Research Practice Report

ON

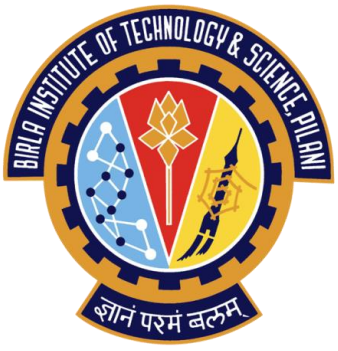
‘Scheduling Directed Acyclic Graphs with Optimal Duplication Strategy on Homogeneous Multiprocessor systems’

BY

Name of the Student	ID Number
NEDUNURI SAI CHARAN	**2020H1120281P**

SUBMITTED TO

Dr. Abhishek Mishra



BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

1. Scope and Introduction:

Modern day applications are often implemented on multiprocessor systems i.e. tasks of the application are allocated to different processors for execution. As tasks are mapped to different processors, obtaining the optimal performance on multiprocessor systems remains challenging due to inter-processor communications between tasks of application. To reduce this overhead incurred by inter-processor communications and improve the schedule performance, various strategies have been employed in the schedule.

1.1 Inter-processor communications overhead:

When tasks of the application are mapped onto different processors for execution, inter-processor communications happens between these during execution, thereby increasing the scheduling length. The reasons includes two or more processors, requires same Resource (Contention Problem). To restrict the access, synchronisation occurs in the system for making the operations atomic and mutual exclusion. To tackle this overhead, task duplication strategy has been employed in the schedule.

1.2 Existing Work

Scheduling is generally classified as three categories, list-based, cluster-based, duplication-based scheduling. List scheduling technique will assign various priorities, e.g., the s-level, b-level and t-level, to tasks during the scheduling. The tasks will be then ordered and scheduled as per the previously assigned priorities. Cluster based scheduling technique will bind tasks which most probably have large intercommunication as a cluster. This cluster is then scheduled on to the same processor. Therefore the tasks having the heavy inter – processor communication are now mapped on to the same processor and the schedule performance will be increased. Duplication-based scheduling, same tasks will be mapped to multiple processors for execution to reduce the schedule length. This approach used ILP for the non-duplication-based scheduling. This approach only focuses on reducing the communication cost, but the ordering and the data dependency and the timing constraints were not considered in this approach.

Full-duplication strategy: It duplicates immediate and indirect ancestors in a bottom-up fashion recursively. The various algorithms employing this strategy are listed below.

i. The Critical Path Fast Duplication (CPFD) algorithm, tries to duplicate the very important parent into the available time slot of the processor until either the time slot is used up or the task start time is not improved any more.
ii. The selective duplication (SD) algorithm, divided into three main steps, i.e. the task sequence generation phase, processor selection phase, and task assignment and duplication phase, was developed to improve the performance by selectively adding duplications of some tasks.
iii. **Drawbacks with Full Duplication**

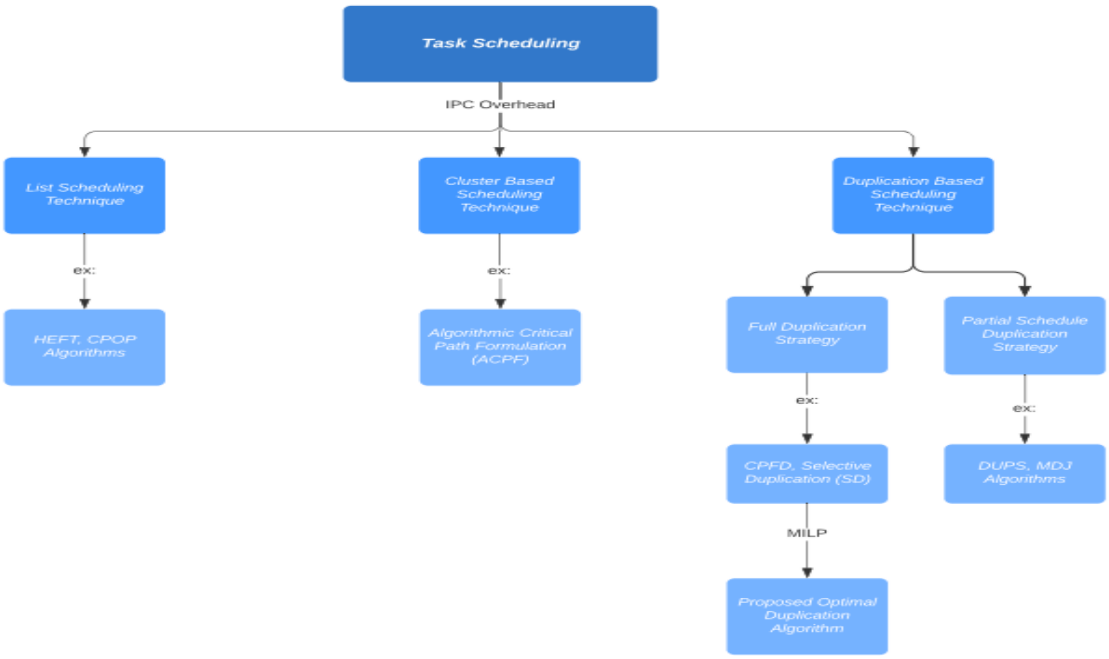
    i. It has larger time complexity.

ii. It incurs redundant duplications, which may increase the schedule length.

a. Duplication-based Scheduling Algorithm Using Partial Schedules (DUPS):

i. DUPS uses two stages. The schedule length is minimized by utilizing partial schedules in First phase; the number of processors required is minimized by eliminating and merging these partial schedules in the Second phase.

ii. The Minimized Duplication at Joint node (MDJ) algorithm: Proposes concept of join node. It schedules join nodes without redundant duplications. In the algorithm, if the ancestor nodes of a join node are duplicated when scheduling the join node, the original allocations of these ancestor nodes are removed using a very efficient method.



2. Task Duplication Strategy:

The communication between tasks assigned to the same processor is considered negligible, whereas tasks allocated to different processors for execution, then, inter-processor communication is inevitable, which delays execution. Owing to the delay-free shared memory based intra-processor communication, i.e., local communication, part of inter-task communications are made local by allocating the task to multiple processors for execution. To be a delay free local communication, the source task is duplicated to the processor where the destination task is mapped to, thus removing the communication delay. Task duplication based scheduling algorithms generate shorter schedules without losing efficiency but leave the computing resources over consumed due to the heavily duplications of tasks. There are various issues with duplication strategy which can be listed as follows.

2.1 Issues with Duplication:

a. Which task(s) to be duplicated?

b. How many times a task should be duplicated and where it should be duplicated?

c. How should tasks on each processor be ordered and timed?

d. How to determine the data precedence's among duplications of a task and its successors?

e. Finding the optimal duplication-based solution with the minimal schedule make span remains an issue.

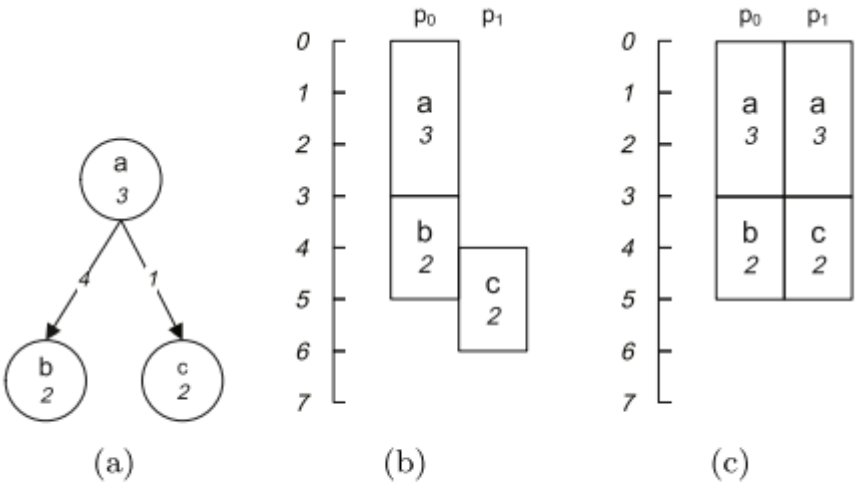


Fig. 1. (a) the example DAG, (b) the schedule without task duplication, (c) the duplication-based schedule.

2.2 Mixed Integer Linear Programming Approach:

The MILP has been widely studied for the scheduling problem. Proposed an application-specific knowledge aware MILP for scheduling DAGs.In the formulation, the bi-linear equation is simplified to mitigate the size of the formulation. This paper proposes a novel Mixed Integer Linear Programming (MILP) formulation, together with a set of key theorems which enable and simplify the MILP formulation and solve the problem with optimality. This approach is guaranteed to be optimal in terms of schedule length for the duplication--based scheduling problem. The task duplication is modeled appropriately using the task duplication variable in the formulation, thus resulting in optimal task duplication strategy. The reformulated data dependence is taken into account without a priori information about task duplication and mapping. A set of optimality theorems are summarized, which help to enable and simplify the modelling of the problem.

A DAG is represented as G = (V, E), V is tasks; E is a finite set of directed edges denoting data precedence among tasks. Each task v ∈ V has computation cost *c(v)*. Each edge e ∈ E is defined as a tuple (a, b), where *a, b* are source and destination tasks respectively. e_{ab} denotes the edge (a, b). Each edge e ∈ E has an integer communication cost *w(e)*, which is the time taken for data transfer. *src(e), dst(e)* represents source task and destination task , *oe(v)* are output edges and

succ(v) are successor tasks. Task without output is called the exit task, and the task without input edge or parent task is called the entry task of the DAG. V_{exit} and V_{entry} represents sets of exit and entry tasks respectively and the number of processors are taken as *P*. Task-to-processor mapping *map (v)* and task start time S_v should be done during scheduling, *f* and *sl* are finish times and schedule lengths respectively.

3. Proposed Optimality Theorems

a. **Lemma I.** There is an optimal duplication-based schedule in which no more than one instance of the same task is duplicated to execute on the same processor. This ensures that each processor have a unique set of tasks.
b. **Theorem I.** There is an optimal duplication-based schedule in which each duplication of the destination task of an edge depends on the output data of one and only one duplication of the source task of the edge. This says that there will be one to one dependency between source and destination tasks if at all dependency exist.
c. **Theorem II.** There is an optimal duplication-based schedule in which each of the duplication of non-exit task feeds data to at least one duplication of its successors. This ensures that there will be definite dependency between the source and the duplicated tasks among processors i.e. no useless executions taking place.
d. **Theorem III.** There is an optimal duplication-based schedule in which: for any pair of different tasks a, b in the DAG, if task b is the only successor of task a, then the data precedence between a, b is obeyed by the partial schedule of the optimal schedule on each processor.
    i. This says that though a, b tasks are mapped to different processors and there might be difference in execution time, but b will be mapped to some other task a on some processor thus satisfying the dependency.
    ii. This is represented as $drt_{bk} = f_{ai} + w(e) \geq f_{aj} + w(e) + c(b) + c(a) + w(e)$, where the data ready time of bk is obtained by redirecting the input data of bk from ai to aj the new maximal data ready time of bk_i
    iii. Redirecting the input data of b_k from a_i to a_j, the new maximal data ready time of b_k is drt' $b_k = f_{aj} + w(e)$.
e. **Theorem IV.** There is an optimal duplication-based schedule in which the data precedence between the computation-dominated task (CDT) and all its successors are obeyed by the partial schedule of the optimal schedule on each processor.
    i. This means that though there can be more than one successor for a task mapped to a processor and its successors are mapped to different processors. There can be difference in execution times between parent task and its successors but the successors will be mapped to some other duplication of parent task and the dependency will be preserved.
    ii. If one dependent task executes before source task, then logically, all its dependent tasks can be mapped to other source making the original source node redundant.
    iii. For a Computationally Dominating Task, where it has one or more dependent tasks, the following equation holds in terms of computation cost and communication cost, for a given outgoing edge on a processor.
    $C(v) + w(e1) + c(dst(e1)) \geq w(e2), \forall e1, e2 \in oe(v), e1 \neq e2$

4. Duplication-based mixed integer linear programming formulation

In the process of milp problem formulation, there are mainly two problem constraints which decide the entire schedule length. These constraints were carefully modelled such that every possible inter task dependency and inter task communications, order of task executions are carefully formulated. The problem constraints to be imposed are as follows.

a. Constraints on order of Task Execution.
b. Constraints on Dependencies between Tasks.
c. Objective Function which minimises the schedule length.

4.1 Constraints on the order of Task Execution:

These are the constraints which are imposed in order to define the timing of the execution of tasks. Tasks are categorised into various types like entry task, exit tasks, redundant tasks, source task which is the parent task and dependent tasks which depends on some source task during the order of their execution in the schedule. These can be mathematically formulated as follows.

a. Entry task should execute only once.

$X_{mi} + X_{mj} = 1, \forall\ m \in \{a, b\}$

b. Non entry task can be executed >= 1 times.

$X_{vi} + X_{vj} \geq 1$

c. When a redundant duplication is found, its Start Time should be limited to zero i.e. the Redundant task never gets executed in the schedule.

$S_{mk} \leq M * X_{mk}\ , \forall\ m \in \{a, b\}, k \in P$

d. If both Source task and Dependent task are mapped to same processor, then their execution should be ordered i.e. Source task should execute before the Dependent task.

$f_{vk} - S_{mk} \leq M * (2 - X_{vk} - X_{mk)}, \forall\ m \in \{a, b\}, k \in P$

e. Since there are dependents to the task, the Finish time of the Source task and Start Times of the Dependent tasks should be ordered.

$f_{ak} - S_{bk} \leq M * (2 - X_{ak} - X_{bk}\ ) + M * Z_{a,b,k}, \forall\ k \in P$

$f_{bk} - S_{ak} \leq M * (2 - X_{ak} - X_{bk}\ ) + M * (1 - {}_{Za,b,k}), \forall\ k \in P$

4.2 Constraints on Dependencies between Tasks:

These constraints were imposed in order to order the dependencies, to avoid any unnecessary dependencies and redundant executions among tasks. There were four major dependency constraints in the milp formulation of our scheduling problem which are as follows.

a. There should be at least one dependent successor task for a source task to be irredundant.

$d_{vkml} \leq X_{vk}$, $\forall$ m $\in$ {a, b}, k, l $\in$ P

b. Redundant dependencies should be removed as this result in redundant duplications.

$d_{viai} + dv_{iaj} + d_{vibi} + d_{vibj} \geq X_{vi}$
$d_{vjai} + d_{vjaj} + d_{vjbi} + d_{vjbj} \geq X_{vj}$

c. The successor dependent task should depend only on one source task in the schedule.

$d_{vimi} + d_{vjmi} = X_{mi}$, $\forall$ m $\in$ {a, b}

$d_{vimj} + d_{vjmj} = X_{mj}$, $\forall$ m $\in$ {a, b}

d. Since tasks are dependent, the dependent task should start execution right after the finish time of the source task.

$f_{vi} - S_{mi} \leq M * (1 - d_{vimi})$, $\forall$ m $\in$ {a, b}

$f_{vj} - S_{mj} \leq M * (1 - d_{vjmj})$, $\forall$ m $\in$ {a, b}

$f_{vi} + W(e1) - S_{aj} \leq M * (1 - d_{viaj})$

$f_{vj} + W(e1) - S_{ai} \leq M * (1 - d_{vjai})$

$f_{vi} + W(e2) - S_{bj} \leq M * (1 - d_{vibj})$

$f_{vj} + W(e2) - S_{bi} \leq M * (1 - d_{vjbi})$

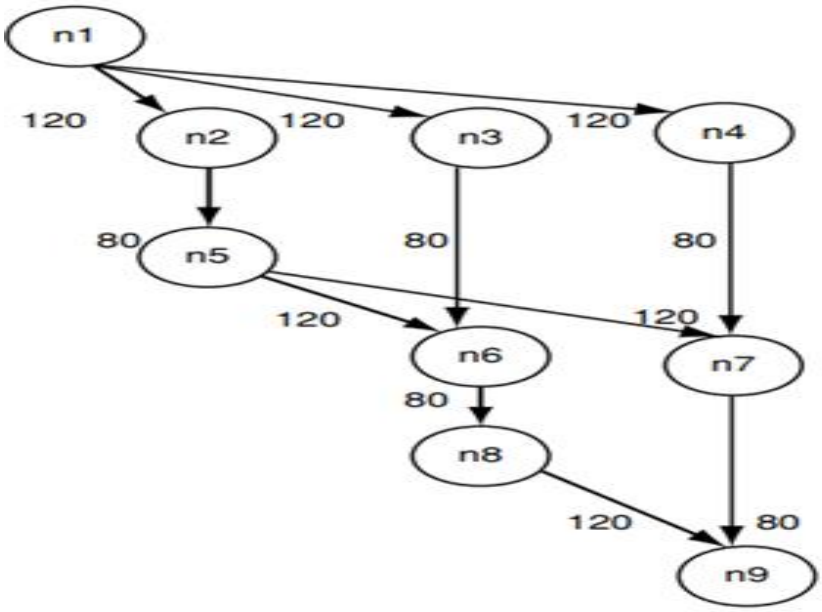
4.3 Objective Function:

The main purpose in the problem of task scheduling is to reduce the schedule length. The schedule length in a schedule can be taken as the maximum of all finish times of tasks that exists in the schedule. Therefore the objective function can be mathematically formulated as follows.

$f_{mk} \leq SL$, $\forall$ m $\in$ {a, b}, k $\in$ P

5. Evaluation Results:

A simple dag is considered for implementation which consisting of 9 nodes representing tasks in the schedule, which are having interdependencies where the order of dependencies and timing of execution is important is chosen. This schedule with sample set of 9 tasks is formulated into the milp as discussed in the paper and the problem is solved with twice to capture the effect of imposing the constraints on the schedule.



The Communication delays and execution times were changed from above graph, but the network for scheduling remains same during the execution.

a. First time the problem is solved with the optimality theorems with having sample schedule and an objective function to minimise the schedule length which in turn same as finding maximum finish time. O

Working code:

No_Constraints_data file

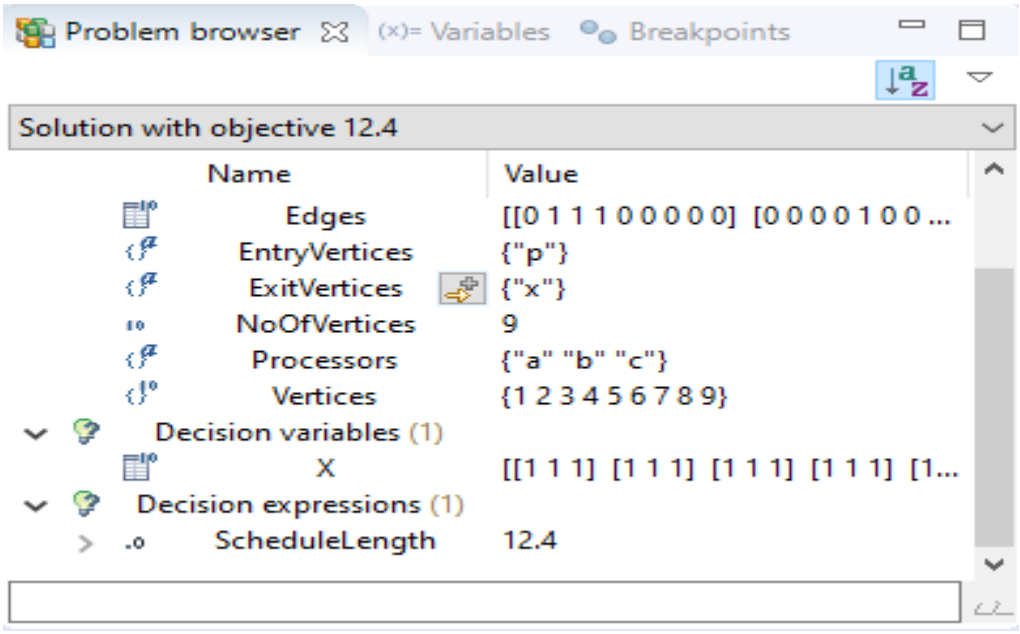
```
/*********************************************
 * OPL 20.1.0.0 Data
 * Author: admin
 * Creation Date: 15-Jun-2021 at 5:14:15 pm
 *********************************************/
 NoOfVertices = 9; //No of vertices taken
EntryVertices = {p};
ExitVertices = {x};
Processors = {a,b,c};
Computation = [ 0.1 0.2 0.1 0.1 0.2 0.1 0.1 0.2 0.1 ]; // computation cost
of tasks
Edges = [[ 0 1 1 1 0 0 0 0 0]  //  Interconnection between tasks.
               [ 0 0 0 0 1 0 0 0 0]
               [  0 0 0 0 0 1 0 0 0]
                [  0 0 0 0 0 0 1 0 0]
                 [  0 0 0 0 1 1 0 0 0]
                  [  0 0 0 0 0 0 0 1 0]
                   [  0 0 0 0 0 0 0 0 1]
                    [  0 0 0 0 0 0 0 0 1]
                     [  0 0 0 0 0 0 0 0 0]];
Communication = [[ 0 0.1 0.1 0.1 0 0 0 0 0]  //  Communication cost between
tasks
               [ 0 0 0 0 0.2 0 0 0 0]
               [  0 0 0 0 0 0.2 0 0 0]
                [  0 0 0 0 0 0 0.2 0 0]
```

```
[  0 0 0 0 0.1 0.1 0 0 0]
 [  0 0 0 0 0 0 0 0.2 0]
  [  0 0 0 0 0 0 0 0 0.2]
[  0 0 0 0 0 0 0 0 0.1]
 [  0 0 0 0 0 0 0 0 0]];
```

No Constraints Model file:

```
/*********************************************
 * OPL 20.1.0.0 Model
 * Author: admin
 * Creation Date: 15-Jun-2021 at 5:14:15 pm
 *********************************************/
 //Indices
 int NoOfVertices = ...;
 {int} Vertices = {1,2,3,4,5,6,7,8,9};
 {string} EntryVertices = ...;
 {string} ExitVertices = ...;
 {string} Processors =...;



 //Parameters and Data
 int Edges[Vertices][Vertices] = ...;
 float Computation[Vertices] = ...;
 float Communication[Vertices][Vertices] = ...;



 //Decision Variables
 dvar boolean X[ Vertices][Processors];


 //Objective Function
 dexpr float ScheduleLength = sum(s in Vertices, c in Vertices)(Computation[s]
+ Communication[s][c]);

 //Model
 minimize ScheduleLength;

 subject to{
   forall( en in Vertices, p in Processors)
      X[en][p] == 1;
    }
```

Output:



b. The later part of the problem is solved by imposing the task execution constraints and the data dependency constraints on the above milp formulation. On solving, the output schedule length is found to be 39.4 ms which increased due to the increase in decision variables in the milp.

Working code:

Rp_data file:

```
/*********************************************
 * OPL 20.1.0.0 Data
 * Author: admin
 * Creation Date: 22-May-2021 at 9:35:36 am
 *********************************************/
 //Index
NoOfVertices = 9; //No of vertices taken
EntryVertices = {p};
ExitVertices = {x};
Processors = {a,b,c};
Computation = [ 0.1 0.2 0.1 0.1 0.2 0.1 0.1 0.2 0.1 ]; // computation cost
of tasks
Edges = [[ 0 1 1 1 0 0 0 0 0]  //  Interconnection between tasks.
                [ 0 0 0 0 1 0 0 0 0]
                [  0 0 0 0 0 1 0 0 0]
                 [  0 0 0 0 0 0 1 0 0]
                  [  0 0 0 0 1 1 0 0 0]
                   [  0 0 0 0 0 0 0 1 0]
                    [  0 0 0 0 0 0 0 0 1]
                     [  0 0 0 0 0 0 0 0 1]
                      [  0 0 0 0 0 0 0 0 0]];
Communication = [[ 0 0.1 0.1 0.1 0 0 0 0 0]  //  Communication cost between
tasks
                [ 0 0 0 0 0.2 0 0 0 0]
```

```
                [  0 0 0 0 0 0.2 0 0 0]
                 [  0 0 0 0 0 0 0.2 0 0]
                  [  0 0 0 0 0.1 0.1 0 0 0]
                   [  0 0 0 0 0 0 0 0.2 0]
                    [  0 0 0 0 0 0 0 0 0.2]
                     [  0 0 0 0 0 0 0 0 0.1]
                      [  0 0 0 0 0 0 0 0 0]];
StartVertices = {p};
Dependents = {q,r,s};
//Tasks_On_SameProcessor = [[ p 0 r 0 t 0 v 0 0]
//                              [ 0 q 0 0 0 u 0 0 x]
//                         [ 0 0 0 s 0 0 0 w 0]]; //  Tasks mapped to same
processor

Z = 1;
M = 10000;
```

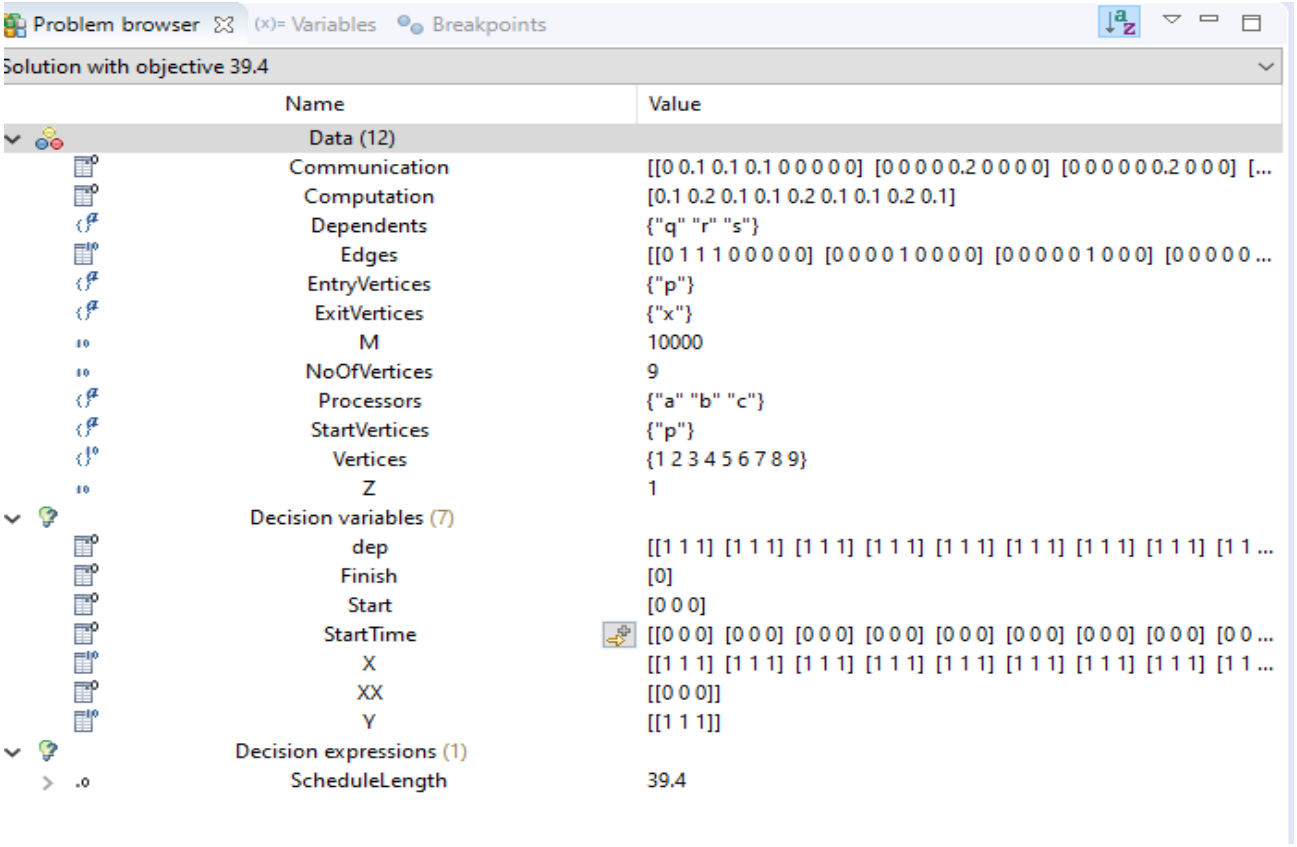
Rp_model file:

```
/*********************************************
 * OPL 20.1.0.0 Model
 * Author: admin
 * Creation Date: 22-May-2021 at 9:35:36 am
 *********************************************/

 // Optimal Duplication - Based Scheduling Problem in Multi Processor
Environment
/*********************************************/


 //Indices
 int NoOfVertices = ...;
 {int} Vertices = {1,2,3,4,5,6,7,8,9};
 {string} EntryVertices = ...;
 {string} ExitVertices = ...;
 {string} Processors =...;
 int M = ...;
 {string} StartVertices =...;
 {string} Dependents =...;
 //{string} Tasks_On_SameProcessor[Processors][Vertices] = ...;
// {string} Fin[Tasks_On_SameProcessor] = ...;
// {string} Starts[Tasks_On_SameProcessor] = ...;
 int Z = ...;

 //Parameters and Data
 int Edges[Vertices][Vertices] = ...;
 float Computation[Vertices] = ...;
 float Communication[Vertices][Vertices] = ...;


 //Decision Variables
 dvar boolean X[ Vertices][Processors];
 dvar boolean Y[ ExitVertices][Processors];
 dvar float+ Finish[StartVertices];
 dvar float+ Start[Dependents];
 dvar float+ StartTime[Vertices][Processors];
 dvar float+ XX[StartVertices][Dependents];
```

```
 dvar float+ dep[Vertices][Processors];


 //Objective Function
 dexpr float ScheduleLength = sum(s in Vertices, c in
Vertices)(Computation[s] + Communication[s][c])
  + sum(en in Vertices, p in Processors)((X[en][p] * dep[en][p]) +
(StartTime[en][p] * dep[en][p]) );

 //Model
 minimize ScheduleLength;

 //constraints
 subject to{

   forall( en in Vertices, p in Processors)
      X[en][p] == 1; //no of times entry process executes = 1


   forall(x in ExitVertices)
     sum(p in Processors)
      Y[x][p] >= 1; //non entry process can execute>1 times.


    forall(m in Vertices, k in Processors)
      StartTime[m][k] <= M*X[m][k];  //start time of redundant task limited
to 0


    forall(st in StartVertices, dp in Dependents)
       Finish[st] - Start[dp] <= M*(1 - XX[st][dp]); //finish time of
parent <= start time of child


       forall(st in StartVertices, e in Dependents)
       Start[e] <=  XX[st][e]; //dependency exists only if parent task is
scheduled


    forall(v in Vertices, p in Processors)
      dep[v][p] >= X[v][p]; //Redundant duplications will be removed as
unnecessary dependencies removed


    forall(st in StartVertices, dp in Dependents)
      XX[st][dp] == Start[dp]; //dependent depends only on one parent


    forall(st in StartVertices, dp in Dependents)
      Start[dp] <= M*(1 - XX[st][dp]);// child starts only  if dependency
exists, avoiding redundant executions


    forall(st in StartVertices, dp in Dependents)
       Finish[st]  - Start[dp] <= M*(1 - XX[st][dp]);//child should start
as soon as parent finishes execution
```

}

Output:



6. Future Works:

a. The MILP formulation is simplified with the model scale being decreased, where in contrary, many real life applications forms huge network of tasks in the schedule.
b. The existing heuristic algorithm, MDJ outperforms the proposed MILP method in terms to computation complexity. Therefore much emphasis in terms of computation complexity for the proposed problem should be taken.
c. To improve the performance, the MILP makes more task duplications on large platforms, whereas on the smallest platform, MILP makes less number of duplications.
d. Very less number of decision Variables are chosen in solving the MILP, which can increase the problem complexity. But these may increase as per the problem.

7. References:

1. Journal of Parallel and Distributed Computing, doi = {https://doi.org/10.1016/j.jpdc.2019.12.012}, URL = {https://www.sciencedirect.com/science/article/pii/S0743731518305653}.
2. CPLEX https://www.ibm.com/analytics/cplex-optimizer

3. H. Topcuoglu, S. Hariri and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," in IEEE Transactions on Parallel and Distributed Systems, vol. 13, no. 3, pp. 260-274, March 2002, doi: 10.1109/71.993206.
4. J. Singh, A. Gujral, H. Singh, J.U. Singh, N. Auluck, Energy aware scheduling on heterogeneous multiprocessors with DVFS and duplication, in: Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings, 2017, pp. 105–112.
5. J. Singh, A. Gujral, H. Singh, J.U. Singh, N. Auluck, Energy aware scheduling on heterogeneous multiprocessors with DVFS and duplication, in: Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings, 2017, pp. 105–112.
6. J. Mei, K. Li, Energy-aware scheduling algorithm with duplication on heterogenous computing systems, in: Proceedings - IEEE/ACM International Workshop on Grid Computing, 2012, pp. 122–129.
7. J. Mei, K. Li, Multi-copy deleting approach for duplication based scheduling on heterogeneous computing systems, in: Proceedings of the International Conference on Parallel Processing Workshops, 2012, pp. 610–613.
    A. Bender, MILP based task mapping for heterogeneous multiprocessor systems, in: European Design Automation Conference with EURO-VHDL
8. D. Bozdag, F. Ozguner, E. Ekici and U. Catalyurek, "A task duplication based scheduling algorithm using partial schedules," 2005 International Conference on Parallel Processing (ICPP'05), 2005, pp. 630-637, doi: 10.1109/ICPP.2005.15.

1. INTRODUCTION

The research paper addresses the task scheduling on a multiprocessor system with processor and precedence constraints. The optimal solution can be found in exponential time complexity therefore there is a need for a heuristic algorithm. Exhaustive list scheduling algorithm is efficient in finding optimal solutions but it is prone to generation of duplicate states..The two kinds of duplicate found in the ELS method are processor permutation duplicates and independent decision duplicates. Processor permutation duplicates refers to a condition in which the schedule differs only in the name of processors, having different Global schedule, this can be taken care of by processor normalisation in ELS method processor normalisation we normalise the names of the processor before adding the states in open list. This is primarily based on the assumption that processors are homogenous. Independent decision to allocate refers to a condition in which different local schedules give rise to the same Global schedule.The paper presents a new approach for solving this problem in two phases, first allocation and then ordering. This is abbreviated as the AO algorithm.

2. BACKGROUND AND RELATED WORK

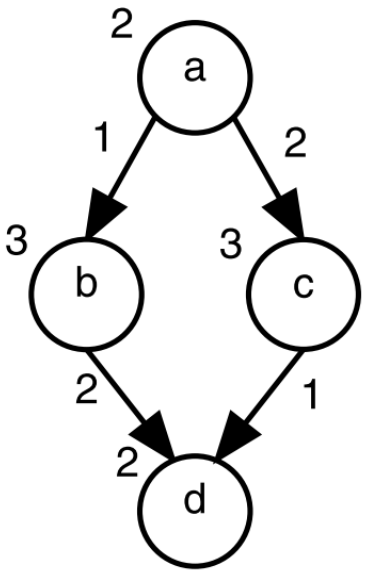
BASIC ASSUMPTIONS

The following classical assumptions are made about the target system

- Tasks are non-preemptive
- Local communication is cost-free
- The system is dedicated, i.e. no other program is executed while G is executed
- There is communication subsystem
- The processors are fully connected
- The Processors are identical.

TASK GRAPH

We schedule the tasks using DAGs. Representation G = {V, E, w, c}. G is a directed acyclic graph wherein each node n ∈ V represents a task, and each edge eij ∈ E represents a required communication from task ni to task nj, w represents a computation cost of a task and c represents the communication cost.Our aim is to produce a schedule S = {proc, ts}, where proc(n) allocates the task to a processor in P, and ts(n) assigns it a start time on this processor.



TASK SCHEDULING MODEL

Modified A* algorithm is used. A* algorithm is a best-first search algorithm in which we explore the node which has the best cost. A* algorithm is based on heuristic. This heuristic is based on the cost function which determines the optimal little of the algorithm, this cost function should be an underestimate of the cost incurred from moving to the goal state.The goal of optimal task scheduling is to find a schedule which has minimum execution time or minimum schedule length. There are two constraints which are related to scheduling processor constraints and President constraints. In processor constraints, we can have one task executing at a time in one processor, in precedence constraints we can schedule the task only if the parents have already been scheduled. There are two important aspects related to task scheduling: top level and bottom level. Top level refers to the cost from moving initial Node to current note.This does not include communication cost. bottom level refers to the cost of moving from current node to goal node excluding communication cost.These aspects are important in the perspective of calculating the cost function Which is responsible for heuristic involved in the algorithm.

RELATED WORK

As the nature of the problem is of exponential complexity so the majority of work has been done in approximation algorithms.Two best approximation algorithms are list scheduling and clustering.List scheduling consists of two stages. In the first stage tasks are ordered according to some priority, in the 2nd stage we schedule the task. In clustering algorithms we form clusters of the task using some similarities and then similar tasks are assigned to two same processors. While ELS is closer to list scheduling, AO is closer to clustering algorithms.

3. AO MODEL

In the first stage, we decide for each task the processor to which it will be assigned. We refer to this as the allocation phase. The method used has the additional benefit of not allowing the production of states which are isomorphic through processor permutation, eliminating the need for a relevant pruning technique. The second stage of the search, beginning after all tasks are allocated, decides the start time of each task. Given that each processor has a known set of tasks allocated to it, this is equivalent to deciding on an ordering for each set. Therefore, we refer to this as the ordering phase.The final step of this model is to combine these tasks into a single state-space.

ALLOCATION PHASE

In this phase the tasks are divided into several groups, this is similar to dividing a set into several subsets. After this the subsets of tasks are allocated to two processors randomly. Next there is a need of admissible heuristic for the cost function that gives the lower bound for the minimum length of schedule resulting from the partial partition of A at state s.The lower bound of a schedule depends on the computational load and communication load.

$$f_{\text{load}}(s) = \max_{a \in A} \left\{ min_{n \in a} tl_{\alpha}(n) + \sum_{n \in a} w(n) + min_{n \in a}(bl_{\alpha}(n) - w(n)) \right\}$$

The function for Communication cost is as follows:

$$f_{\text{acp}}(s) = \max_{n \in V'} \{tl_\alpha(n) + bl_\alpha(n)\}$$

Tighter bound is required, therefore the final f value is the maximum of computation load and communication cost.

$$f_{\text{alloc}}(s) = \max\{f_{\text{load}}(s), f_{\text{acp}}(s)\}$$

ORDERING PHASE

In this phase within a processor tasks are arranged. After arranging the task we get an estimate of the start time of the task. Our main intention is to calculate the estimated start time of the task. In this process maintains a local ready list inspired from list scheduling. This list maintains the task whose parents have already been scheduled.This process concentrates on local optimum which can be declared invalid by global optimum means the local schedule which the processors are generating is not accurate according to global schedule. The process for calculating heuristic cost is different from that of allocation phase as this takes into account the ideal time introduced and communication delays between processors and also between tasks within a processor due to precedence constraints. With respect to the ordering phase, two bounds have been decided. First bound consists of a partially scheduled paath which is a sum of estimated start time and allocated bottom-level.

$$f_{\text{scp}}(s) = \max_{n \in ordered(s)} \{eest(n) + bl_\alpha(n)\}$$

The second bound consists of the maximum finish time of the processor and the weight of unscheduled tasks.

$$f_{\text{ordered-load}}(s) = \max_{p \in P} \left\{ t_f(p) + \sum_{n \in p \cap unordered(s)} w(n) \right\}$$

To obtain the tightest possible bound, the maximum of these bounds is taken as the final f -value.

$$f_{\text{order}}(s) = \max\{f_{\text{scp}}(s), f_{\text{ordered-load}}(s)\}$$

4. PRUNING TECHNIQUES

Identical task: the tasks, which are identical,their order can be fixed. The fixed order is done using virtual edges. This helps in the reduction of search space. Identical tasks means they have the same weight ,communication cost, same parent and same child. In the AO algorithm, particularly in the allocation phase, the number of identical tasks allocated to a processor is known, This identification helps in easy ordering within the processor.

Heuristic schedules:heuristic schedule length is generated, which is used for upper bound pruning. Schedules having length greater than heuristic length are pruned by adding the heuristic schedule in the open list.

Fixed task order: In case of certain task graphs, the order of execution can be fixed. For example in the case of a fork graph the optimal schedule is generated by arranging the edges by non-decreasing order. If the graph structure contains both fork and join then what order should be followed. This is one of the most important questions to answer as a large number of graphs face this issue. Solution to this problem is to order the task in fork order and then verify it according to join order.

NOVEL PRUNING TECHNIQUES

GRAPH REVERSAL

After observing research papers and

experimenting, the researchers have come to a conclusion that join graphs are difficult to solve. In this paper a novel technique of graph reversal has been introduced. For solving a task graph which contains join substructure, the graph is reversed and we find the solution of the fork graph. This solution is then adapted according to the join graph. Reversal of a graph means reversing the edges of a graph. Reversing a schedule means the task which has been allocated first is now being allocated last. This technique can be generalized for in and out-tree as this contains fork and join graph as sub-structure.

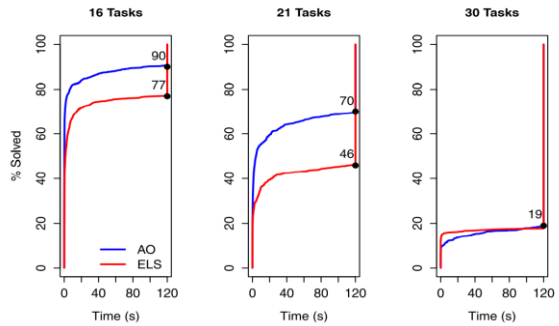
5. EVALUATION

A set of 1360 task graphs with unique combinations of these attributes were generated. These graphs were divided into four groups according to the number of tasks they contained: either 10 tasks, 16 tasks, 21 tasks, or 30 tasks.

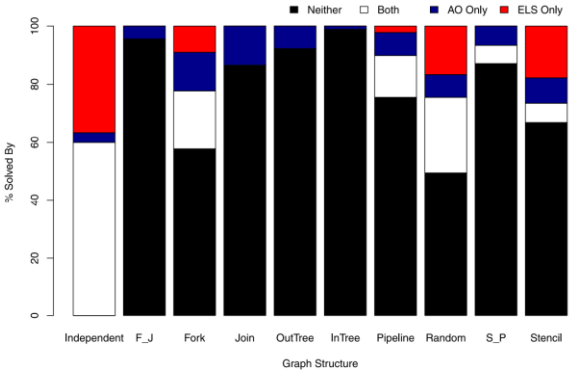
Range of task graphs in the experimental data set.

Graph structure	No. of tasks	CCR values
• Independent	• 16	• 0.1
• Fork	• 21	• 1
• Join	• 30	• 10
• Fork-Join		
• Out-Tree		
• In-Tree		
• Pipeline		
• Random		
• Series–Parallel		

An optimal schedule was attempted for each task graph using 2, 4, and 8 processors, once each for each state-space model. This makes a total of 4080-tasks.



Overall, AO has significantly better performance in these experiments, particularly in the ‘‘medium difficulty’’ 21 task group.



(c) 30 Tasks

In the 30 task group the performance was similar as the task graph is of complex nature.

6. CONCLUSION

The performance of the AO model is better than the performance of the ELS model in all categories of tasks observed. Hence, the duplicate free state-space model used can be of much more practical use than the ELS method.

CODE FOR ELS

```
#include <iostream>
#include <bits/stdc++.h>

using namespace std;

template<typename T>
class Graph{
   map<T,list<pair<T,int>>> adjlist;
   public: Graph(){

   }
   void addedge( T u, T v,int comm ,bool bidir=true){
      adjlist[u].push_back(make_pair(v,comm));

      if(bidir==true){
         adjlist[v].push_back(make_pair(u,comm));
      }
   }
   void print(){
      //Iterate over the map

      for(auto i:adjlist){
         cout<<i.first<<"->";
         //i.second is linked list
         for(auto entry :i.second){

            cout<<"("<<entry.first<<","<<entry.second<<")";
         }
         cout<<endl;
      }
   }



void ELS(){

/*OPEN ←sinit
while OPEN a!=ø do
s←headOf(OPEN)
if s complete state then
return optimal solution s
end-if
```

```
Expand s to new states NEW
for all si belongs to NEW do
Calculate f(si)
Insert si into OPEN, unless duplicate in CLOSED or OPEN
end-for
CLOSED ←CLOSED + s; OPEN←OPEN - s
end-while
*/



    }

int heuristic(){

// max. of idle-time, bottom-level, data-ready time

// calculating the computational weightage for idle-time calculation
// use traversal technique to calculate--complete it using BFS

queue<int> q;
// Assume starting node equal to one
int src=1;
map<int,bool> visited;
map<int,int> parent;

q.push(src);
visited[src]=true;
parent[src]=0;
long long int total_comp=0;

for(int i=1;i<=n;i++){
    total_comp +=computation_cost[i];
}



while(!q.empty()){
    int node=q.front();
    cout<<node<<" ";
    q.pop();


    // Neighbors of current node
```

```
    for(int Neighbour:adjlist[node]){

        if(!visited[Neighbour]){
            q.push(Neighbour);
            visited[Neighbour]=true;
            parent[Neighbour]=node;
        }
    }
}

//for calculation of bottom we from current node to goal node recursively with data-ready time.




}

};

class state{
        // This contains the list of the task associated with each of the processor
    public:
        int proc_list[11][11];
    };

    // This contains the open and closed list used in the implementation of ELS
    vector<state> open;
    vector<state> closed;

bool dulplicates(state s1,state s2){

// Before checking for duplicates we need to normalize both the states .
// This normalising is done by sorting individual processors according to no. of task




//checking same entries
        for(int i=1;i<=10;i++){
            for(int j=1;j<=10;i++){
                if(s1.proc_list[i][j]!=s2.proc_list[i][j]){
                    return false;
```

```
            }

          }
        }
        return true;

}


// Functor for comparision in fork graph maintaining ascending order

bool mycompare(pair<int,int> a, pair<int,int> b){

if(a.second<b.second){
    return true;
}
return false;
}

// Functor for maintaining descending order in join graph

bool mycompare2(pair<int,int> a, pair<int,int> b){

    if(a.second>b.second){
        return true;
    }
    return false;
}




void pruning(){

    //fixed task order
    //equivalent states
    //heuristic
    //identical task scheduling--maintain a parent array while traversing.

    // if task graph is fork
    sort(adjlist[src],adjlist[src]+n-1,mycompare);

    // if graph is join
```

```
    sort(adjlist[src],adjlist[n-1],mycompare2);




}

map<int,int> computation_cost;
 int no_of_processors;

int main() {

     Graph<int> g;
    g.addedge(1,2,1);
    g.addedge(1,3,4);
    g.addedge(2,3,1);
    g.addedge(3,4,2);
    g.addedge(1,4,7);
    g.print();

    int n;
    cout<<"enter the no. of individual nodes";
    cin>>n;


    for(int i=1;i<=n;i++){
        int p;
        cin>>p;
        computation_cost[i]=p;
    }

    /* Another representation of the task graph using matrix
    int n,comm,comp;
    pair<int,int> task_graph[n+1][n+1];
    cout<<"enter the no. of task"<<endl;
    for(int i=1;i<=n;i++ ){
        for(int j=1;j<=n;j++){
            //Enter the computation cost and the communication cost
            cin>>comp>>comm;
            task_graph[i][j]=make_pair(comp,comm);
        }
    }
```

```
    */



    cout<<"Enter the no. of processors "<<endl;
    cin>>no_of_processors;


    // Finding no. of free tasks



    return 0;
}
```

CODE FOR AO

```
#include <iostream>
#include <bits/stdc++.h>

using namespace std;

template<typename T>
class Graph{
    map<T,list<pair<T,int>>> adjlist;
    public: Graph(){

    }
    void addedge( T u, T v,int comm ,bool bidir=true){
        adjlist[u].push_back(make_pair(v,comm));

        if(bidir==true){
            adjlist[v].push_back(make_pair(u,comm));
        }
    }
    void print(){
        //Iterate over the map

        for(auto i:adjlist){
            cout<<i.first<<"->";
            //i.second is linked list
            for(auto entry :i.second){
```

```
            cout<<"("<<entry.first<<","<<entry.second<<")";
        }
        cout<<endl;
    }
}
void AO(){



}

int heuristic(){



}

};

// Graph reserval novel pruning technique -- modification still needed
class reserval {
public:
    void eventualSafeNodes(vector<vector<int>>& graph){
        int n=graph.size();
        unordered_map<int,vector<int>> next;
        unordered_map<int,int> odegree;
        unordered_map<int,bool> visited;
        for (int i=0;i<n;++i){
            odegree[i]=graph[i].size();
            visited[i]=false;
            for (auto j:graph[i]) next[j].push_back(i);
        }

    }

    vector<int> topological_sort(vector<vector<int>>& graph){
            vector<int> ans;
            while (true){
            bool flag=false;
            for (int i=0;i<n;++i)
                if (odegree[i]==0 && !visited[i]){
                    flag=true;
                    visited[i]=true;
                    ans.push_back(i);
```

```
                for (auto j:next[i]) --odegree[j];
            }
        if (!flag) break;
    }
    sort(ans.begin(),ans.end());
    return ans;

  }
};

// Finding the critical path in the directed graph
vector<int> order;

void topo(int src,vector<int> &vis,vector<vector<pair<int,int> > > g){
   vis[src] = 1;
   for(auto x:g[src]){
      if(!vis[x.first])
         topo(x.first,vis,g);
   }
   order.push_back(src);
}

void critical_path(){
 vector<int> vis(v,0);
   for(int i=0;i<v;i++){
      if(!vis[i]){
         topo(i,vis,g);
      }
   }
   vector<int> dist(v);
   for(int i=0;i<v;i++) dist[i] = INT_MIN;
   dist[src] = 0;
   for(int i=v-1;i>=0;i--){
      if(dist[order[i]]!=INT_MIN){
         for(auto x:g[order[i]]){
            int v = dist[x.first];
            int w = x.second;
            int u = dist[order[i]];
            if(u + w > v)
               dist[x.first] = u + w;
         }
      }
   }
   for(int i=0;i<v;i++){
```

```
        if(i!=src and dist[i]!=INT_MIN){
            cout<<src<<" -> "<<i<<" = "<<dist[i]<<endl;
        }
    }

}



map<int,int> computation_cost;
 int no_of_processors;


int main() {

     Graph<int> g;
    g.addedge(1,2,1);
    g.addedge(1,3,4);
    g.addedge(2,3,1);
    g.addedge(3,4,2);
    g.addedge(1,4,7);
    g.print();

    int n;
    cout<<"enter the no. of individual nodes";
    cin>>n;


    for(int i=1;i<=n;i++){
        int p;
        cin>>p;
        computation_cost[i]=p;
    }

    /* Another representation of the task graph using matrix
    int n,comm,comp;
    pair<int,int> task_graph[n+1][n+1];
    cout<<"enter the no. of task"<<endl;
    for(int i=1;i<=n;i++ ){
        for(int j=1;j<=n;j++){
            //Enter the computation cost and the communication cost
            cin>>comp>>comm;
            task_graph[i][j]=make_pair(comp,comm);
        }
```

```
    }
    */



    cout<<"Enter the no. of processors "<<endl;
    cin>>no_of_processors;


    // Finding no. of free tasks



    return 0;
}
```