631-067 Last 10/14

SCHEDULING INDEPENDENT TASKS TO MINIMIZE COMPUTATION AND COMMUNICATION MAKESPAN IN DISTRIBUTED SYSTEMS

Teofilo F. Gonzalez
Department of Computer Science
University California
Santa Barbara, CA 93106-5110, USA
email: teo@cs.ucsb.edu

ABSTRACT
We study the problem of scheduling tasks in a distributed system where the data (and code) for a program may reside on a processor different from the one where it will be executed. The scheduling of the tasks is complex as one must balance execution and communications times. We present an off-line polynomial time approximation algorithm for the case when the processors can be split into storage (client) and processing (server) nodes. Our algorithm is the first constant ratio approximation algorithm for this problem. Then we discuss generalization of our problem as well as the on-line version of our problem.

KEY WORDS
Deterministic scheduling, approximation algorithms, client-server model, dual optimization criteria.

1 Introduction

Scheduling problems arising from several research areas have been studied for more than five decades. The initial work was in operations research, computer science, and applied mathematics. More recently scheduling has been studied in the context of parallel, cluster, and grid computing. Our model falls under the broad umbrella of scheduling data-intensive distributed applications as well as grid scheduling. Heuristic algorithms for different variations of our problem were initially reported in [1, 2, 3, 4]. Our algorithm is the first constant ratio approximation algorithm for our scheduling problem, i.e., the schedule makespan is guaranteed to be within a given percentage of the optimum one. In this paper we discuss the problem of scheduling in distributed systems where the data (and code) for a program might not reside on the processor that will process the program. This scheduling problem is more complex than traditional ones, as we need to balance processors so that the total processing time of the programs assigned as well as the time required to transmit the data needed to process the program.

The system consists of a set of M nodes denoted by $1, 2, \ldots, m$. Each node j consists of m_j processing elements (processors). For example, a node may be a parallel processor system, a cluster of computers, or a single computer with one or more cores. There are n independent tasks denoted by $1, 2, \ldots, n$ to be processed. Task i requires t_i units of time to be scheduled for processing by a processing element.

The data (and code) for each task i is initially assigned to node s_i and, depending on the number of processing elements at the nodes and the total demand on the system, the task may or may not end up being scheduled for execution by a processor in node s_i. If a task is not scheduled to be executed by one of the processors in node s_i, then data requiring d_i units of time need to be transmitted from node s_i to the node where task i is to be processed. All the task's data must be available before a processor can start processing of the task. The data communication between nodes is performed through a channel in the communication network N. Each node j has c_j bi-directional channels. The basic assumptions we make is that any one-to-one interconnection between nodes through the channels is supported by the communication network, and that the routing to achieve this data interchanges can be computed in polynomial time. As a large number of interconnection networks satisfy these properties, the assumptions are not restrictive. From these assumptions we know that each data (file) to be sent from node i to node j we need to specify the time interval when the communication will take place, the out channel to be used in node i and the in channel to be used in node j. Of course, the restriction is that data from two or more programs cannot be transferred using the same channel of the same node at the same time.

In this paper we consider the bipartite version of the problem when the set of nodes is partitioned into two sets called: *Storage (Client)* and *Processing (Server)* nodes. The data needed by the tasks is stored at the storage nodes and the processing of the tasks is to be performed at the processing nodes. To simplify the notation, assume that the first w nodes $(1, 2, \ldots, w)$ are the storage nodes, and the remaining ones $(w+1, w+2, \ldots, m)$ are the processing nodes. It is assumed that the number of channels for each of the processor nodes is identical, i.e., c_j is equal to the number of processors at node (n_j). In every storage node we pair together processors and channels, in such a way that there is a 1-1 correspondence between processors and channels. For the storage nodes, $c_i \geq 1$.

Given a set of independent tasks, their location and data requirements, our problem is to find the node, pro-

cessor, and time at which each task is to be executed, as well as the channels and time where the data (file) required by each task needs to be transmitted. Our scheduling attempts to balance as much as possible the processing and communication times. We discuss the centralized (offline) algorithm to be processed by a central processor that knows all the global information. In Section 5 we discuss the portions of our scheduling algorithm which can be performed significantly faster in a distributed (on line) fashion when processors know only local information.

In Section 2 we discuss some well-known scheduling procedures that our algorithm will invoke as subprocedures. Our scheduling algorithm is discussed in Section 3. In Section 4 we discuss a new approximation algorithm for the two-component vector scheduling problem, which is a central component of the proposed scheduling algorithm in this paper. Then in Section 5 we discuss extensions of our results.

2 Preliminaries

In this section we survey well-known scheduling problem as well as exact and approximation algorithms for their solution. These algorithms are used by the approximation algorithm proposed in this paper.

2.1 Openshop Scheduling

An openshop consists of $m \geq 1$ machines, and $n \geq 1$ jobs. Each job consists of m tasks. The j^{th} task of job i must be processed by the j^{th} machine for $p_{i,j} \geq 0$ time units. We use r to denote the number of tasks with non-zero processing requirements and we use the triplet (P, n, m) to denote a problem instance. A schedule is an assignment of each task to its corresponding machine for a total of $p_{i,j}$ time units in such a way that at each time unit at most one task from each job is assigned to a machine, and each machine is assigned at most one task at a time. A *non-preemptive* schedule is one where every task must be scheduled for processing without interruption. In a *preemptive* schedule the processing of a task may be interrupted and resumed at a later time. The makespan (finish time) for schedule S, denoted by $f(S)$, is the latest point in time a task is scheduled to be processed by a machine. The minimum makespan openshop scheduling problem is giving any problem instance construct a schedule with minimum finish time (makespan).

Given an instance (P, n, m) of the openshop problem, let y_j be the total time that machine j must be busy processing tasks, and x_i be the total time that job i needs to be processed. Let $t = \mathbf{max}\{x_i, y_j\}$. Gonzalez and Sahni [5] have shown that there is always a preemptive schedule with makespan t and one such schedule can be constructed in $O(r(min\{r, m^2\} + m \log n))$ time.[1] The makespan is best possible. Furthermore, when all the $p_{i,j}$s are integers, there is a schedule where preemptions occur only at integer points, and one such schedule is generated by the algorithm in Ref. [5]. For the case of non-preemptive schedules, the problem is NP-complete even when there are only three machines. There are several approximation algorithm for both versions of the problem. The fastest and simplest one is an $O(r \log m)$ time list scheduling algorithm that generates schedules with makespan at most two times the optimum makespan [6].

[1] The preemptive openshop problem can be modeled as the problem of coloring the edges in a multigraph [6].

2.2 Scheduling Identical Processors

The problem of scheduling independent jobs on identical machines is a well known problem that has been studied for the past four decades. The input to the problem is a set of n independent jobs with execution time requirements given by $p_i > 0$ to be scheduled for processing on m identical machines. A schedule is an assignment of jobs to machines in such a way that at any given time every machine is scheduled to process at most one job and each job is assigned to at most one machine. A schedule is *non-preemptive* if every job is scheduled for processing during one continuous time interval, Otherwise the schedule is called *preemptive*. The makespan (finish time) for schedule S, denoted by $f(S)$ is the latest point in time a job is scheduled to be processed by a machine. The minimum makespan identical machine scheduling problem is to construct for any instance of the problem a schedule with minimum finish time (makespan). Constructing an optimum makespan preemptive schedule for any instance of this problem takes linear time with respect to the number of jobs and machines. However, the corresponding non-preemptive scheduling problem is an NP-complete problem. There are many well known algorithms to generate near-optimum non-preemptive schedules, e.g., List [7] and LPT [8] schedules. The former procedure generates schedules in $O(n \log m)$ time with a makespan that is within 2 times the optimum makespan, and the latter algorithm takes $O(n \log n)$ time and generates schedules with makespan at most $(4/3 - 1/3m)$ times the optimum makespan. Additional information about scheduling identical processors can be found in [6, 9].

2.3 Vector Scheduling on Identical Machines.

The vector scheduling problem for identical machines is a well known generalization of the above problem. The difference is that the processing requirement of each job is given by d-component vector, for example $p_{i,1}, p_{i,2}, \ldots p_{i,d}$. The makespan (finish time) of a schedule S, denoted by $f(S)$, is the maximum over each machine j and component k of the sum of the processing time of the k^{th} component of the jobs assigned to machine j in schedule S. For this problem we are just interested in non-preemptive schedules. There is a simple $O(n \log m)$ time algorithm that generates schedules with makespan at

most $d+1$ times the optimum makespan [10]. Chekuri and Khanna [10] developed an algorithm that generates schedules with makespan at most $O(\ln^2 d)$ times the optimum, as well as one with a smaller approximation ratio, $O(\ln d)$, for the case when d is bounded by a constant. However, the constant associated with the approximation ratio is not so small. Chekuri and Khanna [10] developed a polynomial time approximation scheme (PTAS) for the case when d is bounded above by a constant. However this algorithm is very slow in practice. In Section 4 we present a fast linear time algorithm that generates schedules with makespan at most 2 times the optimum makespan.

3 Approximating the Bipartite Problem

As we said before we consider the bipartite version of the problem where the set of nodes is partitioned into two sets called: *Storage (Client)* and *Processing (Server)* nodes.

Let us outline our `two-phase` approximation algorithm based on *restriction*, i.e., generate a schedule for a restricted version of the bipartite problem. The restriction is in the type of schedules it generates. The schedules consist of two separate portions (phases) of a *communication* schedule that takes care of all the communications, followed by a *computation* schedules that takes care of all the processing. The general approach for our scheduling algorithm is as follows.

- Determine the processor where each task i is to be executed and identify the corresponding (in-) channel to be used to receive the data for task i.
- Determine the (out-) channel to be used to send the data for task i.
- Construct the communication schedule $Comm$, i.e., determine the actual time when the data required by the tasks will be sent from the storage node via the out-channel to the receiving processing node via the in-channels.
- Construct the computation schedule $Comp$, i.e., determine the actual time when each task is to be processed.

The basic steps of the above procedure are implemented by solving different scheduling problems. The first step is implemented by solving a two-component vector scheduling problem; step two by scheduling a set of independent jobs on identical machines; the third one by solving an openshop scheduling problem; and the last one, is the simplest one, as the ordering is determined by the ordering of the data arriving to the processor. In what follows we explain the step in our procedure and then we formally specify our algorithm.

Step 1: We determine the processor and in-channel to be used to process and receive the data for each task. This is established by constructing a schedule S_1 (by the algorithm given in Section 4) for the two-component vector scheduling problem P_1 defined below. Let p be the total number of processors (as well as the number of in-channels) in the processing nodes, i.e., $p = \sum_{j=w+1}^{m} n_j$, and q be the total number of out-channels for the data required to process the tasks, $q = \sum_{j=1}^{w} n_j$. The first n_{w+1} processors correspond to node $w+1$, the next n_{w+2} processors correspond to node $w+2$, and so on.

We construct the instance P_1 of the two-component vector scheduling problem as follows. For each task i we define job i with its x-component as t_i and its y-component as d_i. Define

$$T = min\ \{\frac{\sum t_i}{p},\ max\ t_i\}, D = min\ \{\frac{\sum d_i}{p},\ max\ d_i\}$$

$$\text{and } L = max\ \{T, D\}.$$

Clearly the x-component and y-component of each one of the tasks is a value between 0 and L. The sum of the x-components of all the tasks is at most pL and the sum of the y-components of all the tasks is at most pL. We construct a schedule S_1 for the instance P_1 by using the linear time algorithm given in Section 4. All the tasks assigned to the same machine in schedule S_1 will be assigned to the same processor for their execution and their data will be received by the in-channel corresponding to the processor.

As we establish in the next section, in schedule S_1 all the tasks assigned to the same machine are such that the sum of their x-component is at most $2L$ and the sum of their y-component is at most $2L$. Therefore, every processor will be running tasks for at most $2L$ time units, and every in-channel will be receiving data for at most $2L$ units of time.

Step 2: Now lets decide the out-channel to be used to send the data for each task to the processor where the task will be executed. For every storage node k, our algorithm partitions the data stored at node j into c_j groups (remember c_j is the number of out-channels). The data for each task assigned to each group will be sent via a different out-channel to the node where the processing of the task will take place. This partitioning is such that the sum of the communication times of all the data for the tasks assigned to any of the out-channels is least possible. It is well known that this partitioning problem is NP-hard under the assumption that all the data needed by a task has to be transmitted using the same channel.[2] For this version of the problem one can generate a near-optimum solution by modeling the partitioning problem as an instance P_2 of the problem of scheduling independent jobs on identical machines (which is the same as the single-component vector scheduling problem). The execution time for the job is the time required to transmit the data for the task. We can use any of the scheduling algorithms discussed in the previous section to produce a near-optimum schedule S_2 which is then used to obtain near-optimum balanced partitions. If

[2]However, if one can transmit the data using two or more channels, even if one can only transmit the data on one of the channels at a time, the complexity of the problem is different.

we use list scheduling [7] then one can construct schedules with makespan at most 2 times the makespan of an optimum schedule in $O(n \log m)$ time. On the other hand, LPT generates schedules with makespan (finish time) at most $(4/3 - 1/3m)$ times the makespan of an optimum schedule [8].

To summarize, we have determined for every task i the out-channel that will be used to send the data it needs as well as the in-channel that will receive it, and the corresponding processor where the task will execute. We need to construct the communication schedule $Comm$ and the computation schedule $Comp$.

Step 3: The timing of all the communication events is obtained by modeling the problem as an openshop scheduling problem, which we define below. Before defining the openshop instance P_3 it is convenient to begin by defining the bipartite multigraph G consisting of the set of vertices X and Y. Each vertex in set X represents a storage node and one of its communication out-channels. Similarly, each vertex in set Y represents a processor in a processing node and its corresponding communication in-channel. At this point we know the out-channel and in-channel for the transmission of the data for each task. So we use this information to define the edge for each task joining a vertex in X to a vertex in Y. The weight of the edge is d_i, the communication requirement to transmit the data for the corresponding task. Each node in X represents a job and each node in Y represents a machine in the instance of the openshop problem P_3 we construct. We define as $p_{i,j}$ the sum of the weight of the edges joining vertex i in X to vertex j in Y. Let x_i to be the sum of the weight of the edges incident to vertex i in X, i.e. $\sum_j p_{i,j}$, let y_j to be the sum of weight of the edges incident to vertex j in Y, i.e. $\sum_i p_{i,j}$. We define t as $max\ \{x_i, y_j\}$. From [5] we know that there is a preemptive communication schedule S_3 with makespan t for P_3. The schedule S_3 for the actual processing of the jobs by the machines can be constructed by using well known algorithms [5]. From schedule S_3 one can construct schedule $Comm$ that gives the specific times when the data for task i must be transmitted from the storage node where it resides to the processing node where it will be processed using the channels that have been previously selected.

Step 4: The computation schedule $Comp$ is constructed in this last stage. I.e., we determine the exact times when the processing of the tasks will take place. Since we already know which tasks will be processed by each of the processors, the ordering of the tasks may be arbitrary. However, to reduce the makespan, it is better to use the ordering given by the arrival of the data for the tasks at each in-channel. Therefore $Comm$ is the schedule based on this ordering where task i will start processing at the latest of $\{t_1, t_2\}$, where t_1 is the time at which all the tasks in the ordering before task i have completed and t_2 is the time at which all the data for task i has arrived to the processor where the task is to be processed.

Our two-phase algorithm is formally defined below.

`Two-Phase Algorithm`
Let $p = \sum_{j=c+1}^{m} n_j$; //number of processors.
Let $q = \sum_{j=1}^{c} n_j$; //number of out-channels

// Step 1: Determine the processor where each task i is to
// be executed and identify the corresponding in-channel
// to be used to receive the data for task i.
Construct the instance P_1 of the two-component vector scheduling problem as follows.
For each task i we define its x-component as t_i and its y-component as d_i.
$T = min\ \{\frac{\sum t_i}{p},\ max\ t_i\}$; $D = min\ \{\frac{\sum d_i}{p},\ max\ d_i\}$
$D = min\ \{\frac{\sum d_i}{p},\ max\ d_i\}$; $L = max\ \{T, D\}$
Construct schedule S_1 for P_1 via the algorithm given in Section 4;
Assign all the tasks corresponding to the jobs scheduled on the same machine in S_1 to the same processor and corresponding in-channel.

// Step 2: Determine the out-channel to be used to send the data for task i.
For each storage node k define an instance P_2 of the problem of scheduling independent jobs on identical machines. For each task stored at node k define a job with execution time equal to the d_j, the time required to transmit the data for task j which is stored in node k, and define the number of machines as c_k.
Use the list scheduling [7] to construct a non-preemptive schedule S_2 for P_2.
All the tasks corresponding to the jobs assigned to the same machine in S_2 will be using the same out-channel in storage node k.

// Step 3: Construct the communication schedule $Comm$,
Define the bipartite multigraph $G = (X \cup Y, E)$.
There is a vertex in X for each storage node and one of its communication out-channels.
There is a vertex in Y for each processor in a processing node and its communication in-channels.
For each task i we define an edge from the vertex in x representing the out-channel O_i to the vertex in Y representing the in-channel p_i with the weight of the edge equal to d_i.
Each vertex in X represents a job and each vertex in Y represents a machine in the instance of the openshop problem P_3 we construct.
Let $p_{i,j}$ the sum of the weight of the edges joining vertex i in X to vertex j in Y
$x_i = \sum_j p_{i,j}$; $y_j = \sum_i p_{i,j}$, $t = max\ \{x_i, y_j\}$.
The algorithm in Ref. [5] constructs the preemptive schedule S_3 for P_3 with makespan t.
Schedule $Comm$, which can be easily constructed from S_3, defines the specific times when the data for task i must be transmitted from its in-channel to its out-channel in such a way

that each channel transports the data for at most one task at a time.

// Step 4: Construct the computation schedule $Comp$
Construct the schedule $Comp$ that specifies for each processor the order in which the tasks assigned to it are to be processed. The ordering is the same one as the order in which their data arrives to the processor in its in-channel.
End of Two-Phase Algorithm

Theorem 3.1 *Our two-phase algorithm takes $O(np(p + \log q))$ time and generates schedules with makespan at most four times the makespan of an optimum schedule.*

Proof: The number of tasks, nodes, processors and out-channels is n, m, p and q, respectively. In Step 1 we construct an instance of the two-component scheduling problem. Constructing this instance takes $O(n + p)$ time and constructing a schedule for the instance takes $O(n+p)$ time (Section 4).

Step 2 constructs an instance of the identical machine scheduling problem. This can be accomplished in $O(n+q)$ time and constructing an LPT schedule for it takes $O(n \log q)$ time [8].

Assigning the actual time when the data for each task will be transmitted from the out-channel to the in-channel is determined by solving an instance of the openshop preemptive scheduling problem. Constructing the instance P_3 of the openshop problem takes $O(n + p + q)$ time. Constructing the schedule S_3 for the openshop instance takes $O(np(p + \log q))$ time, as the number of processors is p, the number of jobs is q and the number of non-zero tasks is $O(n))$ [5].

Finally, Step 4 takes time $O(n+q)$, as one may simply use the ordering of the data arriving to each processor.

Hence, the overall time complexity is dominated by the solution to the instance P_3 of the openshop problem, which takes $O(np(p + \log q))$ time.

Let us now determine the approximation factor for our approximation algorithm. The total time required to process the tasks is at most $2L$, where L is a lower bound for the total time required for the processing of the tasks by the p processors and a lower bound for the total time required to receive all the data by the p in-channels.

The total time required to send all the data for the tasks by the q processors is at most $(4/3 - 1/3m)L'$, where L' is a lower bound for the time required to send all the data by the q processors.

The solution to the openshop problem is a communication schedule with makespan at most $max\{2L, (4/3 - 1/3m)L'\}$. Therefore, the makespan of the communication schedule is at most $2L$. Since an optimum makespan, f^* is at least $min\{L, L'\}$, it then follows that the schedule we have constructed has makespan at most $4f^*$, where f^* is the makespan of an optimum schedule. This concludes the proof of the theorem.

□

From the proof of Theorem 3.1 on can gather that the time complexity is dominated by the time required to solve the openshop problem P_3. A sub-optimum algorithm for the openshop problem (Sub-section 2.1) can be used at the expense of increasing the makespan. This results in a schedule with makespan at most 8 times the optimum one, and the time complexity is dominated by $O(n \log q)$.

4 Approximating the Two Component Vector Scheduling Problem

In this section we present an algorithm to construct a schedule with makespan at most twice the finish time of an optimum makespan schedule for the two-component vector scheduling problem. The two-component vector scheduling problem consists of n independent jobs and m identical machines. Job i has the two-component pair (x_i, y_i) specifying its two component processing time. Define $T = min\ \{\frac{\sum x_i}{m},\ max\ x_i\}$, $D = min\ \{\frac{\sum y_i}{m},\ max\ y_i\}$, and $L = max\ \{T, D\}$. During the construction of our schedule we assign jobs to machines. Let P_j be the set of jobs assigned to processor j. We say that j_x (j_y) is the sum of the x-component (y-component) of the jobs in P_j assigned to it. A processor is said to be of type

$$\begin{array}{ll} E, & \text{if } 0 \le j_x \le L \text{ and } 0 \le j_y \le L; \\ F_x, & \text{if } L < j_x \le 2L \text{ and } 0 \le j_y \le L; \\ F_y, & \text{if } 0 < j_x \le L \text{ and } L < j_y \le 2L; \text{ and} \\ F_{xy}, & \text{if } L < j_x \le 2L \text{ and } L < j_y \le 2L. \end{array}$$

We say that a job i fits in processor j if the x-component of job i plus j_x is at most $2L$, and the y-component of job i plus j_y is at most $2L$. Processors j and j' are said to be x-compatible and y-compatible if $j_x + j'_x \le 2L$ and $j_y + j'_y \le 2L$, respectively. Processors j and j' are said to be xy-compatible if they are both x-compatible and y-compatible. Processors j and j' are said to be *incompatible* if they are not x-compatible nor y-compatible. Initially every processor is said to be *unmatched*. During the execution of our algorithm we will identify pairs of processors and say that each pair is *matched* in such a way that each processor will be matched to at most one other processor.

Procedure Approx (X, Y, n, m)
Initially P_j is empty for $1 \le j \le m$ and therefore
all processors are of type E;
for i=1 to n do
case
:There is a type E processor:
Let j be a type E processor;
Assign job i to processor j;
:else:
Let j be an unmatched type F_x processor;
Let j' be an unmatched type F_y processor;

```
    while job i does not fit in processor j or j' do
     case
    :Processors j and j' are xy-compatible:
     Move all jobs from j' to processor j;
    :Processors j and j' are incompatible:
     Match processors j and j';
     Let j be an unmatched type F_x processor;
     Let j' be an unmatched type F_y processor;
    :Processors j and j' are x-compatible:
     while a job i' assigned to j' fits in processor j do
      Move job i' from j' to j
     endwhile
     if processor j' is type F_y then
       Let j be an unmatched type F_x processor;
    :Processors j and j' are y-compatible:
     while a job i' assigned to j fits in processor j' do
      move job i' from j to j'
     endwhile
     if processor j is type F_x then
       Let j' be an unmatched type F_y processor;
   endcase
  endwhile
  if job i fits in processor j
   then assign job i to processor j;
   else assign job i to processor j';
 endfor
endcase
endfor
End of Procedure Approx
```

Lemma 4.1 *The above algorithm generates a schedule with finish time at most $2L$ for any two-component vector scheduling problem.*

Proof: For brevity we do not include the proof. □

5 Discussion

We have presented a two-phase algorithm takes $O(np(p + \log q))$ time and generates schedules with makespan at most four times the makespan of an optimum schedule for the case when the set of nodes is partitioned into storage and processing nodes. We have shown that the time complexity bound can be decreased to $O(n \log q)$, but then we can only guarantee solutions that are within eight times the optimal one. Our algorithms can be extended for other cases, but for brevity we cannot include these results. Other versions of interest are when processors have different processing speeds and when all processors are both storage and processing nodes.

Another interesting problem is to transform our algorithm to a distributed one. The portion that is transformable to a distributed on-line algorithm is the solution to the openshop problem by using the algorithm developed by Anderson and Miller [11]. However one needs to solve on-line the other scheduling problems. List scheduling is an on-line algorithm, but it is not a distributed one. Transforming it to a distributed one as well as transforming our algorithm for the two-component vector scheduling problem, while maintaining the same approximation ratios at the same time decreasing significantly their time complexity bounds, are challenging open problems.

References

[1] Ranganathan, K. and Foster, I., "Computation and Data Scheduling in Distributed Data Intensive Applications," *Proc. of HPDC'02,* July 2002.

[2] Beaumont, O., Legrand, A., and Robert, Y., "Optimal Algorithms for Scheduling Divisible Workloads on Heterogeneous Systems," *Proc. of IPDPS'03,* 2003.

[3] Lampsas, P., Loukopoulos, T., Dimopoulos, F., and Athanasiou, M., Scheduling Independent Task Scheduling in Heterogeneous Environments under Communication Constraints," *Proc. of PDCAT'06,* 2006.

[4] Loukopoulos, T., Lampsas, P., and Sigalas, P., Improved Genetic Algorithms and List Scheduling Techniques for Independent Task Scheduling in Distributed Systems, *Proc. of PDCAT 2007,* 67 – 74, 2007.

[5] Gonzalez, T.F. and Sahni, S., "Open Shop Scheduling to Minimize Finish Time," *JACM,* 23(4), 665 – 679, 1976.

[6] Leung, J. Y-T., ed., Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Chapman & Hall/CRC, 2004.

[7] Graham, R.L., "Bounds for Certain Multiprocessing Anomalies," *Bell System Tech. J,* 45, 1563 – 1581, 1966.

[8] Graham, R.L., "Bounds on Multiprocessing Timing Anomalies," *SIAM J. of Applied Math.,* 17(2), 416 – 429, 1969.

[9] Gonzalez, T. F, ed., Handbook of Approximation Algorithms and Metaheuristics, Chapman & Hall/CRC, 2007.

[10] Chekuri, C. and Khanna, S., "On Multidimensional Packing Problems," *SIAM J. on Comput.,* 33(4), 837 - 851, 2004.

[11] Anderson, R.J. and G. L. Miller, G.L., "Optical Communications for Pointer Based Algorithms," TRCS CRI 88 – 14, USC, 1988.