Computation and Communication Schedule Optimization for Jobs with Shared Data

En-Jan Chou Pangfeng Liu
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.
{*r94149,pangfeng*}*@csie.ntu.edu.tw*

Jan-Jan Wu
Institute of Information Science
Academia Sinica
Taipei, Taiwan, R.O.C.
wuj@iis.sinica.edu.tw

Abstract

Almost every computation job requires input data in order to find the solution, and the computation cannot proceed without the required data becoming available. As a result a proper interleaving of data transfer and job execution has a significant impact on the overall efficiency. In this paper we analyze the computational complexity of the shared data job scheduling problem, with and without consideration of storage capacity constraint. We show that if there is an upper bound on the server capacity, the problem is NP-complete, even when each job depends on at most three data. On the other hand, if there is no upper bound on the server capacity, we show that there exists an efficient algorithm that gives optimal job schedule when each job depends on at most two data. We also give an efficient heuristic algorithm that gives good schedule for cases where there is no limit on the number of data a job may access.

1. Introduction

As cluster and gird systems become increasingly popular, we are able to dispatch computational intensive jobs to remote servers so that the latency of obtaining the final solution is dramatically reduced. By dispatching jobs to other processing units, possibly on remote locations, we are able to utilize all the available computing power via network connectivity.

Almost every computation job in the cluster or grid systems requires input data in order to find the solution. For example, *AppLeS Parameter Sweep Template* [2] provides a framework for fast deployment and execution of grid applications. The execution model consists of independent jobs, each job requires a different set of data, and jobs may share these data. The computation cannot proceed without the required data. Other applications like *Basic Local Alignment Search Tool* [1] and many data-intensive computations, also demands efficient data transfer from location to location. The efficiency of data transfer has a significant impact on the overall efficiency.

Modern computer systems are able to overlap computation and communication for latency hiding. For example, if a job a has uploaded all of its data and starts its computation, a waiting job b can starts uploading its data so that when job a finishes, job b can proceed to its computation as early as possible, As a result a efficient scheduling algorithm must consider both computation and communication of multiple jobs, and try to overlap them as often as possible, in order to improve performance.

There are several issues in the efficiency of executing jobs that require input data. The first issue is to schedule the execution order of jobs so that the data transfer and CPU computation from consecutive jobs overlap better. The optimization of this execution order is complicated by the fact that different jobs may share data. Here we assume that there is a storage on a server, so that the data used by the previous job may be reused by the next job. That is, a datum may be uploaded into a server by one job, and reused by another job if these two jobs share the datum. As a result it is a difficult problem to find the best job j to run after the current job since we must consider not only the data required by j but also the data that have already been uploaded by previous jobs.

The second issue in the execution efficiency of multiple jobs is that of data capacity constraints. The data required by jobs may be huge, so the storage capacity should be also taken into consideration. We may

978-1-4244-1890-9/07/$25.00

assume that there exists an upper bound on the number of data that can be stored in the server, and if the next incoming job needs to upload an amount of new data that exceeds the available free space left on the server, we need to delete old data to make room for new ones. This significantly complicates the scheduling.

We focus on the problem of scheduling jobs that may share data on single-processor environment. When there are multiple heterogeneous servers in the system, Giersch et al. [4] showed that the scheduling problem is NP-complete, even if the execution time of all the jobs are the same. It is well known that if jobs do not share data and there are multiple heterogeneous servers, the problem remains NP-complete. Maheswaran et al. [6] proposed several heuristic algorithms for this scheduling problem. When there is only one server Giersch et al. [4] also showed that the scheduling problem is NP-complete, Johnson [5] gave an optimal solution in the special case where jobs do not share data.

In this paper we analyze the computational complexity of the shared data job scheduling problem, with and without consideration of storage capacity constraint. We show that if there is an upper bound on the server capacity, the problem is NP-complete, even when each job depends on at most *three* data. On the other hand, if there is no upper bound on the number of data on the server, we show that there exists an efficient algorithm that gives optimal job schedule when each job depends on at most *two* data. We also give an efficient heuristic algorithm that gives good schedule for cases where there is no limit on the number of data a job may require. Experimental results indicate that this heuristic algorithm performs very well.

The rest of the paper is organized as follow. Section 2 describes the system model. Section 3 describes the algorithms for the case where storage capacity is unlimited, and prove that the algorithm finds optimal solution. Section 4 discusses the computational complexity when storage capacity is limited. Section 5 proposes an efficient heuristic algorithm that schedules when storage capacity is unlimited and there is no limit on the number of data a job may require. Section 6 describes experimental results of our scheduling algorithm. Section 7 is the conclusion and a brief discussion on the future work.

2. System Model

This section describes the system model of the multiple shared data job scheduling problem. The system model consists of a computation server, a data warehouse, M data that are in the data warehouse, N jobs that will run on the server, and a binary relation describes the dependency between jobs and data.

We describe two system models — the *unlimited capacity model* and the *limited capacity model*. The difference between these two models is that in the unlimited capacity model, we can uploaded unlimited number of data to the server so that jobs can proceed. However, in the limited capacity model, the number of data the server can keep in its local storage is limited.

2.1. Unlimited Capacity Model

We consider N independent jobs $J = \{j_1, j_2, \ldots, j_N\}$, M different data $D = \{d_1, d_2, \ldots, d_M\}$, and a binary relation $R = \{(j,d) \mid j \in J, d \in D\}$ so that (j,d) is in R if and only if job j requires datum d to execute.

A job j requires a set of data $D_j = \{d \mid d \in D, (j,d) \in R\}$. We must upload all the data in D_j from the data warehouse to the server before we can run job j on the server.

In our models we can perform computation on the server and data transferring into the server simultaneously. We denote the time to upload a datum d from the warehouse to the server by $|d|$, and the time to execute a job j on the server by $|j|$. We further assume that all $|j|$ are integer, and all $|d|$ are 1.

2.2. Limited Capacity Model

The limited capacity model is similar to the unlimited capacity model, except that the server has a local storage capacity limit. If the amount of data has reached the capacity limit, the server has to remove some data in order to accommodate the incoming ones. Because the job will access data while running, the data required by the running job has not to be removed.

2.3. Problem

Our goal is to schedule jobs on server to minimize the *makespan*, which is the amount of time from uploading the first datum to finishing the last job. To be more specific, we would like to find an *uploading sequence* for the data and an *execution sequence* for the jobs, as well as the exact time at which to upload data and to execute jobs.

3. Algorithms for Unlimited Capacity Model

We now describe our results for the unlimited capacity model. It is very difficult to find the optimal schedule for the shared-data job scheduling problem.

Giersch et al. showed that when the number of data a job may depend on is unbounded, the decision problem of whether there exists a schedule within a given time bound is NP-Complete, even when each job requires at most *three* data [4]. In contrast we show that there exists an algorithm that finds the optimal schedule when every job depends on at most *two* data. In addition, we will describe a heuristic algorithm in Section 5 for jobs that depend on more than two data, with experimentally verified good performance.

We first define our terminology. We denote the data uploading sequence by $\mathscr{U} = < u_1, \ldots, u_M >$, which is a permutation of the integers from 1 to M. Similarly we denote the job execution order by $\mathscr{E} = < e_1, e_2, \ldots, e_N >$, which is a permutation of integers from 1 to N. These two sequences describe the order in how we upload data and run jobs. For example, a data uploading sequence $\mathscr{U} = < 3,1,2 >$ means we upload data in the order of d_3, d_1, d_2.

It is easy to see that there is no potential saving in time when we delay uploading data, so we may assume that the data in an uploading sequence are sent to the server without any delay. As a result, given a data uploading sequence $\mathscr{U}$, we start uploading every datum in $\mathscr{U}$ as soon as we finish uploading the previous datum in the sequence. The starting time to upload a datum d_{u_i} (denoted by $T(d_{u_i})$) can now be formulated as follows. Note that we use the $\mathscr{U}$ subscript to emphasize that the uploading sequence is $\mathscr{U}$.

$$T_{\mathscr{U}}(d_{u_1}) = 0 \quad (1)$$

$$T_{\mathscr{U}}(d_{u_i}) = T_{\mathscr{U}}(d_{u_{i-1}}) + |d_{u_{i-1}}| \quad (2)$$

A job j must upload all of its data before its execution, so we define a *data ready time* of job j (denoted by $R(j)$) to be the time when all of its required data (D_j) have been uploaded to the server. Now given a data uploading sequence $\mathscr{U}$ and the uploading time function $T_{\mathscr{U}}$, we can compute the data ready time of a job j by Equation 3.

$$R(j) = \max_{d \in D_j} (T_{\mathscr{U}}(d) + |d|) \quad (3)$$

We now describe an procedure that when given a data uploading sequence $\mathscr{U}$, determines an optimal job execution sequence $\mathscr{E}$ that minimizes the makespan. We sort the jobs in J into a sequence $\mathscr{E}$ according to their data ready time, so that the job with an earlier data ready time appears earlier in the sequence $\mathscr{E}$. If two jobs have the same data ready time, the job with smaller index runs first. It is easy to see that for this sequence $\mathscr{E}$, the following equation holds.

$$R(j_{e_a}) \leq R(j_{e_b}), \quad \forall\ 1 \leq a < b \leq N. \quad (4)$$

Obviously we cannot run a job j_{e_i} before the previous job $j_{e_{i-1}}$ finishes, or before all of its required data $D_{j_{e_i}}$ are uploaded to the server, so after given the uploading sequence $\mathscr{U}$ and the execution sequence $\mathscr{E}$, we can derive the time to start running a job j (denoted by $T_{\mathscr{E}}(j)$) as follows. Note that we use the $\mathscr{E}$ subscript to emphasize that this is for the execution sequence $\mathscr{E}$.

$$T_{\mathscr{E}}(j_{e_1}) = R(j_{e_1}) \quad (5)$$

$$T_{\mathscr{E}}(j_{e_i}) = \max\{R(j_{e_i}), T_{\mathscr{E}}(j_{e_{i-1}}) + |j_{e_{i-1}}|\} \quad (6)$$

Theorem 1. *Given a data uploading sequence $\mathscr{U}$, the job execution sequence $\mathscr{E}$ described by Equation 4, 5 and 6 minimizes the makespan.*

Proof. We prove the theorem by contradiction and assume that the execution sequence $\mathscr{E}$ from Equation 4, 5 and 6 does not minimizes the makespan. Instead a different execution sequence $\mathscr{E}' = < e'_1, \ldots, e'_N >$ and a $T'_{\mathscr{E}'}$ minimize the makespan.

Because $\mathscr{E}'$ is different from $\mathscr{E}$, we locate a minimum index a such that $a < b$ but e'_a appears later than e'_b in $\mathscr{E}'$. We now delay the execution of jobs $j_{e'_i}$, for all $i, a \leq i < b$, and run job $j_{e'_b}$ at the time where we ran job j'_a. Since the data ready time of j'_b is earlier than j'_a, this adjustment is always possible. In addition, those jobs $j_{e'_i}$ ($a \leq i < b-1$) will be delay by at most $|j_{e'_b}|$, which is exactly the empty slot left behind by j'_b, the makespan will not be affected. We repeat this process until $\mathscr{E}'$ becomes the same as $\mathscr{E}$. The theorem follows. □

3.1. Optimal Algorithm for Jobs with Two Data

Given a data set D, a jobs set J, a dependency relation R in which each job depends on at most two data. We describe an scheduling algorithm that gives optimal makespan.

We represent D, J, and R by a bipartite graph B. Let $C = \{c_1, c_2, \ldots, c_K\}$ be the set of connected components in B. Jobs in the same component may share data, and a job requires at most two data from the same component.

The algorithm consists of two steps: In the first step we schedule connected components in the bipartite graph B one component at a time, and determine the data uploading and job execution sequence for each component. In the second step we determine an order among the components, by which all the data uploading and job execution for one component are performed before those of the next component.

Lemma 1. *There exists an optimal uploading sequence $\mathscr{U}$ in which all the data uploading from the same connected component are adjacent in $\mathscr{U}$.*

Proof. Suppose there exists an optimal uploading sequence $\mathscr{U}$ in which *not* all the data uploading from the

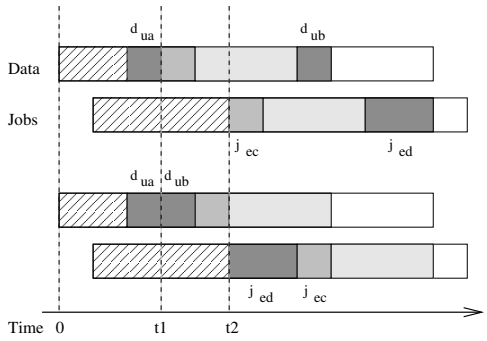

Figure 1. A timing diagram for Lemma 1.

same connected component are adjacent in $\mathscr{U}$. We denote the job execution sequence and exact timing to upload data and execute job, be $\mathscr{U}$, $\mathscr{E}$, $T_{\mathscr{U}}$ and $T_{\mathscr{E}}$.

Because not all the data uploading from the same connected component are adjacent in $\mathscr{U}$, we can find a job j_{e_d} whose data (d_{u_a} and d_{u_b}) are separated by a datum not in the same connected component in $\mathscr{U}$. Now consider jobs require only data $d_{u_1},\ldots,d_{u_a}$, which are those jobs with data ready time no later t_1 in Figure 1. We also assume that these jobs will finishes at at t_2, and the next job to run at t_2 is j_{e_c}. As a result job j_{e_c} require some data other than $d_{u_1},\ldots,d_{u_a}$ and cannot start before time t_1+1.

Now consider the data that are between d_{u_a} and d_{u_b} in $\mathscr{U}$, namely $d_{u_{a+1}},\ldots,d_{u_{b-1}}$, and the jobs between j_{e_c} and $j_{e_{d-1}}$. We now switch the order of uploading d_{u_b} and $d_{u_{a+1}},\ldots,d_{u_{b-1}}$, and the order of running j_{e_d} and $j_{e_c},\ldots,j_{e_{d-1}}$, please refer to the bottom half of Figure 1 for an illustration. Since every datum has an uploading time 1, it is always possible to switch the order, therefore we only need to consider the feasibility of switching jobs.

Because $|d_{u_b}|=1$, the data ready time of j_{e_d} will change to t_1+1 after we make the switch. That means job j_{e_d} can start running at $\max\{t_2,t_1+1\}$. We also know that the starting time of j_{e_c} is at least t_1+1 since it requires data other than $d_{u_1},\ldots,d_{u_a}$, and its starting time is also at least t_2 since the jobs before it ends at t_2. Therefore it is always possible to run j_{e_d} at the time where we ran j_{e_c} before the switch.

Jobs $j_{e_c},\ldots,j_{e_{d-1}}$ will be delayed by $|j_{e_d}|$ after the switch. However, the corresponding data $d_{u_{a+1}},\ldots,d_{u_{b-1}}$ is delayed by exactly one time unit after the switch. Since $|j_{e_d}|\geq 1$, the new schedule is feasible.

After considering the effects of the switch on both data uploading and job execution, we conclude that the makespan will not be increased. We repeat this procedure until we cannot find any jobs with its two data separated by a datum from another connected component in the uploading sequence $\mathscr{U}$. The theorem follows. □

From Lemma 1 we conclude that there exists an optimal schedule in which the data from every connected component are grouped together in $\mathscr{U}$. Note that this property implies that while searching for the optimal uploading sequence $\mathscr{U}$, we only need to consider all the $K!$ possibilities of enumerating the connected components, where K is the number of connected components.

Now we describe an algorithm that finds an optimal schedule for a connected component c. If there are jobs depending on only one datum, then we can put the datum in the front of the uploading sequence. If there is no job depending on only one datum, then we randomly choose a job and put both of its data in the front of the uploading sequence. After deciding which data have to be uploaded first, we choose datum d so that there are jobs depending on both d and another datum already in the uploading sequence, then we append d to the uploading sequence.

Lemma 2. *We can find the optimal scheduling for all the connected components.*

Proof. Because every job depends on at most two data and these jobs are in the same connected component, once a job starts running, we only need to upload another datum so that a new job could start running. Therefore once jobs start running, they finishes at a rate of one job per time unit. Let J_c is the set of jobs in connected component c, the makespan is therefore $\sum_{j\in J_c}|j|+1$ if there are jobs that require only one data. Otherwise the makespan is $\sum_{j\in J_c}|j|+2$, which are the best possible. □

Lemma 3. *There exists an optimal uploading sequence $\mathscr{U}$ and its corresponding execution sequence $\mathscr{E}$, such that $\mathscr{U}_i$ and $\mathscr{E}_i$ are the subsequence of $\mathscr{U}$ and $\mathscr{E}$ from the i-th connected component c_i, and $\mathscr{U}_i$ and $\mathscr{E}_i$ are optimal for c_i.*

Proof. We assume that there exists an integer i such that $\mathscr{U}_i$ and $\mathscr{E}_i$ are not optimal for c_i. Instead the optimal uploading sequence the execution sequence are $\mathscr{U}_i'$ and $\mathscr{E}_i'$ respectively. Note that in the construction of optimal solution for individual connected components, the only difference is the amount of time overlapping between the data uploading and job execution. An optimal solution means it has the longest time for this overlap.

Now we consider a new schedule, which consists of the uploading sequence $\mathscr{U}_1,\ldots,\mathscr{U}_{i-1},\mathscr{U}_i',\mathscr{U}_{i+1},\ldots,\mathscr{U}_K$, and execution sequence $\mathscr{E}_1,\ldots,\mathscr{E}_{i-1},\mathscr{E}_i',\mathscr{E}_{i+1},\ldots,\mathscr{E}_K$, where K is the number of connected components. This makespan of

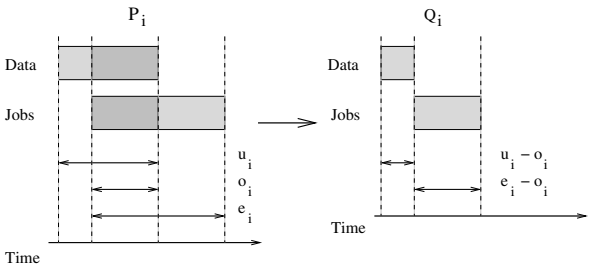

Figure 2. To remove the overlapping time of data uploading and job execution.

this new sequence will not be larger than $\mathscr{U}$ and $\mathscr{E}$, since $\mathscr{U}_i'$ and $\mathscr{E}_i'$ has a longer time overlapping than $\mathscr{U}_i$ and $\mathscr{E}_i$. We can repeat this step until we have an optimal sequence that consists of optimal solutions for individual connected components. The lemma follows. □

Now we describe a two-machine flowshop problem, which will serve as a building block for our scheduling problem. The are n jobs that must go through the first and the second machine, and at most one job can be on a machine. Let A_i and B_i be the time needed by the i-th job on the first machine and the second machine respectively. The goal is to find the optimal order of jobs to minimize the makespan.

Theorem 2. *[5] A schedule $\mathscr{S}$ gives the minimum execution time for the two-machine flowshop problem by first process the jobs with $A_i \leq B_i$ in non-decreasing A_i order, and then process the remaining jobs in non-increasing B_i order.*

We now describe an algorithm that finds an optimal schedule when the optimal schedule of each connected component is computed. We use a set P to represent the optimal schedules from connected components. Let $P_i = (u_i, e_i, o_i)$ be the scheduling result of the i-th connected component, where u_i is the total data uploading time, e_i is the total job execution time, and o_i is the length of the overlapping time of data uploading and job execution.

We can construct a new set of connected components by eliminating the overlap time of each connected component in P_i, as in Figure 2. The resulting schedule is $Q_i = (u_i - o_i, e_i - o_i)$. We then apply the two-machine flowshop algorithm [5] in Theorem 2 to find an optimal components schedule sequence $\mathscr{S}$. This is a sequence that determine the order among the optimal schedules from connected components.

We apply the sequence $\mathscr{S}$ on Q and the makespan is T_1. Similarly we apply the same $\mathscr{S}$ on P and the total execution time is T_2. Now we insert the overlapping part into Q and transform the schedule back to P. The relative position of data and jobs will not be affected since the data ready time and job finish time of the previous job are both delayed by the same amount of time, therefore $T_2 = T_1 + \sum o_i$.

Lemma 4. *Given a set of connected components C, and optimal schedules of each connected components, we can determine a sequence of connected components of C according to Theorem 2, which minimize the makespan.*

Proof. We prove the lemma by contradiction. Now suppose we have a schedule $\mathscr{S}'$ that has a shorter makespan $T_3 < T_2$. Without lose of generality we assume that $\mathscr{S}'$ is still a schedule produced by permuting the optimal schedules from individual components. Now we apply this schedule $\mathscr{S}'$ to a new problem instance, with the overlapping parts removed, much like we did to P. The makespan will decrease by $\sum o_i$. That implies we will have a better schedule for the two-machine flowshop problem, since $T_3 - \sum o_i$ is smaller than $T_1 = T_2 - \sum o_i$. This contradicts to the fact that T_1 is optimal. The lemma follows. □

Theorem 3. *There exists an algorithm that finds an optimal solution for unlimited capacity model when each job requires at most two data.*

From the previous construction it is easy to see that the final schedule is optimal.

4. Limited Capacity Model

We show that for the limited capacity model when we are given a data set D, a jobs set J, the dependency relation R, a storage capacity C, and a time bound B, the decision problem of determining that whether all job can finish in time B is NP-Complete, even when every job depends on at most *three* data. The NP-completeness proof is based on the fact that finding a Hamiltonian path in a graph in which every node has degree at most three, remains NP-complete [3].

Theorem 4. *[3] Given an undirected graph G where the degree of each node is at most three, the decision problem to determine if there is a Hamiltonian path in G is NP-Complete.*

Now we describe the construction. We are given an undirected graph $G = (V, E)$ in which the degree of every node is at most three, and we would like to know if there is a Hamiltonian path in G. We first modify the graph G to ensure that every node has degree of exactly three. If a node v has a degree of less than three, we add dummy nodes (those solid node in Figure 3) and

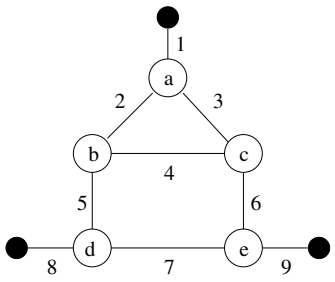

Figure 3. A graph for which we would like to find a Hamiltonian path, with dummy nodes added.

dummy edges (those edges adjacent to dummy nodes in Figure 3) until the degree of each node in G becomes exactly three. We use G' to denote the resulting graph.

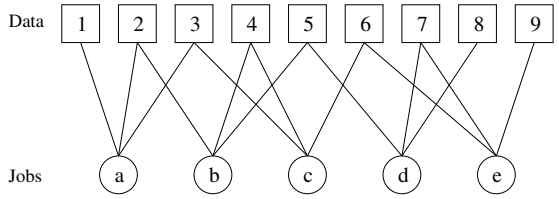

Figure 4. The bipartite of data and jobs with dependency relation from the graph in Figure 3.

We now construct a problem instance of our scheduling problem, based on the graph G that we would like to find a Hamiltonian path. Every datum in D corresponds to an edge in G', and every job in J corresponds to a node in G, and execution time of all the job are 1. A job j requires a datum d if and only if the corresponding node of j is adjacent to the corresponding edge of datum d. Please refer to Figure 4 for an illustration of the scheduling problem instance from the previous Hamiltonian path problem instance in Figure 3. Finally we set the storage capacity C to 3, and time bound B to $3|J|+1$.

It is important that by our transformation each job require exact three data, two jobs may share at most one datum and it happens when there exist an edge between the two corresponding non-dummy nodes.

Theorem 5. *For the limited capacity model when we are given a data set D, a job set J, the dependency relation R, a storage capacity C, and a time bound B, the decision problem of determining that whether all job can finish in time B is NP-Complete, even when every job depends on at most three data.*

Proof. We prove the theorem by showing that given a Hamiltonian path problem instance $G=(V,E)$, G has a Hamiltonian path if and only if we can schedule all jobs in J, with data in D, dependency in R, capacity $C=3$, in $B=3|J|+1$ time units.

By our construction each job requires exact three data and the server can hold at most three data. However, none of two jobs share more than one datum. Every time we switch from a job to another, we need to upload *two* data if the incoming job share a datum with the finishing job, otherwise we have to upload *three* data for the incoming job. For example, in Figure 5 we illustrate a case where the transitions require only two data.

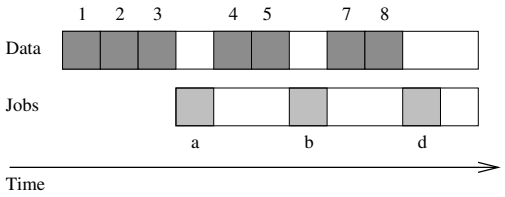

Figure 5. The timing diagram of the scheduling for the problem instance in Figure 4.

Now we analyze the time bound. Because we have to upload three data before the first job can run, the first job requires at least 3 time units. Also due to the fact that the capacity limit is equal to the number of data required by a job, we cannot upload any datum that is not required by the current job. If every job transition requires only two new data, then the total time will be $3|J|+1$. Otherwise the time bound cannot be achieved. From the construction two jobs share data if and only if their corresponding nodes in G are adjacent. In other words, the graph G has a Hamiltonian path if and only if the makespan is $3|J|+1$. The theorem follows. □

This problem remains open when the number of dependent data per job is reduced to two — we do not know if the scheduling problem remains NP-complete, or there is a polynomial time algorithm that gives optimal schedules.

5. Heuristic Algorithm

Giersch et al. showed that when the number of data a job may depend on is *unbounded*, the decision problem of whether there exists a schedule within a given time bound is NP-Complete, even in the unlimited capacity model [4]. Nevertheless, we propose a heuristic algorithm that finds schedules for jobs that may depend on more than two data, with experimentally verifiable good performance. For jobs that depend on at most two data, we have already shown, in Section 3, that we can find optimal schedule for them.

We use L to indicate the set of jobs that have not yet been scheduled. Initially L is set to J, the data uploading

sequence $\mathscr{U}$ is set to empty, and all the data in D is mark as not yet uploaded.

The heuristic algorithm runs in iteration. In each iteration we choose a job j from L by a heuristic function. We then determine the set of data j requires, but not yet available at the server. This set of extra data needed to run j is denoted by W_j. We upload W_j to the server, mark them as uploaded, and append them to $\mathscr{U}$. We repeat this process until all jobs have been scheduled.

Now we describe three heuristic evaluation functions used to choose an job in each iteration.

5.1. Minimum-Upload-Maximum-Execute

The Minimum-Upload-Maximum-Execute (MUME) algorithm is inspired by the algorithm in Theorem 2, which gives optimal schedules for two-machine flowshop problem in [5]. In fact the two-machine flowshop problem is a special case of our shared-data scheduling problem, where there is no data sharing among jobs.

For each job j in L, we calculate the time to upload W_j to server (denoted by T_u), and the time to run job j (denoted by $T_e = |j|$). We then split jobs into two groups — the first group consists of jobs whose T_u is less or equal to T_e, and the second group consists of the other jobs. If the first group is not empty, we choose the job j that has the minimum T_u. Otherwise we choose a job j that has the maximum T_e from the second group.

5.2. Earliest Completion First

The Earliest Completion First (ECF) is a common technique in job scheduling. The intuition is that by scheduling the job with the earliest expected finishing time first, it is likely that we are able to reduce the makespan.

For each jobs in L, we choose the job that has the earliest expected completion time. Suppose that from the previous iteration the data upload finishes at time T_u and job execution finishes at T_e, then the completion time of a job j is $\max\{\ T_u + \sum_{d \in W_j} |d|, T_e\} + |j|$. We simply choose the job j that minimizes this expected finishing time. After we choose the earliest completion job j, the new T_u is updated as $T_u + \sum_{d \in W_j} |d|$ and the new T_e is $\max\{T_u, T_e\} + |j|$.

5.3. Random

We randomly choose a job j from unscheduled jobs L. This method is mainly used for comparison purpose.

6. Experimental Result

We compare the three heuristic algorithms and observe their relative performance. We also compare their makespan with a theoretical lower bound in order to determine the absolute performance.

The parameters in our simulations are as follow. There are 200 data in the system. Each job requires a set number of randomly chosen data and the job execution time is randomly chosen from 1 to 10.

The X coordinate of these figures indicates the number of jobs and the Y coordinate indicates the ratio between the makespan and the theoretical lower bound. Let $S(d)$ be the total job execution time for jobs that require datum d, and $S(j)$ be the total upload time for data that are required by job j, the theoretical lower bound T_L is maximum of the total data upload time plus the minimum $S(d)$ for all d in D and the total job execution time plus the minimum $S(j)$ for all j in J.

$$T_L = \max\{\sum_{d \in D} |d| + \min_{d \in D} S(d), \sum_{j \in J} |j| + \min_{j \in J} S(j)\}. \quad (7)$$

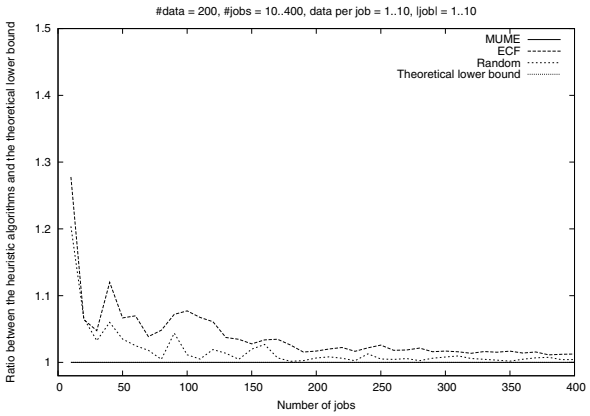

Figure 6. The ratio between the makespan of heuristic algorithms and the theoretical lower bound, measured from the case with 200 data, and 10 to 400 jobs, each job requires 1 to 10 data.

In Figure 6, each job requires n randomly chosen data, where the number of data n is also randomly chosen between 1 to 10. Figure 6 indicates that the Minimum-Upload-Maximum-Execute (MUME) heuristic produces schedules that whose makespan is almost identical to the theoretical lower bound. In fact we cannot even distinguish it from the theoretical lower bound in Figure 6.

In Figure 7 and Figure 8, each job requires 10 to 20 and 20 to 30 randomly chosen data respectively. When

we compare Figure 6, Figure 7, and Figure 8, we observe that when jobs requires more data, it becomes difficult to find good schedules. For example, most of the ratio by MUME in Figure 6 is 1, and rise to 1.07 in Figure 7 in the worst case, and continue to rise to 1.17 in Figure 8.

We also observe that there are "peaks" in both Figure 7 and Figure 8 when the number of jobs is about 30. The reason may be that our lower bound estimation is least accurate when the number of jobs is about 30. Recall that the theoretical lower bound is the maximum of the total data upload time plus the minimum $S(d)$ for all d in D and the total job execution time plus the minimum $S(j)$ for all j in J. When the number of jobs is about 30, the total data upload time plus the minimum $S(d)$ will dominate the total time. However, in average a datum will be shared by more than three jobs when the number of jobs is 30. That means there will be many remaining job that require data other than the last datum we upload after all the data are ready. This will be very different from the lower bound, which assumes only jobs requiring the last datum will remain.

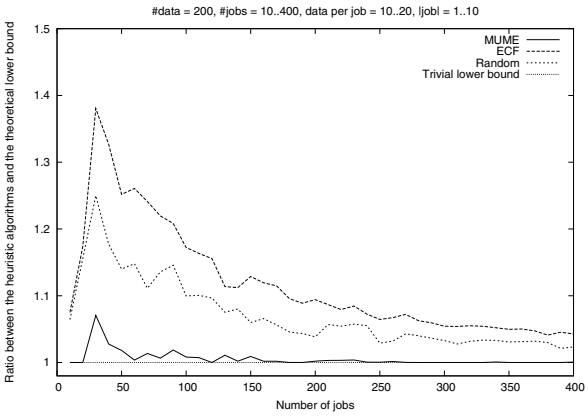

Figure 7. Same as Figure 6, except each job requires 10 to 20 data.

7. Conclusion

In this paper we analyze the computational complexity of the shared job scheduling problem, with and without consideration of storage capacity constraint. We show that if there is an upper bound on the server capacity, the problem is NP-complete, even when each job depends on at most *three* data. On the other hand, if there is no upper bound on the number of data on the server, we show that there exists an efficient algorithm that gives optimal job schedule when each job depends on at most *two* data.

We also give an efficient heuristic algorithm that

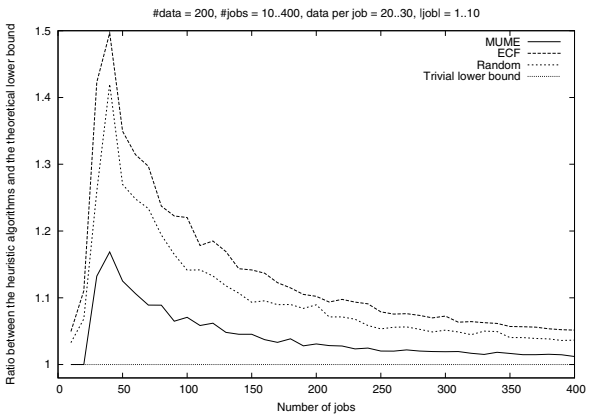

Figure 8. Same as Figure 6, except each job requires 20 to 30 data.

gives good schedule for unlimited capacity model where there is no limit on the number of data a job may require. Experimental results indicate that this heuristic algorithm performs very well. In some cases the derived makespan is almost identical to the theoretical lower bound. In most cases the performance is within 20% of the theoretical lower bound.

For limited capacity model, the problem remains open when the number of dependent data per job is reduced to *two* — we do not know if the scheduling problem remains NP-complete, or there is a polynomial time algorithm that gives optimal schedules.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The apples parameter sweep template: user-level middleware for the grid. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 75–76, 2000.

[3] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the ACM symposium on Theory of computing*, pages 47–63, 1974.

[4] A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. *Journal of Systems Architecture*, 52(2):88–104, 2006.

[5] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954.

[6] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proceedings of the Heterogeneous Computing Workshop*, pages 30–44, 1999.