process

NOW=first_node;
NEXT=[];
N = zeros(nnodes,1);
source_nodes=[];
D = zeros(nnodes,NP);
INARRAY=zeros(1,nnodes);

%Begin the shortest path solution. Loop NP times to determine correct
%number of source nodes.

for i = 1:NP
    d=inf(nnodes,1);
    d(NOW)=0;
    source_nodes(:,i) = NOW;
    while (~isempty(NOW) || ~isempty(NEXT))
        cnode=NOW(numel(NOW));
        istart=IA(cnode);
        iend=IA(cnode+1)-1;
        for location=istart:iend
            jnode=JA(location);
            if (d(jnode) > d(cnode)+1)
                d(jnode)=d(cnode)+1;
                if (INARRAY(jnode)==0)
                    NEXT(end+1)=jnode;
                    INARRAY(jnode)=1;
                end
            end
        end
        NOW(end)=[];
        INARRAY(cnode)=0;
        if isempty(NOW)
            NOW=NEXT;
            NEXT=[];
        end
    end
    D(:,i)=d;
    distance_with_rank = [sum(D,2) R];
    for j=1:size(source_nodes,2)
        distance_with_rank(source_nodes(j),:)=0;
    end
```

```
    range_D = range(D,2);
    sort_row = sortrows([transpose(1:nnodes)...
               distance_with_rank range_D], [3, -2, 4]);

    NOW = sort_row(find(sort_row(:,2)==max...
(distance_with_rank(:,1)),1),1);
end

Step_4 = toc; %Stops the stopwatch and records the time for STEP 4

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 5: POPULATE SUB-DOMAINS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic %Resets and starts the stopwatch for STEP 5

sub_domains = source_nodes; %Defines the varialble to store the sub-
domain
                            %information. It is intitated with the
source
                            %node values

available_nodes = transpose(1:nnodes);  %Defines an array of nodes
which
                                        %are available to be added to a
                                        %sub-domain

R_Orig = R; %Saves the original rank information as there will be
            %modifications made to the variable R in this step

D_Orig = D; %Similar to R, the original D values are preserved

%The following for loop removes the source nodes from the list of
available
%nodes which can be selected as well as the arrays [R] and [D]
for i = 1:size(source_nodes,2)
    j = source_nodes(1,i);
    k = find(available_nodes(:,1) == j);
    available_nodes(k,:) = [];
    D(k,:) = [];
    R(k,:) = [];
    range_D(k,:) = [];
end

%The algorithm now generates a serach matrix which allows the algorithm
%to determine the most attractive node to be added to a sub-domain

search = [available_nodes D R range_D];

next_node = zeros(nnodes-NP,NP+1);  %This variable represents the next
node
next_node(:,1) = available_nodes;   %to be added to a given sub-domain.
At
                                    %this point the variable is simply
```

```
                                        %pre-allocated

for i = 1:NP
    search = sortrows(search,[i+1,NP+2,-(NP+3)]);
    next_node(:,i+1)=search(:,1);
end

found = zeros(nnodes-NP,NP+1);
found(:,1)=available_nodes;

for i = 1:NP
[number,idx] = sort(next_node(:,i+1));
found(:,i+1) = idx;
end

processor_assignment = zeros(nnodes, 2);    %This variable assigns the
                                            %node to a sub-domain

processor_assignment(:,1) = 1:nnodes;

for i = 1:NP
    processor_assignment(source_nodes(i),2) = i;
end

j=2;

%The following while loop searches through all of the nodes and
%continues adding nodes to sub-domains until all nodes have been
%assigned.
while any(found(:,1)) == 1
    for i=1:NP
        if any(found(:,1)) == 0
            break
        else
            [val,row] = min(found(:,i+1));
            next_possible_node = found(row,1);
            connected_nodes = C(IA(next_possible_node)...
                :IA(next_possible_node +1)-1,2);
            for k = 1:length(connected_nodes)
                if any(sub_domains(:,i)==connected_nodes(k)) == 1
                    found(row,:) = nan;
                    break
                end
            end
            if isnan(found(row,:)) == 0
                next_possible_node = 0;
            end
            sub_domains(j,i) = next_possible_node;
            if next_possible_node ~= 0
            processor_assignment(next_possible_node,2) = i;
            end
        end
    end
```

```
%The next two loops are the portion of the code which assigns a value
%of 0 to a given sub-domain if a node cannot be added during a
%particular iteration.

    if sum(sub_domains(end,:)) == 0
        for i=1:NP
            if any(found(:,1)) == 0
                break
            else
                temp = sortrows(found,i+1);
                next_possible_node = temp(2,1);
                connected_nodes = C(IA(next_possible_node)...
                    :IA(next_possible_node +1)-1,2);
                for k = 1:length(connected_nodes)
                    if any(sub_domains(:,i)==connected_nodes(k)) == 1
                        idx = find(found(:,1)==next_possible_node);
                        found(idx,:) = nan;
                        temp(2,:) = nan;
                        break
                    else
                        next_possible_node = 0;
                    end
                end
                if isnan(found(idx,:)) == 0
                    next_possible_node = 0;
                end
                sub_domains(j,i) = next_possible_node;
                if next_possible_node ~= 0
                processor_assignment(next_possible_node,2) = i;
                end
            end
        end
    end

%A second loop was added in the event there was another iteration which
%still could not add a node.

    if sum(sub_domains(end-1:end,:)) == 0
        for i=1:NP
            if any(found(:,1)) == 0
                break
            else
                temp = sortrows(found,i+1);
                next_possible_node = temp(3,1);
                connected_nodes = C(IA(next_possible_node)...
                    :IA(next_possible_node +1)-1,2);
                for k = 1:length(connected_nodes)
                    if any(sub_domains(:,i)==connected_nodes(k)) == 1
                        idx = find(found(:,1)==next_possible_node);
                        found(idx,:) = nan;
                        temp(3,:) = nan;
                        break
                    else
                        next_possible_node = 0;
                    end
                end
```

```
                if isnan(found(idx,:)) == 0
                    next_possible_node = 0;
                end
                sub_domains(j,i) = next_possible_node;
                if next_possible_node ~= 0
                processor_assignment(next_possible_node,2) = i;
                end
            end
        end
    end
j = j+1;
end

Step_5 = toc; %Stops the stopwatch and records the time for STEP 5

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 6: DETERMINE WHICH NODES ARE SYSTEM BOUNDARY NODES/INTERIOR NODES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic %Resets and starts the stopwatch for STEP 6

boundary_nodes=[];  %Pre-allocates the variable as a matrix
processor_comparison = zeros(size(A,1),2); %Pre-allocation

%The following loop determines which sub-domain each node is assigned
%to. This comparison is the basis for determining which nodes are SBN.
for i = 1:size(A,1)
    processor_comparison(i,1) = processor_assignment(A(i,1),2);
    processor_comparison(i,2) = processor_assignment(A(i,2),2);
end

%The following variable, comparison_delta, makes the final
%determination if two nodes connected by a common link fall in separate
%sub-domains
comparison_delta = processor_comparison(:,1)-processor_comparison(:,2);

boundary_rows = find(comparison_delta); %Identifies which rows the
                                        %boundary nodes occur in

%The following for loop searches the rows indentified in the previous
%line and extracts the corresponding boundary node.
for i = 1:size(boundary_rows,1)
    boundary_nodes = [boundary_nodes A(boundary_rows(i),:)];
end

boundary_edges = nnz(find(comparison_delta));   %Determines the number
                                                %of boundary edges for
                                                %a direct comparison
                                                %with METIS


boundary_nodes = unique(boundary_nodes); %Stores the SBN
Number_Boundary_Nodes = size(boundary_nodes,2); %Stores the number of
                                                %SBN
```

```
%Now that the boundary nodes have been determined. The code determines
%which nodes are defined as interior nodes. This is needed for the
%reordering to be performed in Step 7.

interior_nodes = setdiff(sub_domains,boundary_nodes,'stable');

ind = find(interior_nodes == 0, 1);
interior_nodes(ind) = [];

Step_6 = toc; %Stops the stopwatch and records the time for STEP 6

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 7: RE-NUMBER THE NODES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic %Resets and starts the stopwatch for STEP 7

node_numbering = interior_nodes(:);
node_numbering = vertcat(node_numbering, transpose(boundary_nodes));
node_numbering = [transpose(1:nnodes) node_numbering];

Step_7 = toc; %Stops the stopwatch and records the time for STEP 7

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%OUTPUT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Number_Boundary_Nodes %Outputs the number of SBN

Boundary_Nodes = boundary_nodes %Outputs a list of SBN

Boundary_Edges = boundary_edges %Outputs the number of boundary edges

sub_domains %Outputs the sub-domain partitioning results

node_numbering  %Outputs the new node numbering scheme. Note the first
                %column represents the NEW node number while the second
                %column represents the OLD node number.

Step_1 %Outputs the time required to perform STEP 1
Step_2 %Outputs the time required to perform STEP 2
Step_3 %Outputs the time required to perform STEP 3
Step_4 %Outputs the time required to perform STEP 4
Step_5 %Outputs the time required to perform STEP 5
Step_6 %Outputs the time required to perform STEP 6
Step_7 %Outputs the time required to perform STEP 7

TOTAL = Step_1+Step_2+Step_3+Step_4+Step_5+Step_6+Step_7; %Total time
```

APPENDIX C: MATLAB Source Code for the DDSP Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Next, we will build on the SDDA to Solve the Shortest Path Problem.
This portion of the code represents the Domain Decomposition Based
%Shortest Path Algorithm

%This algorithm is a 4 Step algorithm which uses a domain partitioning
%approach to solve the shortest path problem. This code uses the
%classical Dijkstra method to solve the problem, however it should be
%noted any shotest path solution algorithm can be substituted for
%Dijkstra's algorithm to solve the problem.

%The code will continue the step numbering assuming Step 1 of the DDSP
%algorithm is Step 8 of this code.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 8: Partition the Network.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%This step is trivial as it has already been done by Steps 1-7 above.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 9: Solve the SP Promblem from All-to-All for Each sub-domain
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic %Resets and starts the stopwatch for STEP 9

R = accumarray(A(:,1),1); %Determine Rank for IA computation

if numel(R)<nnodes
    R(nnodes) = 0;
end

IA = zeros(nnodes + 1,1); %Recalculate IA based on original network
IA(1)=1;

for i = 2:nnodes+1
    IA(i) = IA(i-1) + R(i-1);
end

JA = sortrows(A); %Recalculate JA based on original network
JA = JA(:,2);

clear R;

D = inf(nnodes,nnodes); %Pre-allocate shortest time matrix [D]
PRED = zeros(nnodes,nnodes); %Pre-allocate shortest path matrix [PRED]

for i = 1:NP

    temp = sub_domains(:,i); %Determine sub-domain connectivity
    temp = temp(temp~=0);
```

```
    for j = 1:numel(temp)

    d = inf(nnodes,1); %pre-allocate loop variable d
    pred = zeros(nnodes,1); %pre-allocate loop variable pred

        for k = 1:numel(temp)

            source = temp(j);
            destination = temp(k);
            d(source) = 0;
            Sf = source;
            Search = [d transpose(1:nnodes)];
            Search(source,:) = nan;

            %Begin the shortest path solution based on Dijkstra
            %Solve each sub-domain independent of one another

            while (Sf(end)~= destination)
                istart = IA(Sf(end));
                iend = IA(Sf(end)+1)-1;
                for location=istart:iend
                    jnode=JA(location);
                    if (d(jnode) > d(Sf(end))+An(location))
                        d(jnode)=d(Sf(end))+An(location);
                        pred(jnode)=Sf(end);
                        Search(jnode,1) = d(Sf(end))+An(location);
                    end
                end
                [val, next] = min(Search(:,1));
                Search(next,:) = nan;
                Sf(end + 1,:) = next;
            end
            D(destination,source) = d(destination);
            PRED(destination,source) = pred(destination);
        end
    end
end

Step_9 = toc; %Stops the stopwatch and records the time for STEP 9

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 10: Solve the SP Problem from One-to-All for Each SBN to all
nodes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic %Resets and starts the stopwatch for STEP 10

for i = 1:size(boundary_nodes,2)

    d = inf(nnodes,1); %pre-allocate loop variable d
    pred = zeros(nnodes,1); %pre-allocate loop variable pred

    for j = 1:nnodes
```

```
        source = boundary_nodes(i);
        destination = j;

        d(source) = 0;
        Sf = source;

        Search = [d transpose(1:nnodes)];
        Search(source,:) = nan;

        %Begin the shortest path solution based on Dijkstra
        %Solve from each SBN to ALL other nodes in network.

        while (Sf(end)~= destination)
            istart = IA(Sf(end));
            iend = IA(Sf(end)+1)-1;
            for location=istart:iend
                jnode=JA(location);
                if (d(jnode) > d(Sf(end))+An(location))
                    d(jnode)=d(Sf(end))+An(location);
                    pred(jnode)=Sf(end);
                    Search(jnode,1) = d(Sf(end))+An(location);
                end
            end
            [val, next] = min(Search(:,1));
            Search(next,:) = nan;
            Sf(end + 1,:) = next;
        end
        D(destination, source) = d(destination);
        PRED(destination, source) = pred(destination);
    end
end

Step_10 = toc; %Stops the stopwatch and records the time for STEP 10

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%STEP 11: Check Previously Computed Values and Compute Remaining Values
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic %Resets and starts the stopwatch for STEP 11

%The algorithm now checks each value to make sure the shortest path
%does not cross sub-domain boundaries. To do so, it compares the
%distance from the source node to each SBN plus the distance from
%the same SBN to the destination node. If the calculated distance
%is shorter than what was previously computed, the algorithm updates
%both arrays [D] and [PRED]

boundary_nodes = [boundary_nodes; zeros(1,size(boundary_nodes,2))];

for i = 1:size(boundary_nodes,2)
    [row col] = find(sub_domains==boundary_nodes(1,i));
    boundary_nodes(2,i) = col;
end

boundary_nodes = transpose(boundary_nodes);
```

```
boundary_nodes = sortrows(boundary_nodes,2);

for i = 1:NP %Loop over each sub-domain
    temp_sub_domains = sub_domains(:,i);
    temp_sub_domains = temp_sub_domains(temp_sub_domains~=0);

    temp_boundary_nodes = boundary_nodes;
    temp_boundary_nodes = find(boundary_nodes(:,2)==i);
    temp_boundary_nodes = boundary_nodes(temp_boundary_nodes(1)...
        :temp_boundary_nodes(end),1);

    for j = 1:numel(temp_sub_domains) %Loop over each SBN in sub-domain
        for k = 1:nnodes
            for l = 1:numel(temp_boundary_nodes)
                if D(temp_boundary_nodes(l),temp_sub_domains(j))+...
                        D(k,temp_boundary_nodes(l))<...
                        D(k,temp_sub_domains(j));

                         D(k,temp_sub_domains(j)) = ...
            D(temp_boundary_nodes(l),temp_sub_domains(j))...
                        +D(k,temp_boundary_nodes(l));

                    PRED(k,temp_sub_domains(j)) = ...
                        PRED(k,temp_boundary_nodes(l));
                end
            end
        end
    end
end

Step_11 = toc; %Stops the stopwatch and records the time for STEP 11

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Allow user to determine if they would like to output certain O-D
%pairs. Input 1 to verify path and distance for 5 pairs. Input 0 to end
%script.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

strvcat({'Do you wish to output the calculated shortest path and', ...
'distance for 5 source to destination pairs? Enter a value of 1 to',
...
'verify. Any other value will terminate the code.'})

VERIFY = input('Please input the value of 1 to print output.');
verify_array = zeros(5,2); %pre-allocate storage array
CHECK_RANGE = zeros(numel(verify_array),1);

if VERIFY == 1 %Allows user to input 5 node pairs for verification

    while ismember(0,CHECK_RANGE) == 1

    verify_array = zeros(5,2); %pre-allocate storage array

    SOURCE1 = input ('Please enter the source node for pair 1. ')
```

```
    DEST1 = input ('Please enter the destination node for pair 1. ')

    verify_array(1,1) = SOURCE1;
    verify_array(1,2) = DEST1;

    SOURCE2 = input ('Please enter the source node for pair 2. ')
    DEST2 = input ('Please enter the destination node for pair 2. ')

    verify_array(2,1) = SOURCE2;
    verify_array(2,2) = DEST2;

    SOURCE3 = input ('Please enter the source node for pair 3. ')
    DEST3 = input ('Please enter the destination node for pair 3. ')

    verify_array(3,1) = SOURCE3;
    verify_array(3,2) = DEST3;

    SOURCE4 = input ('Please enter the source node for pair 4. ')
    DEST4 = input ('Please enter the destination node for pair 4. ')

    verify_array(4,1) = SOURCE4;
    verify_array(4,2) = DEST4;

    SOURCE5 = input ('Please enter the source node for pair 5. ')
    DEST5 = input ('Please enter the destination node for pair 5. ')

    verify_array(5,1) = SOURCE5;
    verify_array(5,2) = DEST5;

    clc

    for i = 1:numel(verify_array)
        value = ismember(verify_array(i), 1:nnodes);
        CHECK_RANGE(i) = value;
        if CHECK_RANGE(i) == 0
            strvcat({'A value entered is not a valide node number!'...
            'Please re-enter source/dest nodes to verify.'})
        end
    end

    end

    disp('The pairs you have entered are:')
    verify_array
    temp = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Compute path and distance for 1st O-D Pair
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Return the shortest time by looking in the matrix [D]
    disp('The shortest distance for pair 1 is: ')
    D1 = D(DEST1,SOURCE1)
```

```
    %Return the shortest path by traversing the matrix [PRED]
    disp('The shortest path for pair 1 is: ')
    PATH1 = [DEST1];
    k = DEST1;
    for i = 1:nnodes
        temp = PRED(k, SOURCE1);
        if temp == 0;
            PATH1 = 'No path exists from this source to destination.'
            break
        else
            PATH1 = [PATH1 temp];
            k = temp;
        end
        if temp == SOURCE1
            break
        end
    end

    %Usting this method, the path is returned in the reverse order.
    %The next line loop returns the transpose of the computed path.

    if temp ~= 0
    PATH1 = flipdim(transpose(PATH1),1)
    end

    d_path = 0;
    dd = 0;

    for i = 1:numel(PATH1)-1 %Generate shortest path
        dd = D(PATH1(i+1),PATH1(i));
        d_path = d_path+dd;
    end

    %Verify the correct shortest time was achieved by summing the
    %value for each step along the shortest path. Due to numerical
    %accuracy of MATLAB it has been determined that if the two
    %paths are within 0.1% the correct solution has been found.

    if abs(d_path-D1)/d_path < 0.001
        disp('Path yields correct shortest time. ')
    else
        disp('Path does not yield correct shortest time. ')
    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Compute path and distance for 2nd O-D Pair
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Return the shortest time by looking in the matrix [D]
    disp('The shortest distance for pair 2 is: ')
    D2 = D(DEST2,SOURCE2)

    %Return the shortest path by traversing the matrix [PRED]
    disp('The shortest path for pair 2 is: ')
    PATH2 = [DEST2];
```

```
    k = DEST2;
    for i = 1:nnodes
        temp = PRED(k, SOURCE2);
        if temp == 0;
            PATH2 = 'No path exists from this source to destination.'
            break
        else
            PATH2 = [PATH2 temp];
            k = temp;
        end
        if temp == SOURCE2
            break
        end
    end

    %Usting this method, the path is returned in the reverse order.
    %The next line loop returns the transpose of the computed path.

    if temp ~= 0
    PATH2 = flipdim(transpose(PATH2),1)
    end

    d_path = 0;
    dd = 0;

    for i = 1:numel(PATH2)-1
        dd = D(PATH2(i+1),PATH2(i));
        d_path = d_path+dd;
    end

    %Verify the correct shortest time was achieved by summing the
    %value for each step along the shortest path. Due to numerical
    %accuracy of MATLAB it has been determined that if the two
    %paths are within 0.1% the correct solution has been found.

    if abs(d_path-D2)/d_path < 0.001
        disp('Path yields correct shortest time. ')
    else
        disp('Path does not yield correct shortest time. ')
    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Compute path and distance for 3rd O-D Pair
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Return the shortest time by looking in the matrix [D]
    disp('The shortest distance for pair 3 is: ')
    D3 = D(DEST3,SOURCE3)

    %Return the shortest path by traversing the matrix [PRED]
    disp('The shortest path for pair 3 is: ')
    PATH3 = [DEST3];
    k = DEST3;
    for i = 1:nnodes
        temp = PRED(k, SOURCE3);
```

```
        if temp == 0;
            PATH3 = 'No path exists from this source to destination.'
            break
        else
            PATH3 = [PATH3 temp];
            k = temp;
        end
        if temp == SOURCE3
            break
        end
    end


    %Usting this method, the path is returned in the reverse order.
    %The next line loop returns the transpose of the computed path.

    if temp ~= 0
    PATH3 = flipdim(transpose(PATH3),1)
    end

    d_path = 0;
    dd = 0;

    for i = 1:numel(PATH3)-1
        dd = D(PATH3(i+1),PATH3(i));
        d_path = d_path+dd;
    end

    %Verify the correct shortest time was achieved by summing the
    %value for each step along the shortest path. Due to numerical
    %accuracy of MATLAB it has been determined that if the two
    %paths are within 0.1% the correct solution has been found.

    if abs(d_path-D3)/d_path < 0.001
        disp('Path yields correct shortest time. ')
    else
        disp('Path does not yield correct shortest time. ')
    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Compute path and distance for 4th O-D Pair
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Return the shortest time by looking in the matrix [D]
    disp('The shortest distance for pair 4 is: ')
    D4 = D(DEST4,SOURCE4)

    %Return the shortest path by traversing the matrix [PRED]
    disp('The shortest path for pair 4 is: ')
    PATH4 = [DEST4];
    k = DEST4;
    for i = 1:nnodes
        temp = PRED(k, SOURCE4);
        if temp == 0;
            PATH4 = 'No path exists from this source to destination.'
```

```
            break
        else
            PATH4 = [PATH4 temp];
            k = temp;
        end
        if temp == SOURCE4
            break
        end
    end

    %Usting this method, the path is returned in the reverse order.
    %The next line loop returns the transpose of the computed path.

    if temp ~= 0
    PATH4 = flipdim(transpose(PATH4),1)
    end

    d_path = 0;
    dd = 0;

    for i = 1:numel(PATH4)-1
        dd = D(PATH4(i+1),PATH4(i));
        d_path = d_path+dd;
    end

    %Verify the correct shortest time was achieved by summing the
    %value for each step along the shortest path. Due to numerical
    %accuracy of MATLAB it has been determined that if the two
    %paths are within 0.1% the correct solution has been found.

    if abs(d_path-D4)/d_path < 0.001
        disp('Path yields correct shortest time. ')
    else
        disp('Path does not yield correct shortest time. ')
    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Compute path and distance for 5th O-D Pair
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Return the shortest time by looking in the matrix [D]
    disp('The shortest distance for pair 5 is: ')
    D5 = D(DEST5,SOURCE5)

    %Return the shortest path by traversing the matrix [PRED]
    disp('The shortest path for pair 5 is: ')
    PATH5 = [DEST5];
    k = DEST5;
    for i = 1:nnodes
        temp = PRED(k, SOURCE5);
        if temp == 0;
            PATH5 = 'No path exists from this source to destination.'
            break
        else
            PATH5 = [PATH5 temp];
```

```
            k = temp;
        end
        if temp == SOURCE5
            break
        end
    end

    %Usting this method, the path is returned in the reverse order.
    %The next line loop returns the transpose of the computed path.

    if temp ~= 0
    PATH5 = flipdim(transpose(PATH5),1)
    end

    d_path = 0;
    dd = 0;

    for i = 1:numel(PATH5)-1
        dd = D(PATH5(i+1),PATH5(i));
        d_path = d_path+dd;
    end

    %Verify the correct shortest time was achieved by summing the
    %value for each step along the shortest path. Due to numerical
    %accuracy of MATLAB it has been determined that if the two
    %paths are within 0.1% the correct solution has been found.

    if abs(d_path-D5)/d_path < 0.001
        disp('Path yields correct shortest time. ')
    else
        disp('Path does not yield correct shortest time. ')
    end
end
```

APPENDIX D: Example Input/Output for the SDDA

This appendix will demonstrate an example input and associated output for the SDDA utilizing the 15 Node example shown in Chapter 2 of this manuscript, which is given by the following figure:

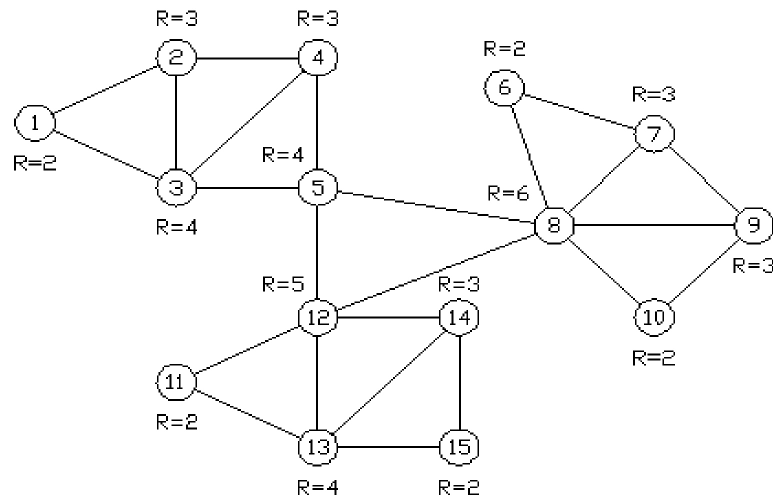


Figure D1: Example Network

As can be seen by the source code provided by Appendix B above, simply input the number of sub-domains (given by variable NP) when prompted. For this example, the following input data was provided to the program (via .txt file). The first column represents the source node, the second represents the destination node, and the final column is the cost to travel from the source to the destination:

```
1 2 1
1 3 1
3 2 1
2 4 1
3 4 1
3 5 1
5 4 1
5 12 1
5 8 1
12 8 1
6 8 1
6 7 1
7 8 1
7 9 1
```

8 9 1
8 10 1
9 10 1
11 12 1
11 13 1
12 13 1
12 14 1
13 14 1
13 15 1
14 15 1
2 1 1
3 1 1
2 3 1
4 2 1
4 3 1
5 3 1
4 5 1
12 5 1
8 5 1
8 12 1
8 6 1
7 6 1
8 7 1
9 7 1
9 8 1
10 8 1
10 9 1
12 11 1
13 11 1
13 12 1
14 12 1
14 13 1
15 13 1
15 14 1

Once the algorithm reads the text file, a number of output functions are available. Again referring to Appendix B, one can see the output variables available directly after Step 7. These variables are:

Number_Boundary_Nodes
Boundary_Nodes
Boundary_Edges
sub_domains

node_numbering

Step_1
Step_2
Step_3
Step_4
Step_5
Step_6
Step_7

TOTAL

The output provided by each of these variables is presented below. If one is to call the function Number_Boundary_Nodes, the algorithm simply outputs the number of SBN the given partition provides. In this example with NP = 3, the result of this call is:

```
Number_Boundary_Nodes =

     3
```

The next output argument is Boundary_Nodes. This call provides the actual boundary nodes returned by the algorithm. The result of this call is:

```
Boundary_Nodes =

     5     8    12
```

The next output argument is Boundary_Edges. This is similar to the first output; however, instead of reporting the number of nodes, this actually returns the number of edges or links connecting the boundary nodes. For this example, when this function is called, the algorithm returns:

Boundary_Edges =

6

It is important to note that the boundary edges for this partition are represented by the three links connecting nodes 5, 8, and 12. Since the links are bi-directional, a value of 6 is returned, instead of 3. The next available output variable is sub_domains. This returns the actual partitions provided by the algorithm in terms of which nodes have been assigned to a given sub-domain. The output is structured in a column-wise format, where each column represents a sub-domain. The input for this problem requires the network to be sub-structured into three sub-networks. The user should expect three columns of data, similar to the following:

sub_domains =

```
    1    15     6
    2    14     7
    3    13     8
    4    11    10
    5    12     9
```

The last step of the SDDA algorithm is to renumber the nodes. The revised node numbering can be realized by the output variable named node_numbering. This variable returns two columns of data where the first column represents the new node number and the second column represents the original node number as follows:

node_numbering =

```
 1    1
 2    2
 3    3
 4    4
 5   15
 6   14
 7   13
 8   11
 9    6
10    7
11   10
12    9
13    5
14    8
15   12
```

While performing these steps, the algorithm records the time for each step, as well as a total aggregate time. When calling these functions, the following results are returned:

Step_1 =

0.0039

Step_2 =

0.0033

Step_3 =

0.0011

Step_4 =

0.0090

Step_5 =

0.0238

Step 6 =

0.0029

Step_7 =

9.2229e-04

TOTAL =

0.0449

APPENDIX E: Example Input/Output for the DDSP Algorithm

Utilizing the same example shown in Figure D1, the DDSP algorithm returns the shortest distance and path for a given problem. There is no input required, since all input parameters have previously been provided as part of the SDDA. At the end of the SDDA, the program transitions directly into the DDSP algorithm, based on the partitions and output returned by the SDDA. The main output of the DDSP algorithm is simple. It returns two arrays: namely, the array [D] which stores all the shortest time information for each O-D pair, and the array [PRED] which stores the predecessor information for every possible O-D pair. If one wanted to know the shortest time and predecessor for a single source to single destination, the input would be as follows (assuming a source node of 1 to a destination node of 5):

```
EDU>> D(5,1)
ans =
    2
EDU>> PRED(5,1)
ans =
    3
```

If the entire matrix is desired, simply omit the indices shown above and the full matrix for each variable will be returned. As another way to return a set of O-D pairs, the algorithm has been written with an option to return solutions for five pairs of data. Suppose one wants to find the solutions to the pairs 1-5, 5-1, 1-15, 5-10, 8-4. Once the algorithm has completed its running, it will prompt the user with the following question:

```
ans =
Do you wish to output the calculated shortest path and
distance for 5 source to destination pairs? Enter a value of 1 to
verify. Any other value will terminate the code.
```

By entering a value of 1 when prompted, the program will then prompt the user to enter five separate source and destination nodes. After inputting the pairs indicated above, the output appears as follows:

```
The pairs you have entered are:
verify_array =
    1     5
    5     1
    1    15
    5    10
    8     4

The shortest distance for pair 1 is:
D1 =
    2
The shortest path for pair 1 is:
PATH1 =
    1
```

3

5

Path yields correct shortest time.

The shortest distance for pair 2 is:

D2 =

2

The shortest path for pair 2 is:

PATH2 =

5

3

1

Path yields correct shortest time.

The shortest distance for pair 3 is:

D3 =

5

The shortest path for pair 3 is:

PATH3 =

1

3

5

12

13

15

Path yields correct shortest time.

The shortest distance for pair 4 is:

D4 =

2

The shortest path for pair 4 is:

PATH4 =

5

8

10

Path yields correct shortest time.

The shortest distance for pair 5 is:

D5 =

2

The shortest path for pair 5 is:

PATH5 =

8

5

4

Path yields correct shortest time.

VITA

Paul W. Johnson, III

Education

Ph.D. in Civil Engineering	Old Dominion University	Norfolk, VA	Dec. 2016
M.S. in Civil Engineering	Clemson University	Clemson, SC	Dec. 2008
B.S. in Civil Engineering	Old Dominion University	Norfolk, VA	Dec. 2006

Professional Experience

05/2014 – Present	Ecospan Engineer	Nucor – Vulcraft/Verco Group
07/2009 – 05/2014	Design Manager	Naval Facilities Engineering Command
06/2007 – 07/2009	Engineer I	O'Neal, Inc.
10/2005 – 05/2007	Staff Engineer	The LandMark Design Group

Publications and Presentations

Johnson III, P. W. (2016). *Efficient Domain Decomposition Algorithms and Applications in Transportation and Structural Engineering*. (Unpublished Doctoral Dissertation). Old Dominion University, Norfolk, VA.

Johnson, P., Nguyen, D., & Ng, M. (2016). "Domain Decomposition Based Shortest Path Algorithm." Submitted for Publication. In *Transportation Research Board 96th Annual Meeting.*

Johnson, P., Nguyen, D., & Ng, M. (2016). Large-scale network partitioning for decentralized traffic management and other transportation applications. *Journal of Intelligent Transportation Systems*, 1-13.

Johnson III, P. W., Nguyen, D., & Ng, M. (2014). An Efficient Shortest Distance Decomposition Algorithm for Large-Scale Transportation Network Problems. In *Transportation Research Board 93rd Annual Meeting* (No. 14-2921).

Allen, S., **Johnson, P.**, Nguyen, D., & Ng, M. (2013). "*Use of LCA for Developing the Automated Domain Decomposition Partitioning Algorithm*." 2013 SIAM Conference, Old Dominion University. Webb Center, Norfolk, VA. 13. Conference Presentation.

Rassati, G. A., Fortney, P. J., Shahrooz, B. M., & **Johnson, P. W.** (2011). Performance Evaluation of Innovative Hybrid Coupled Core Wall Systems. In *Composite Construction in Steel and Concrete VI* (pp. 479-492). ASCE Publications.

Johnson, P. W. (2008). *PM Characteristics of Reinforced Concrete Sections* (Master's Thesis). Clemson University. Clemson, SC.

Licenses

Licensed Professional Engineer in the Commonwealth of Virginia (License # 47148)