ENHANCED THROUGHPUT FOR WORKFLOW SCHEDULING USING PARALELLISM COMPUTATION AND INFORMED SEARCH

By

Kaite Tang

A thesis submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

Middle Tennessee State University

August 2017

Thesis Committee:

Dr Yi Gu,

Dr Joshua L. Phillips,

Dr Cen Li

Dr Medha Sarkar

ACKNOWLEDGEMENTS

This thesis would not have become possible without the support of the Department of Computer Science at Middle Tennessee State University. I am grateful to all of those with whom I have had the pleasure to work during my Master's study.

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Yi Gu, for her excellent guidance, caring, patience, and providing me with an excellent atmosphere for doing research. I have learned not only academic knowledge and expertise, but also methods and skills of how to do the research, which will benefit my entire life.

I would also like to thank the members of my committee, Dr. Joshua Phillips, Dr. Cen Li, Dr. Medha Sarkar, for their valuable comments and suggestions, leading to a successful thesis.

Finally, I am grateful to my parents, Mrs. Honglu Zhou and Mr. Jianping Tang, for their spiritual and physical supports. Without their unconditional, unchanging, and unending love and care, this work would never have come into existence.

ABSTRACT

Next-generation e-science is producing a huge amount of data that needs to be processed by geographically isolated scientists and users through different steps, which can be modeled as a Directed Acyclic Graph (DAG) structured computing workflow. Many big data science applications, especially streaming applications with complex DAG-structured workflows, require a smooth dataflow for the Quality of Service (QoS) guarantee. Even with the ever-increasing computing power available in the High Performance Computing (HPC) environments, i.e., parallel processing on a PC cluster, the execution time of such high-demand streaming applications may still take a few hours or even days in some cases. Therefore, supporting and optimizing the performance of such scientific workflows in wide-area networks, especially in Grid and Cloud environments, are crucial to the success of collaborative scientific discovery.

In this thesis, we focus on optimizing and improving the performance of an existing workflow mapping algorithm, Layer-oriented Dynamic Programming (LDP), by (i) parallelizing the workflow executions on a PC cluster using MPI and OpenMP, and (ii) removing unnecessary search steps in order to reduce the algorithm runtime using informed search techniques inspired by depth-first search (DFS) and breadth-first search (BFS). The performance superiority of the modified algorithm is illustrated by an extensive set of simulations in comparison to the original LDP algorithm in terms of both throughput and runtime.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

BFS – Breadth-First Search

DFS – Depth-First Search

DP – Dynamic Programming

HPC – High Performance Computing

LDP – Layer-oriented Heuristic Algorithm Using a DP Procedure

MED – Minimum End-to-end Delay

MFR – Maximum Frame Rate

MPI – Message Passing Interface

CHAPTER I

INTRODUCTION

Next-generation e-science is producing a huge amount of data that needs to be processed by geographically isolated scientists and users through different steps, which can be modeled as a Directed Acyclic Graph (DAG) structured computing workflow. Many big data science applications, especially streaming applications with complex DAG-structured workflows require a smooth data flow for the Quality of Service (QoS) guarantee. Even with the ever-increasing computing power available in High Performance Computing (HPC) environments, i.e., parallel processing at a PC cluster, the execution time of such high demand streaming applications may still take a few hours or even days in some cases. Thus, how to carefully choose an appropriate set of computer nodes and wisely allocate available computing resources to the tasks are very critical for end-to-end performance of the workflows. For example, task A is the preceding task of task B and has a large data transfer to task B. If we choose two inappropriate computer nodes connected through slow bandwidth, the data transfer time from task A to task B would determine the throughput of entire workflow execution, which is known as a *bottleneck* (BN), and also increase the latency of workflow executions.

In this thesis, we consider a streaming application modeled as a DAG-structure workflow consisting of a set of computing modules with intricate inter-module dependencies. Each module receives data (input) from each of its preceding modules and sends data (output) to each of its succeeding modules, respectively. We define the underlying computer network as a distributed heterogeneous environment consisting of a set of computer nodes with different computing capabilities and communication links with different bandwidths

and minimum link delays. The network is represented by a directed weighted graph, where vertices represent computer nodes and edges denote communication links. Nodes may not always have direct connections to each other as they may have different security policies or belong to different domains. Note that when several concurrent modules are mapped onto the same node and execute tasks simultaneously, the node's processing power is shared by a fair manner among those modules. The same policy applies when several datasets are transferred on the same communication link. The bandwidth of that link is fairly shared by the concurrent data transfers.

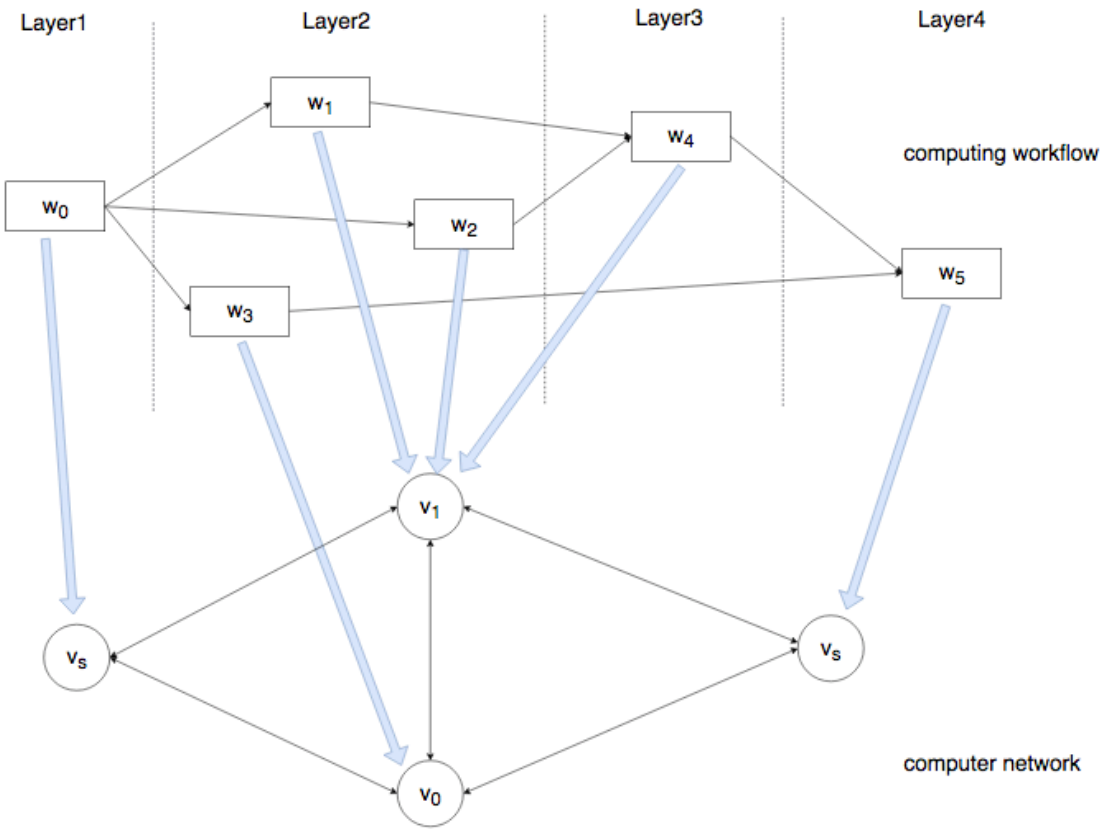

Figure 1. An Example of Workflow Mapping

To illustrate the complexity of resource sharing among concurrent module executions and data transfers, we use a smplified workflow and network secnario in Figure 1 as an example, which consists of six computing modules and four computer nodes. Modules have dependencies to each other and only start after all preceding modules finish execution and transfer their datasets to subsequent module. In this example, we have node v_s for the first module w_0 which starts workflow execution and node v_d for the last module which produces the final results for the workflow. The computer network is not a completely connected network, for example, the start node v_s has no direct connection to the destination node v_d.

The mapping scheme assigns modules w_1, w_2, and w_4 to node v_1, where modules w_1 and w_2 may execute simultaneously and share computing resource of node v_1, while module w_4 can exclusively use the resource of node v_1 during its execution time since w_4 is dependent on modules w_1 and w_2. Many factors must be considered to make the whole workflow execute efficiently, such as whether to choose a new node or reuse the previous node, or to find all possible nodes and pick the best candidate for the module.

Since the workflow is represented by a DAG, we can sort different modules into different layers based on their dependencies, which is known as topological sorting in graph theory. In Figure 1, w_0 is in layer 1; w_1, w_2, w_3 are in layer 2; w_4 is in layer 3; w_5 is in layer 4. Modules within the same layer may or may not execute at the same time and thus may or may not share resources. However, modules in different layers will not share resources even if they are mapped to the same node because they are dependent on each other. The difficulty of these problems essentially arises from the topological matching nature in the

spatial domain, which is further compounded by the resource sharing complicacy in the temporal dimension.

In this thesis, we analyze and improve the performance of an existing Layer-oriented Dynamic Programming (LDP) algorithm to achieve better throughput and algorithm runtime, respectively. We parallelize workflow executions on a PC cluster using MPI and OpenMP and narrow the search space in order to reduce algorithm runtime using informed search techniques inspired by depth-first search (DFS) and breadth-first search (BFS) algorithms. The superiority of the proposed solutions is verified by an extensive set of performance comparisons with the original LDP algorithm.

The rest of the thesis is organized as below. In Chapter II, we conduct an extensive and thorough literature review. In Chapter III, we mathematically formulate the cost models and the workflow mapping problem, and briefly introduce the existing algorithm to be improved. We provide technical solutions in Chapter IV, including pseudocode and implementation details. Experimental results and comparisons are presented in Chapter V, and Chapter VI concludes our work.

CHAPTER II

RELATED WORK

Workflow mapping problems have been studied in many disciplines because of their importance and practicality [4,5,6]. The workflow mapping problem usually focuses on two perspectives. One is to assign the subtasks in a computing workflow to an appropriate set of selected computer nodes in order to achieve a certain Quality of Service (QoS) guarantee. The other is to decide the order and resources shared on a computer node or processor when running multiple concurrent tasks, which is not the focus of this thesis.

There are many existing workflow management systems with different mapping algorithms [7,8,9,10]. For example, Apache Hadoop has Yarn as MapReduce2, which mainly focuses on processing and splitting massive data and assigning on different nodes [8]. It has a capacity scheduler which focuses on managing shared resource by virtualizing computing resources to a predefined number of segments and assigns them to different tasks. However, the Yarn scheduler does not support DAG workflows which means the users have to manage the dependencies of tasks manually. Also, Yarn does not have a mechanism to select which node would be appropriate to map. Condor DAGMan utilizes a Round Robin approach which is an algorithm for allocating tasks among a group of eligible resources on a cyclic basis [7]. Condor supports DAG workflows by defining the dependency between tasks in XML file before workflow execution. Some workflow management systems, such as Spark, not only have their own scheduler but can also use frameworks such as Mesos or Hive [9,10]. For most of the workflow management applications, the scheduler does not take the computer network environment and hardware condition of individual nodes into

consideration to achieve maximum throughput. Instead, they mainly focus on maintaining the task execution order or balancing resource utilization. In a heterogeneous network environment, the bandwidth of network links vary. If we assign data transfers to a network link with low bandwidth, it may take a long time to move the data. In addition, since the hardware conditions of computer nodes are different, the workflow tasks may be mapped to some nodes that do not have enough processing power while other idle nodes may have enough powerful processors. Therefore, a well-balanced workflow mapping algorithm is crucial for end-to-end performance in heterogeneous network environments.

Based on when the mapping is performed, workflow mapping algorithms can be classified into static algorithms and dynamic algorithms. In a static algorithm, the entire mapping is generated before the workflow is actually executed, while in a dynamic algorithm, the mapping results are calculated dynamically during workflow execution [2].

There are several existing workflow algorithms, (i) Greedy A^* [18] is a scheduling algorithm that determines a static allocation of modules among a set of sensor nodes; (ii) Streamline[17] maps the best resources to the most needy modules; (iii) Greedy makes the locally optimal choice at each search stage; (iv) LDP[2] is a static workflow mapping algorithm that focuses on how to obtain maximum throughput during resource allocation. In [2], the author implements and tests four algorithms in several random generated problem cases with different problem size. The experiment shows that the MFR performance of LDP algorithm outperforms others in all problem cases.

The LDP algorithm first sorts the modules into different layers based on their dependencies, and then layer-by-layer assigns the modules in the current layer to computer nodes according to the modules' computational requirements using a dynamic programming

procedure. However, the original LDP algorithm has a high time complexity, resulting in a long runtime when the sizes of the computing workflow and computer network increase, respectively. For instance, it may take several hours to calculate the mapping scheme when we try to map a computing workflow of around 100 modules to a network consisting of around 100 computer nodes and 10000 communication links. Moreover, LDP may miss some of the possible solutions in the search tree when the previous layer has a relatively high bottleneck. To produce smooth data flow in streaming applications and achieve better algorithm efficiency, we aim to optimize and improve LDP algorithm in terms of both maximum frame rate (MFR) and runtime.

CHAPTER III

WORKFLOW MAPPING PROBLEM

Cost Model

We specify the cost models for both computing workflows and computer networks. For convenience, we tabulate the parameters defined in the cost models in Table 1. The workflow can be modeled as a Directed Acyclic Graph $G_w = (V_w, E_w)$, $|V_w|$ = m, where each computing model is represented as $w_i \in V_w$. The workflow starts from module w_0 and ends at module w_{m-1}. The dependency between two modules can be represented as a directed edge $e_{i,j} \in E_w$. Module w_j receives data input from each of its preceding modules, including module w_i. Each module has its own computational requirement CR_i (i ∈ 0,1 … m – 1), which is calculated as $z_{w_i} * \lambda_i$, where z_{w_i} is aggregated input data size of module w_i and λ_i is computational complexity of w_i. Note that the complexity of a module is an abstract quantity that not only depends on the computational complexity of the algorithm defined in the module but also the implementation details such as the specific data structures used in the program. In our workflow model, we assume there is always one start module and one end module. An application with multiple start or end modules could be converted to a single start and a single end model by inserting a virtual start or end module of complexity zero connected to all source or destination modules with zero-sized output or input data transfers. All other modules in the workflow may receive input datasets from one or more preceding modules and generate output datasets to one or more succeeding modules.

The computer network can be modeled as a weighted graph $G_c = (V_c, E_c)$, where $|V_c| = n$ nodes and are connected by $|E_c|$ overlay links. A node is represented as v_i (i ∈ 0,1…n – 1). Each node has its own processing power as p_i, and r_i represents the number of modules that are currently running on the node v_i. $l_{i,j}$ is the link between v_i and v_j and it has the bandwidth $b_{i,j}$ and the minimum link delay $d_{i,j}$. We assume there is a source node v_s to run the start module and a destination node v_d to execute the end module. We use L_i (i ∈ 1, 2…k) to represent the workflow layer, where $|L_i| = k$, L_1 has the start module mapped to the start node, and L_k has the end module mapped to the end node.

Table 1. Parameters in the cost models and problem formulation

Parameters	**Definitions**
$G_w = (V_w, E_w)$	Workflow graph
w_i	Module
e_i	Dependency between modules
z_{w_i}	Aggregated input data size of module w_i
CR_i	Computational requirement
λ_i	Computational complexity
$G_c = (V_c, E_c)$	Network graph
v_i	Node
p_i	Node processing power
$l_{i,j}$	Network link
$b_{i,j}$	Bandwidth
$d_{i,j}$	Minimum delay
r_i	Node reuse counter
L_i	Workflow layer
$T_{tran(e,l)}$	Data transfer time of edge e over link l
$T_{exec(w,v)}$	Execution time of module w on node v
BN_i	Bottleneck for one layer

Performance Metric

When we map a module to a node, we have $T_{tran(e_i,l_{k,j})}$, which is the data transfer time of edge e_i over link l_j, and $T_{exec(w_i,v_j)}$, which is execution time of module w_i on node v_j. We use BN_i to define local bottleneck in each layer, which is calculated by selecting the longest time among all module execution times and data transfer times computed from the start module to every other module in the current layer. For each BN_i, we find the largest one as the global bottleneck time BN_{global}. The execution time of module w_i mapped to node v_j is defined as $\frac{CR_{\mathrm{w}_i} * r_{\mathrm{v}_j}}{p_{\mathrm{v}_j}}$. Since the node's processing power is shared in a fair manner if there are multiple modules running on same node, where r_j is the counter which indicates how many modules are running on node j. For the data transfer from node v_k to node v_j, the transfer time is calculated as $\frac{CR_{\mathrm{w}_i} * r_{\mathrm{v}_j}}{b_{k,j}} + d_{k,j}$. To find the bottleneck of the current layer, we need to find all the module execution times and data transfer times in that layer. So the global bottleneck time can be computed as:

$$\mathrm{BN}_{global} = \max_{w_i,e_i \in G_w\, v_j,l_j \in G_c} (T_{tran(e_i,l_{k,j})}, T_{exec(w_i,v_j)})$$

$$= \max_{w_i,e_i \in G_w\, v_j,l_j \in G_c} (\frac{CR_{\mathrm{w}_i} * r_{\mathrm{v}_j}}{p_{\mathrm{v}_j}}, \frac{CR_{\mathrm{w}_i} * r_{\mathrm{v}_j}}{b_{k,j}} + d_{k,j})$$

Frame rate or throughput is the inverse of global bottleneck time of the workflow. It can be considered as the rate that data is produced at the last module, and the frame rate is the most important performance index for streaming applications which continuously generate datasets and feed them into workflow.

Problem Formulation

The formal workflow mapping problem for MFR is defined as follows:

Definition 1. Given a DAG- structured computing workflow $G_w = (V_w, E_w)$ and a heterogeneous computer network $G_c = (V_c, E_c)$, the objective of the problem is to find an appropriate mapping scheme that assigns each module to a node so that the mapped workflow achieves the MFR, i.e.:

$$\text{MFR} = \max_{all\ possible\ mappings} \left(\frac{1}{\text{BN}_{global}}\right)$$

where the maximum frame rate is analogous to producing smooth data flow in streaming applications.

CHAPTER IV

TECHNICAL SOLUTIONS

Analysis of the Original LDP Algorithm

Input: workflow graph $G_w = (V_w, E_w)$, network graph $G_c = (V_c, E_c)$
Output: workflow mapping
1: Topology sort G_w and categorize modules to different layer
2: For each layer L_i:
3: get the module w_{Li} in layer L_i
4: sort w_{Li} by CR
5: for each w_{Li}:
6: get all node available for w_{Li}
7: end for
8: initialize a 2 dimension DP matrix M, the row stands for one mapping combination as COMB
9: set result bottleneck BN_{res} and $COMB_{res}$ empty
10: for $COMB_j$ in M:
11: get the BN_{COMB_j} for one combination
12: if ($BN_{COMB_j} < BN_{res}$)
13: update $BN_{res} = BN_{COMB_j}$
14: update $COMB_{res} = COMB_j$
15: end if
16: end for
17: save the mapping with minimum BN for current layer
18: end for

Figure 2. Original LDP algorithm

Figure 2 shows the pseudocode of the original LDP algorithm. The algorithm tries to find a proper mapping for the current layer with the minimum global bottleneck based on the mapping in the previous layer, and then uses the mapping generated in this layer to calculate the bottleneck in the next layer. For each module, based on the nodes where their preceding modules are mapped, we can calculate a list of possible nodes for the current

module that are connected to the previously selected nodes, and list all the combinations for different mappings. We choose the mapping with the smallest bottleneck for the current layer, and repeat this process for every layer until we obtain the global bottleneck of the entire workflow which produces the maximum frame rate among all possible mapping schemes. Sometimes the original LDP algorithm will calculate one bottleneck for every map combination. It may not be able to choose optimal mapping in current layer when there is a module or link from the previous layer with a higher bottleneck than the module in the current layer.

Figure 3 shows an example when the bottleneck of the previous layer is larger than the bottleneck of the current layer. We assume the mapping results for layer 1 and layer 2 have already been calculated. In layer 1, module w_0 is mapped to node v_0. In layer 2, w_1 is mapped to v_1, and w_2 is mapped to v_2 and has a longer module execution time. When we try to map w_4 in layer 3, v_2 is still running during bottleneck time 200. From the previous mapping results and the network topology we know that module w_4 can be mapped to 4 possible nodes, which are v_1, v_2, v_3 and v_5, respectively. If we map w_4 to v_2, which already has module w_2 running on it, it would increase the global bottleneck and thus the algorithm will select other nodes. In Figure 3, if we map w_4 to v_1, the bottleneck would be 100, and nodes v_4 and v_5 will have bottlenecks of 50 and 30, respectively. Since the original algorithm will only return a global bottleneck to make sure the overall performance of the workflow is optimized, we may miss some of the better mapping solutions.

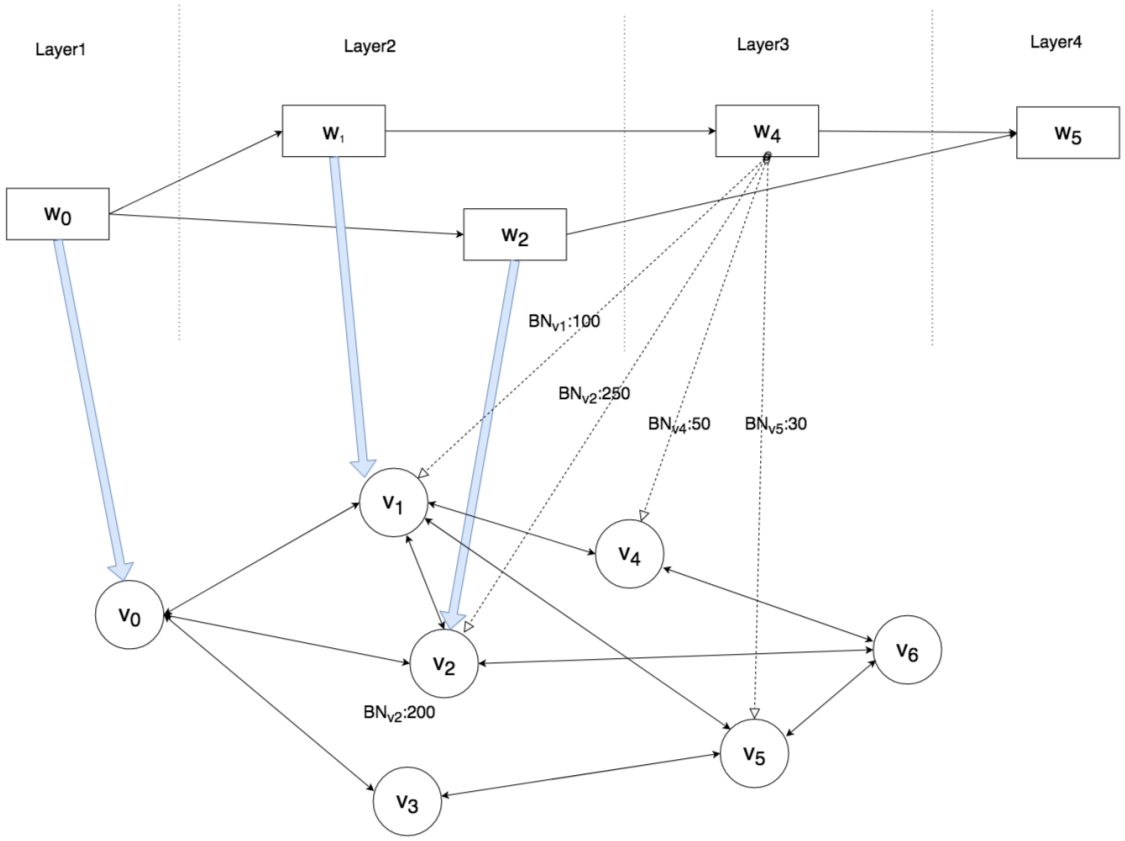

Figure 3. An example of the original LDP algorithm without checking the local minimum bottleneck

Algorithm Runtime Analysis

The original LDP algorithm has high time complexity. The runtime gets very large when executing a workflow with a large number of modules and nodes. Here we analyze the time complexity by workflow layers. On average, we assume there are n modules in each layer. Each module has k proceeding modules and m candidate nodes can be used. From Figure 2 line 4 we know that we need to sort modules based on their computational requirements in the current layer. If we use quick sort or merge sort, the time complexity is $o(n\log n)$[14,15]. From Figure 2 line 5 and 6 we need to find all available mapping nodes

of each module. This needs an intersection operation to all possible mapping nodes for the module in the previous mapping, since the intersection operation for two sets of nodes is m^2 and each module on average has k proceeding modules[16]. The time complexity to find all available mapping nodes is o($n * k * m^2$). From Figure 2 line 8 to line 16 we generate all possible combinations for the mappings and calculate the global minimum bottleneck. We need to generate a DP array with size $n * m^n$ and traverse the entire table to calculate the bottleneck for each combination. The bottleneck calculation for each combination has o(n) time complexity. The time complexity of finding the minimum bottleneck is o($n^2 * m^n$) . The time complexity of the mapping process in one layer is o($n^2 * m^n$ + nlog $n + n * k * m^2$). Therefore, increasing the sizes of the workflow or the network will dramatically increase the algorithm's runtime. Since the process of finding the minimum bottleneck takes up most of the time, we should use better search schemes to speed up this process.

Optimization On Maximum Frame Rate Performance

Sometimes the minimum bottleneck among all current combination mappings is not in the current layer but in the previous layers. Since the original algorithm only returns a global bottleneck value for the current combination mapping, there are several combination mappings that all have the same bottleneck which is the minimum bottleneck, and we may get a random selection in the current layer mapping. To solve this problem, two values are generated every time after we finish the bottleneck calculation for one combination mapping. One value holds the global bottleneck of the combination mapping and the other holds the bottleneck for the current layer. If we find a smaller global bottleneck, we will update the new global bottleneck, the new bottleneck of the current layer and the new

combination mapping. If we find the combination mapping has the same global bottleneck but smaller bottleneck for the current layer, we update the new bottleneck in the current layer and the new combination mapping. The pseudocode of the newly improved LDP algorithm is presented in Figure 4.

Input: workflow graph $G_w = (V_w, E_w)$, network graph $G_c = (V_c, E_c)$
Output: workflow mapping
1: Topologically sort G_w and categorize modules to different layers
2: For each layer L_i:
3: get the module w_{Li} in layer L_i
4: sort w_{Li} by CR
5: for each w_{Li}:
6: get all node available for w_{Li}
7: end for
8: initialize a 2 dimension DP matrix M, the row stands for one mapping combination as COMB
9: set result bottleneck BN_{res} and $COMB_{res}$ empty
10: for $COMB_j$ in M:
11: get the BN_{COMB_j} for one combination
12: if ($BN_{COMB_j} < BN_{res}$)
13: update $BN_{res} = BN_{COMB_j}$
14: update $COMB_{res} = COMB_j$
15: update $BN_{curLayerRes}$
16: else if ($BN_{COMB_j} == BN_{res}$ && $BN_{curLayer} < BN_{curLayerRes}$)
17: update $BN_{curLayerRes}$
18: update $COMB_{res} = COMB_j$
19: end if
20: end for
21: save the mapping with minimum BN for current layer
22: end for

Figure 4. Improved LDP algorithm

Optimization On Algorithm Runtime Performance

When the algorithm processes the workflow with a large number of modules and nodes, the runtime of the LDP algorithm largely increases. We design three approaches to optimize the runtime of the algorithm. The first one is the optimization on runtime of multiple workflow mappings. We use MPI to parallel process multiple workflow mappings by sending workflow mapping tasks to a cluster. The second and the third ones are the optimization on the process of searching the minimum bottleneck among all mapping solutions. We use two informed search techniques inspired by depth-first search (DFS) and breadth-first search (BFS). The depth-first informed search speeds up the search process by skipping unpromising combination mapping. The breadth-first informed search uses the greedy approach to process the module with the highest priority in each layer and iteratively search the mapping solution.

Parallelizing the Workflow Mapping Executions

When multiple workflows need to be executed, we can send them to a PC cluster to speed up the mapping calculation using MPI. To further speed up the workflow execution, we also implement OpenMP inside some of the functions. This implementation largely reduces the workflow mapping execution time, especially for large problem cases. The original code was developed in Microsoft Visual Studios environment in C++ language. We adapt the code to work in the Linux environment to use the cluster at MTSU. We use a Single Program Multiple Data (SPMD) approach to implement the algorithm. The workflow mapping tasks are stored in a text file. The root node will read those workflow mapping tasks one by one and pass to all child nodes until all child nodes have a task. Then the root

node takes the following mapping tasks. We repeat this process until all the tasks in the text file are completed.

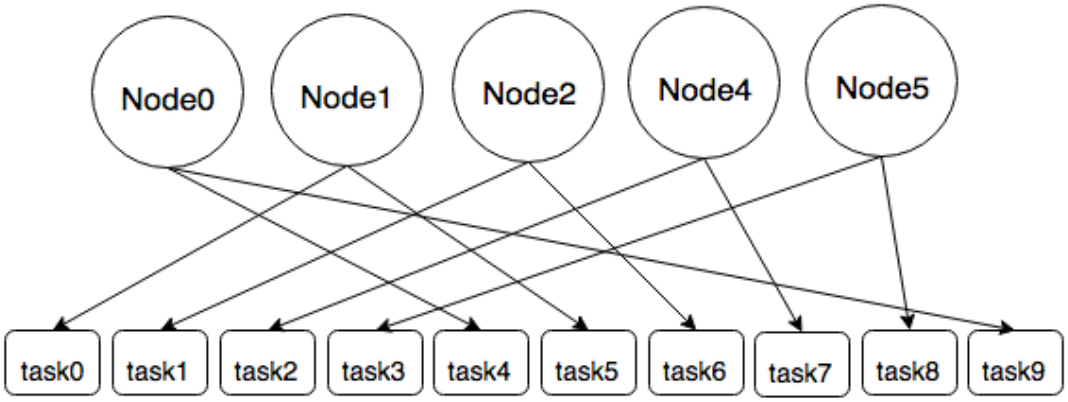

Figure 5. Using MPI to parallel processing multiple workflow mappings

Figure 5 shows an example of using MPI for parallel processing with multiple workflows. The circles represent computer nodes in cluster, and the rectangles represent the mapping tasks. We use 5 nodes to process 10 workflow mapping tasks. In this example, we set node 0 as root node. Node 0 will read workflow mapping tasks file and send those tasks to child nodes. Node 1 will get the first mapping task which is task 0. After all child nodes have one task to process, Node 0 will process task 4. We repeat such process until all mapping tasks are completed. We provide the performance evaluation and comparison in Chapter V. Due to the intricate dependencies among the implementation of the original LDP algorithm, we only implement data parallelism. Thus it is beneficial for the scenarios when multiple workflows need to be executed or multiple datasets of the same workflow need to be processed for streaming applications. We need to explore more possibilities of task parallelism in the LDP mapping algorithm to further improve the runtime performance.

Optimization on Searching Minimum Bottleneck

Searching the bottleneck in the original LDP algorithm has a very high time complexity, which may affect the workflow performance when we have a large computing workflow or a computer network. To solve this problem, we design two solutions and implement them to speed up the process of finding the minimum bottleneck. One solution uses depth-first informed search technique, and the other uses breadth-first informed search technique [11,12,13].

Depth-First Informed Search Solution

When we calculate the bottleneck for one mapping combination, we need to find the largest value among all module execution times and data transfer times for this combination. When a new module is mapped to a node, the module execution time and the data transfer time are calculated and compared with the current minimum bottleneck. The current mapping combination will not be the best solution for all future mappings if the minimum bottleneck is smaller than any of the above two values. We can skip the current mapping combination and move to the next one. Thus, some unnecessary computations can be reduced. If the newly added module execution time and data transfer time are smaller than the current minimum bottleneck, which means this mapping combination is still "promising", we continue to process the rest of unmapped modules and edges.

Figure 6 (a) is an example of using depth-first informed search. In layer 2, we need to map modules w_1, w_2, w_3, and the candidate nodes are v_1, v_2, v_3. Figure 6 (b) is the partial search tree of figure 6 (a). We assume that w_1, w_2 and w_3 have already been sorted based on their computational requirements. In node 4 of the search tree in Figure 6 (b), modules

w_1, w_2 and w_3 are all mapped to node v_1. We set the minimum bottleneck in the current mapping combination as the global minimum bottleneck.

In the following calculation, whenever we add a module to a node as a new combination, we calculate the new module execute time or data transfer time. If the time for the newly added module is less than the global minimum bottleneck, we continue to process the rest of the mapping in the current combination. Otherwise, we stop this combination and do not need to try any other combinations with higher bottlenecks. For example, in Figure 6 (b) in node 7 of the search tree, if we map w_2 to v_2 and get a higher bottleneck than the global bottleneck, we can skip all combinations with w_1 being mapped to v_1 and w_2 being mapped to v_2 in all future calculations. The same situation may happen in nodes 8 or 9 of the search tree.

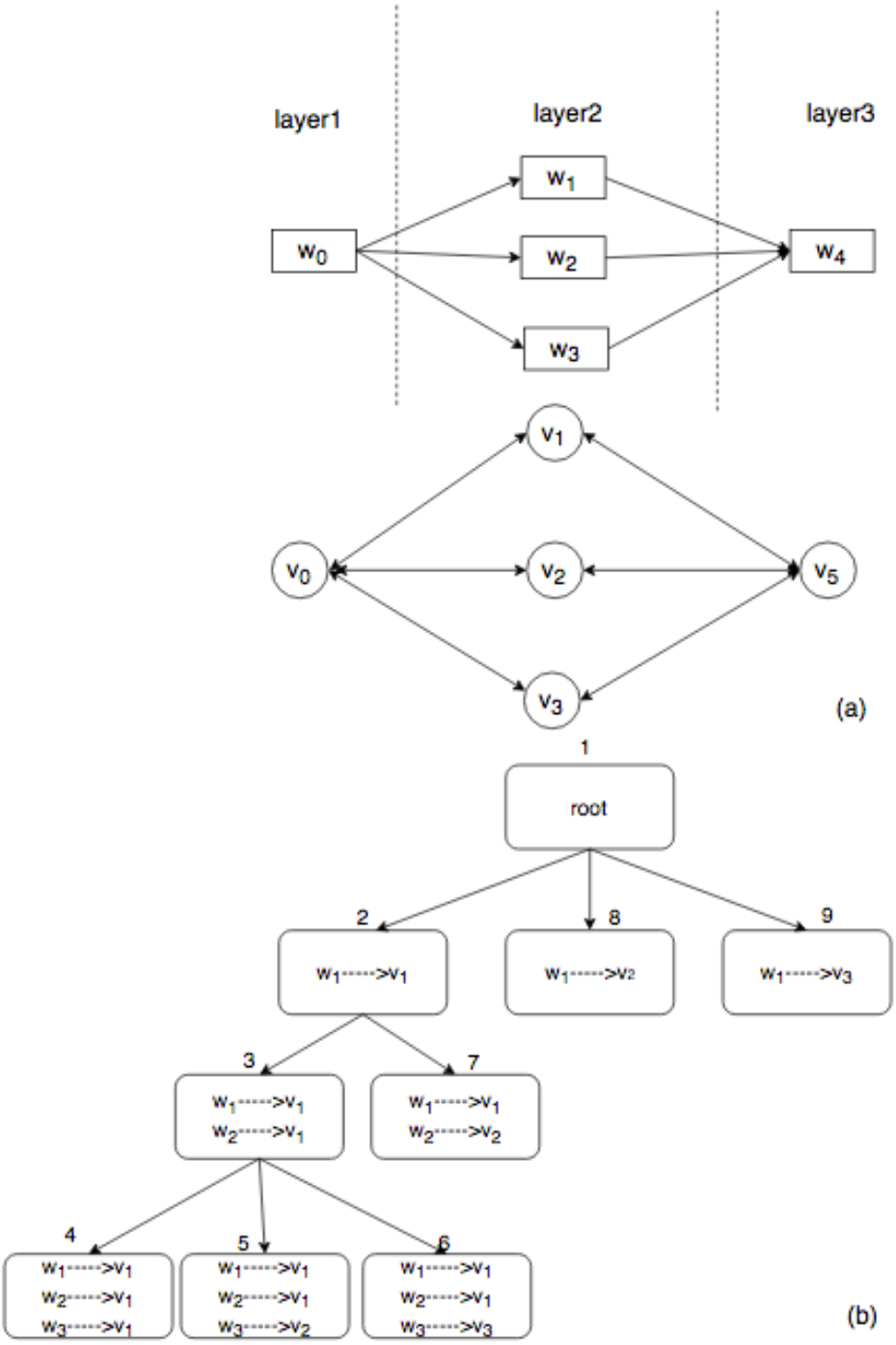

Figure 6. Depth-First informed search solution example

Input: workflow graph $G_w = (V_w, E_w)$, network graph $G_c = (V_c, E_c)$, current layer module sorted list named ModList, available node list for each module named NodeList, search index I, global minimum bottleneck BN_{global}
Output: minimum bottleneck for current layer
1: set global minimum bottle neck to +infinity, searchindex as 0
2: getMiniminBN(G_w, G_c, ModList,NodeList, i, BN_{global}, BN_{current})
3: if i == ModList.size()
4: if ($BN_{\text{current}} < BN_{\text{global}}$)
5: $BN_{\text{global}} = BN_{\text{current}}$
6: return
7: for ith Module
8: for node j in NodeList
9: add node j in mapping
10: update reuseConuter
11: calculate bottleneck for node j as BN_{j}
12: if ($BN_{\text{j}} > BN_{\text{global}}$)
13: remove node j in mapping
14: update reuseConuter
15: continue
16: if ($BN_{\text{j}} > BN_{\text{current}}$)
17: $BN_{\text{current}} = BN_{\text{j}}$
18: getMiniminBN(G_w, G_c, ModList,NodeList, i + 1, BN_{global}, BN_{current})
19: end if
20: remove node j in mapping
21: update reuseConuter
22: end for
23: end for
24: return
25:end function

Figure 7. Pseudocode of informed search solution based on DFS

Figure 7 is the pseudocode for the depth-first informed search algorithm for finding the minimum bottleneck. We go through all the modules, map them to available nodes, and

see if the mapping is promising or not. Note that the reuseCounter is a counter associated with each node in the network to check the number of modules being mapped to the selected node. Every time when we map a new module to a new node, we need to update reuseCounter to reflect the current resource sharing.

If we consider the search process as a search tree in Figure 6 (b), the algorithm will try every possible combination mapping in the worst case. Since there are m^n leaves, the complexity of the worst case is $o(m^n)$. However, in most cases, the search process barely reaches the leaves in the search tree because most search processes are stopped due to non-promising mappings. Therefore, the complexity of the average case could be much smaller than $o(m^n)$.

Breadth-First Informed Search Solution

Although the depth-first informed search solution has reduced the algorithm complexity and runtime, in some mapping cases with a large number of nodes or modules, it may still take too much time. To handle these types of computing workflows, we design and implement another solution by using breadth-first informed search and greedy algorithm. In large cases, normally there would be many more nodes available than modules in one layer. The modules are very likely to be mapped to different nodes unless there are some very powerful nodes which may still have low module execute time or data transfer time even if multiple modules are executed at the same time.

Also, the module with the highest computational requirement is more likely to take more time to execute if the module is not mapped properly, which means the modules with higher computational requirements sometimes need to be mapped first in order to achieve lower overall bottlenecks. Instead of traversing the search tree in depth-first informed search, we

map the modules to the node with the lowest data transfer time or module execution time among the node list based on computational requirements in a descending order. Then we update the mapping result and iterate to the next module.

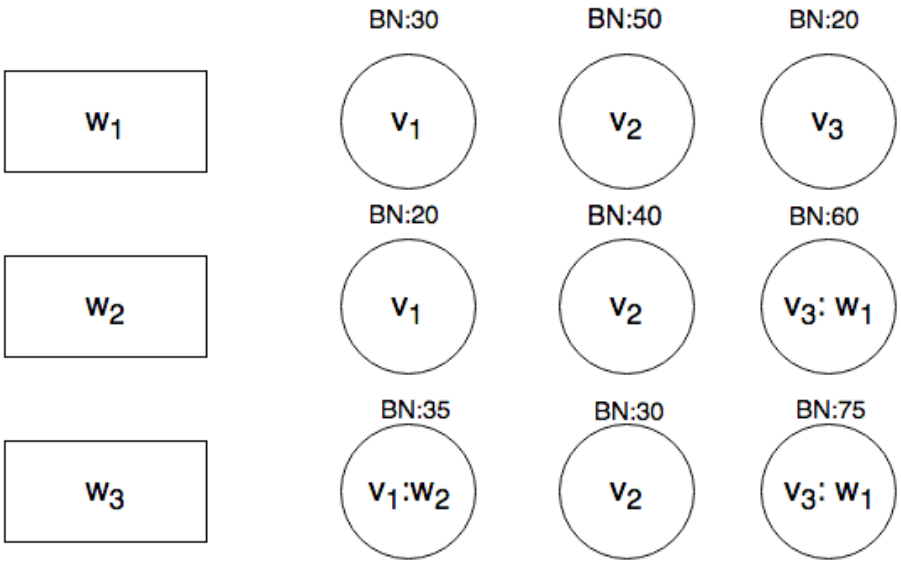

Figure 8. Breadth-First informed search solution example

Figure 8 is the mapping process for Figure 6 (a), in which we assume the modules are sorted as w_1, w_2 and w_3. The BN value on top of each node is the time of mapping the current module to the node. So we only need to go through the node list once and choose the node with the smallest time. Note that when we process w_2, w_1 has already been mapped to v_3. So even if v_3 has some advantages like more powerful processor or shorter data transfer time, the bottleneck time may still increase because multiple modules are mapped to the same node.

```
Input: workflow graph G_w = (V_w, E_w), network graph G_c = (V_c, E_c) current layer
module sorted list named ModList, available node list for each module named
NodeList, search index I, global minimum bottleneck BN_global
Output: minimum bottleneck for current layer
1: set global minimum bottleneck to +infinity,
2: getMiniminBN(G_w, G_c, ModList,NodeList)
3:      for ith Module
4:          for each node j in NodeList
5:              add node j in mapping
6:              update reuseConuter
7:              calculate BN for node j as BN_j
8:              if (BN_j < BN_global)
9:                  BN_global = BN_j
10:                 if (BN_j > BN_current)
11:                     BN_current = BN_j
12:                 end if
13:             end if
14:             remove node j
15:             update reuseConuter
16:         end for
17:         set the mapping for node j
18:     end for
19:     return
20: end function
```

Figure 9. Pseudocode of informed search solution based on BFS

Figure 9 is the pseudocode of breadth-first informed search for finding the minimum bottleneck. We go through the module list and try each available node to find a minimum bottleneck. The purpose of reuseCounter is the same as the one used in depth-first informed search. Note that in this solution we do not go through every combination, so that the final result may be trapped in the locally optimal choice. However, this solution does reduce the time complexity dramatically as it only goes through all the modules and nodes once, resulting in a lower time complexity o(mn).

CHAPTER V

PERFORMANCE EVALUATION

We implement the depth-first informed search solution, the breadth-first informed search solution and MPI parallel solution in C++ language[1]. The input of each algorithm is the workflow graph and network graph, and the output is the maximum frame rate (MFR). Both informed search solutions consider the optimization of the current layer bottleneck. All experiments were performed on the MTSU ranger cluster. Each node has Intel Xeon CPU E5-2640 of 2.60GHz. For comparison, we first provide the MFR comparison of the original and the improved LDP algorithms by varying the problem sizes from small scales to large ones. Theoretically, the depth-first informed search solution should traverse all promising combination mapping to get the global optimal MFR result, and the breadth-first informed search solution uses greedy approach and may be trapped in local optimal choice. We also need to test the MFR performance of two informed search solutions. We provide the MFR comparison of the original and two informed search solutions incorporated with current layer bottleneck optimization. Finally, for the algorithm runtime comparison, we compare the runtime and MFR performance among original algorithm, depth-first informed search solution, breath-first informed search solution and MPI parallel solution.

[1] You can contact kt3j@mtmail.mtsu.edu for more implementation details

Table 2. Problem case information

Problem case index	**Definitions** m, $\lvert E_w \rvert$,n,$\lvert E_c \rvert$
1	4,6,6,35
2	6,10,10,96
3	10,18,15,222
4	13,24,20,396
5	15,30,25,622
6	19,36,28,781
7	22,44,31,927
8	26,50,35,1215
9	30,62,40,1598
10	35,70,45,2008
11	38,73,47,2200
12	40,78,50,2478
13	45,96,60,3580
14	50,102,65,4220
15	55,124,70,4890

Table 2 is the information of different problem cases we use to evaluate the proposed algorithm, where m is the number of modules, $|E_w|$ is the number of edges between a pair of adjacent modules, n is the number of nodes, and $|E_c|$ is the network link between two nodes. The sizes of the problem cases increase with the problem indices. The larger problem cases are more, but not necessarily, likely to have a longer execution time. Since MFR is not particularly related to the problem size, these results lack an obvious increasing or decreasing trend in response to the increasing problem sizes.

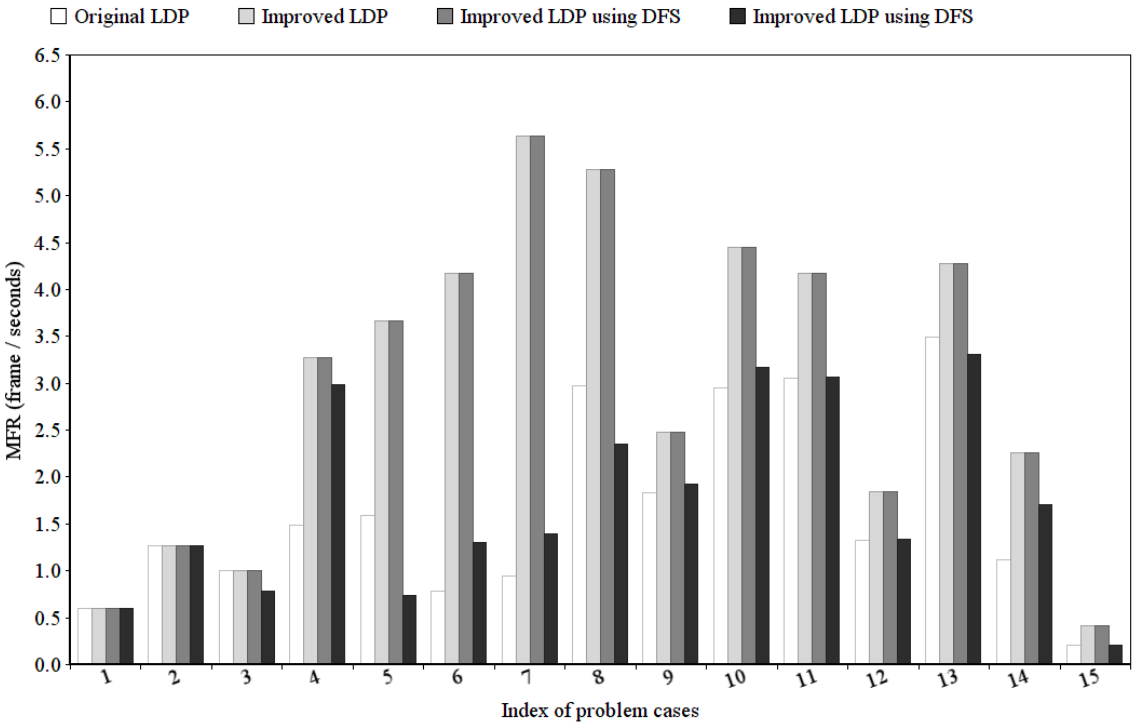

Figure 10. MFR comparison among four algorithms

Figure 10 shows the MFR comparison among the four algorithms. From the comparison between original LDP and improved LDP, we observe that the MFR increases after we optimize the bottleneck mapping in the current layer. Note that in the small cases like cases 1, 2 and 3, the MFR value stays the same as they have less mapping options. On average, the MFR increases by 81.3%.

Both breadth-first informed search and depth-first informed search have already incorporated with the current layer bottleneck optimization. From the figure, we observe that the DFS has the best MFR performance that is the same as the improved LDP algorithm, which means the depth-first informed search traverses all promising combination mappings and gets the global optimal MFR result. Also, the figure shows the breadth-first informed

search solution has lower MFR result than improved LDP algorithm, and hence this solution is trapped in local optimal choice. However the MFR of breadth-first informed search is still 7% higher than the original LDP algorithm. The algorithm is more focused on algorithm runtime optimization and we will show the runtime comparison in the next figure.

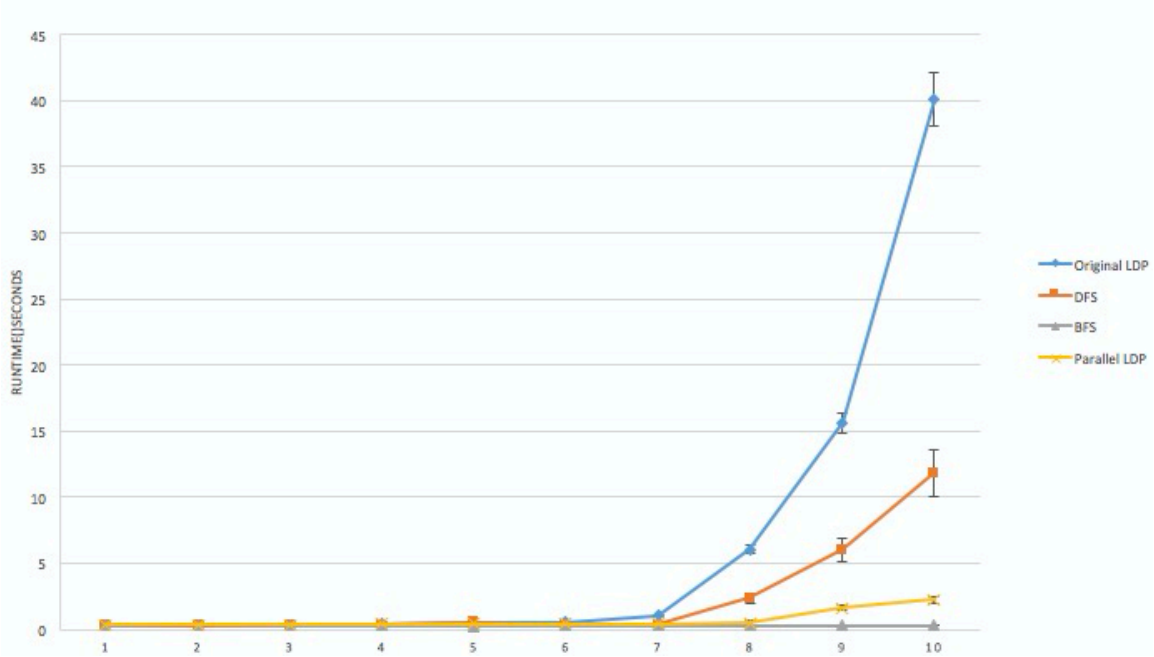

Figure11. Runtime comparison among four algorithms

Figure 11 shows the runtime comparison among the four algorithms. We run each problem case ten times for each algorithm and use the average as the actual vaule in the figure. With the increase of the problem sizes, the runtime of the original LDP algorithm increases dramatically. However, the runtime of the depth-first informed search increases slowly, while the parallel MPI and the breath-first informed search only take a few seconds. For the largest case in Figure 11, the original LDP algorithm takes 40 seconds to complete the

workflow mapping execution while the DFS costs less than 12 seconds and BFS only takes less than 1 second. The parallel LDP with MPI uses 10 nodes to run 10 tasks in the cluster, and each node gets one task each time. We calcualte the average runtimes and the standard error per task of parallel LDP algorithm and plot them in Figure 11. Due to the overhead of MPI message passing, the runtime parallel LDP algorithm is slightly larger than the original algorithm for small problems. However when we increase the problem size, the parallel LDP takes much less time than the original algorithm. It is also better than the depth-first informed search but not as good as the breath-first informed search. Note that although parallel LDP achieves better runtime performance, it uses more computing resources.

CHAPTER VI

CONCLUSION

In this thesis, we reviewed the original LDP algorithm and its cost model, and found that the original LDP algorithm is not able to maintain the minimum bottleneck in the current layer among all mapping combinations. We modified the mapping process of finding the bottleneck in order to obtain a better global minimum bottleneck by checking the minimum bottleneck in each layer. By implementing this optimization, the improved LDP algorithm achieves a better maximum frame rate.

Moreover, since the original algorithm has high search complexity in the process of finding the minimum bottleneck, especially when the sizes of the computing workflow and computer network increase. We proposed two new informed search solutions to reduce the search time. The depth-first informed search decreases runtime by skipping unnecessary calculations for unpromising mappings. It can effectively decrease the search time while still achieving better MFR results. The breadth-first informed search can speed up the search time by using a greedy approach to search the mapping which most likely has the minimum bottleneck with some sacrifice in MFR performance. However, BFS still results in a better MFR than the original algorithm. We also implemented the parallel LDP algorithm using MPI for sending multiple mapping tasks across a computing cluster. This approach can reduce runtime but uses more computing resources.

In the current cost models, we used a normalized quantity to represent the processing power and bandwidth for simplicity. However, a single constant is not always sufficient to describe node computing and link transfer capabilities. It is of our future interest to investigate more sophisticated cost models to characterize real-time node and link

behaviors in dynamic network environments. Since the breadth-first informed search solution has better runtime efficiency compared to the depth-first informed search solution, we will try to further improve the performance of the breadth-first informed search solution algorithm without sacrificing its runtime efficiency. We also plan to test the proposed mapping solutions and evaluate their performance in real large-scale networks.

BIBLIOGRAPHY

[1].Sriprayoonsakul, S., Uthayopas, P., Zheng, C., Lee, J., Livny, M., & Frey, J. (2008). Interfacing SCMSWeb with Condor-G - A joint PRAGMA-Condor effort. *Proceedings - 4th IEEE International Conference on eScience, eScience 2008*, 570–575.

[2].Gu, Y., & Wu, C. Q. (2016). Performance Analysis and Optimization of Distributed Workflows in Heterogeneous Network Environments. *IEEE Trans*, *65*(4), 1266–1282.

[3].Sarwar, G., Boreli, R., Lochin, E., & Mifdaoui, A. (2012). Performance evaluation of multipath transport protocol in heterogeneous network environments. *2012 International Symposium on Communications and Information Technologies (ISCIT)*, 985–990. Retrieved from

[4].Lin, C., Lu, S., Fei, X., Pai, D., & Hua, J. (2009). A task abstraction and mapping approach to the shimming problem in scientific workflows. *SCC 2009 - 2009 IEEE International Conference on Services Computing*, 284–291. http://doi.org/10.1109/SCC.2009.77

[5].Wu, C. Q., & Cao, H. (2016). Optimizing the performance of big data workflows in multi-cloud environments under budget constraint. *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016*, 138–145. http://doi.org/10.1109/SCC.2016.25

[6].Li, Z., Ge, J., Hu, H., Song, W., Hu, H., & Luo, B. (2015). Cost and energy aware scheduling algorithm for scientific workflows with deadline constraint in clouds. *IEEE Transactions on Services Computing*, *PP*(99), 1–15. http://doi.org/10.1109/TSC.2015.2466545

[7].Thomas Sterling, "Condor: A Distributed Job Scheduler," in *Beowulf Cluster Computing with Windows* , 1, MIT Press, 2001, pp.307-343

[8].Dongyu Feng, Ligu Zhu, & Lei Zhang. (2016). Review of hadoop performance optimization. *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, 65–68.

[9].Liu, Y., Guo, S., Hu, S., Rabl, T., Jacobsen, H.-A., Li, J., & Wang, J. (2016). Performance Evaluation and Optimization of Multi-dimensional Indexes in Hive. *IEEE Transactions on Services Computing*, *14*(8), 1–1.

[10].Li, Y., Zhang, J., & Liu, Q. (2016). Cluster resource adjustment based on an improved artificial fish swarm algorithm in Mesos, 1843–1847.

[11].R. Tarjan, "Depth-first search and linear graph algorithms," *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, East Lansing, MI, USA, 1971, pp. 114-121.

[12].H. Milišić, D. Ahmić, H. Sinanović, E. Sarić, A. Asotić and A. Huseinović, "Parallelization challenges of BFS traversal on dense graphs using the CUDA platform," *2016 XI International Symposium on Telecommunications (BIHTEL)*, Sarajevo, 2016, pp. 1-5.

[13].Tovey, C., & Koenig, S. (2003). Improved analysis of greedy mapping. *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, *4*(October), 3251–3257.

[14].W. Xiang, "Analysis of the Time Complexity of Quick Sort Algorithm," (2011*)* International Conference on Information Management, Innovation Management and Industrial Engineering, Shenzhen, pp. 408-410.

[15].C. E. Radke, "Merge-sort analysis by matrix techniques," (1966) in *IBM Systems Journal*, vol. 5, no. 4, pp. 226-247.

[16].U. Tamm, "Communication complexity of functions related to set intersection," (2016) *2016 Information Theory and Applications Workshop (ITA)*, La Jolla, CA , pp. 1-4.

[17].B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran, "Streamline: scheduling streaming applications in a wide area environment,"(2007) Multimedia Systems, vol. 13, no. 1, pp. 69–85.

[18].A. Sekhar, B. Manoj, and C. Murthy, "A state-space search approach for optimizing reliability and cost of execution in dis- tributed sensor networks,"(2005) in Proc. Int. Workshop Distrib. Comput. pp. 63–74.