##### RELIABLE TASK SCHEDULING IN HETEROGENEOUS DISTRIBUTED SYSTEMS

**A PROJECT REPORT**

***Submitted by***

|  |  |
| --- | --- |
| **BHIRUNDHA . M** | **(Reg.No. 201306006)** |
| **RAJA PRIYA . P** | **(Reg.No. 201306035)** |
| **SRI SRUTHI . S** | **(Reg.No. 201306054)** |

***in partial fulfillment for the award of the degree***

***of***

##### BACHELOR OF TECHNOLOGY

in

**INFORMATION TECHNOLOGY**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**MEPCO SCHLENK ENGINEERING COLLEGE, SIVAKASI**

**(An Autonomous Institution affiliated to Anna University Chennai)**

**April 2017**

**BONAFIDE CERTIFICATE**

##### Certified that this project report titled RELIABLE TASK SCHEDULING IN HETEROGENEOUS DISTRIBUTED SYSTEMS is the bonafide work of M.BHIRUNDHA, P.RAJA PRIYA and S.SRI SRUTHI, who carried out the research under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

**Mrs. S. Saroja, M. E., Dr. T. Revathi, M. E., Ph. D.,**

**Internal guide Head of the Department**

**Assistant Professor (Senior Grade), Senior Professor,**

**Department of Information Technology, Department of Information Technology,**

**Mepco Schlenk Engineering College, Mepco Schlenk Engineering College,**

**Sivakasi. Sivakasi.**

Submitted for viva-voce Examination held at **MEPCO SCHLENK ENGINEERING COLLEGE, SIVAKASI (AUTONOMOUS)** on **……………..............**

**Internal Examiner External Examiner**

**ACKNOWLEDGEMENT**

First and foremost, we thank the almighty for showering grace and blessings on us for making this project a great success.

We extend our sincere gratitude to our college management and our honorable Principal **Dr. S. ARIVAZHAGAN, M.E., Ph.D.,** for providing sufficient working environment such as systems and library facilities. We also thank him very much for providing us adequate lab facilities, which enable us to complete our project.

We would like to extend our heartfelt gratitude to **Dr. T. REVATHI, M.E, Ph. D,** Professor and Head, Department of Information Technology, Mepco Schlenk Engineering College for her valuable guidance and timely encouragements. She put all her valuable experience and expertise in directing, suggesting and supporting us throughout the project to bring out of the best.

We would like to extend our gratitude to **Mrs. S. SAROJA, M. E.,** Assistant Professor (Senior Grade), Department of Information Technology, Mepco Schlenk Engineering College for being our Project Coordinator and directing us through our project.

We extend our thanks to all staff members and lab technicians for their moral support in completing our project.

ABSTRACT

**ABSTRACT**

Task Scheduling is one of the major challenges in efficiently using parallel and distributed systems. Even though those systems can be easily established, quality of service including reliability, performance and throughput is of major concern. In this paper we have introduced a set of algorithms which schedules the task by considering fault recovery in heterogeneous distributed systems to increase the performance and reliability. Here we have two phases in the first phase is to obtain the favorable number of allotments for each tasks and second the second phase has the scheduling method. And to reduce the expected execution time we have replicated the tasks based upon two strategies. In the first strategy the tasks are replicated to its allotment, its replication is never stopped and in the second strategy the replication of the tasks are stopped once the task is completed. Since recovery of failed machines increases the processing time, this scheduling helps to reduces the processing time by replicating the tasks instead of recovering the machines.

LIST OF SYMBOLS

**LIST OF SYMBOLS**

|  |  |
| --- | --- |
| **Notation** | **Description** |
| *P , L* | A set of processors and its links. |
| *pi ,li,j* | A processor and a communication link between pi and pj. |
| *w(pi)* | Weight of *pi* (Number of operations per unit time). |
| *w(li,j)* | Weight of *li,j* (Data transfer rate). |
| *V , E* | A set of tasks and precedence constraints among the tasks. |
| *w(vi)* | Weight of task *vi* (Computational Cost). |
| *w(ei,j)* | Weight of edge *ei,j* (Communication Cost). |
| *Si* | Set of processors allocated to vi. |
| *tp(vi,px)* | Processing time of vi on processor px. |
| *tc(ei,j,lk,x)* | Communication time of ei,j on link lk,x. |
| *pred(vi)* | Set of direct predecessors of task vi. |
| *succ(vi)* | Set of direct successors of task vi. |
| *λi* | Failure rate of processor pi |
| *λli,j* | Failure rate of link li,j |
| *γi* | Probability that a failure on processor pi is recoverable. |
| *R[Evi,px,pk,px]* | Communication failure probability. |
| ***Notation*** | **Description** |
| *R[Evi,px,pk,px]* | Communication reliability probability. |
| *F[Evi,px]* | Failure probability of task vi on processor px |
| *R[Evi,px]* | Reliability probability of task vi on processor px |
| *[Evi,px]* | Expected overhead of task vi on processor px |
| *p[vi]* | Mean processing time of task vi |
| *c[ei,j]* | Mean communication time of edge ei,j |
| *[Evi]* | Mean reliability of task vi |
| *[Eei,j]* | Mean communication reliability probability between vi and vj |
| *[Evi]* | Mean overhead of task vi |
| *Fi* | Mean failed probability of task vi |
| *Tp(vi, αi)* | Expected processing time of task vithat executes on αiprocessors. |
| *W(vi,αi)* | Expected work of task vithat executes on αiprocessors |
| *AL(i)* | Lower bound on the number of allotted processors for task vi |
| *β* | Weight of reliability overhead |
| *EST(vi,pk)* | Earliest start time of task vi on a processor pk |
| *EFT(vi,pk,Si)* | Earliest finish time of task vi on processor pk |
| *EFT(ei,j,Si,px)* | Earliest communication finish time of edge ei,j |

**LIST OF FIGURES**

**LIST OF FIGURES**

|  |  |
| --- | --- |
| **Fig No** | **Fig Name** |
| 1.1 | Input window of Task Graph Generator |
| 1.2 | Topology of the randomly generated graph |
| 1.3 | Output File 1 of TGG |
| 1.4 | Output File 2 of TGG |
| 4.1 | Architecture of the System |
| 4.2 | WorkFlow of the project |
| 5.1 | Output of Strategy 1(1) |
| 5.2 | Output of Strategy 1(2) |
| 5.3 | Output of Strategy 2(1) |
| 5.4 | Output of Strategy 2(2) |

CONTENTS

|  |  |  |  |
| --- | --- | --- | --- |
| **TABLE OF CONTENTS** | | | |
| **CONTENTS** | |  | **PAGE NO** |
| **Bonafide certificate** | |  | ii |
| **Acknowledgement** | |  | iii |
| **Abstract** | |  | iv |
| **List of Symbols** | |  | vi |
| **List of Figures** | |  | ix |
| **CHAPTER 1** |  | **INTRODUCTION** |  |
|  |  |  |  |
|  | 1.1 | PROJECT OBJECTIVE | 2 |
|  | 1.2  1.3  1.4  1.5 | PURPOSE  SCOPE  REQUIREMENTS  SUMMARY | 2 |
| **CHAPTER 2** |  | **LITERATURE REVIEW** |  |
|  |  |  |  |
|  | 2.1 | OVERVIEW | 4 |
|  | 2.2 | DYNAMIC LEVEL SCHEDULING ALGORITHM | 4 |
|  | 2.3 | LEVELIZED MIN TIME ALGORITHM | 4 |
|  | 2.4 | HETEROGENEOUS EARLIEST FINISH TIME ALGORITHM | 5 |
|  | 2.5  2.6 | CRITICAL PATH ON A MACHINE  RELIABILITY AWARE SCHEDULING ALGORITHM | 6 |
| **CHAPTER 3** |  | **SYSTEM STUDY** |  |
|  |  |  |  |
|  | 3.1 | OVERVIEW | 8 |
|  | 3.2 | PROPOSED SYSTEM | 8 |
|  |  | 3.2.1 WORK FLOW |  |
|  |  | 3.2.2 INPUTS AND OUTPUTS | 8 |
|  | 3.3 | SUMMARY | 8 |
|  |  |  |  |
| **CHAPTER 4** |  | **SYSTEM DESIGN** |  |
|  |  |  |  |
|  | 4.1 | OVERVIEW | 10 |
|  | 4.2 | SYSTEM ARCHITECTURE |  |
|  | 4.3  4.4 | 4.2.1 DESCRIPTION  PROJECT WORKFLOW  SUMMARY | 11 |
| **CHAPTER 5** |  | **IMPLEMENTATION METHODOLOGY** |  |
|  |  |  |  |
|  | 5.1 | PREREQUISITES | 13 |
|  | 5.2 | ALGORITHMS | 13 |
|  |  | 5.2.1 ALLOTMENT LOWER BOUND | 13 |
|  |  | 5.2.2 THE ALLOTMENT AWARE SCHEDULING | 14 |
|  |  | 5.2.3 THE EXPECTED TIME PROCEDURE | 15 |
|  | 5.3 | CODING | 18 |
|  | 5.4 | SCREENSHOT | 44 |
| **CHAPTER 6** |  | **CONCLUSION** |  |
|  |  |  |  |
| **REFERENCES** |  |  | 49 |
|  |  |  |  |
|  |  |  |  |

**INTRODUCTION**

1. **INTRODUCTION**

As technology advances, in recent days more powerful computers are needed to efficiently perform certain tasks. Hence developing task scheduling algorithms has been proved to be NP-Complete. Recovering tasks when it fails may extend the make span of the processor. So replication mechanisms were followed to decrease the make span of the processor. The active replication of task and data communication is a common means of increasing the reliability of a system. Intuitively, adding replications increases not only reliability but also, in general, the schedule length.

**1.1 PROJECT OBJECTIVE**

The objective of the project is to schedule tasks reliably by considering fault recovery (System continuing to be in consistent state though failure occurs) on heterogeneous distributed systems. The expected make span of the system has also to be reduced.

**1.2 PURPOSE**

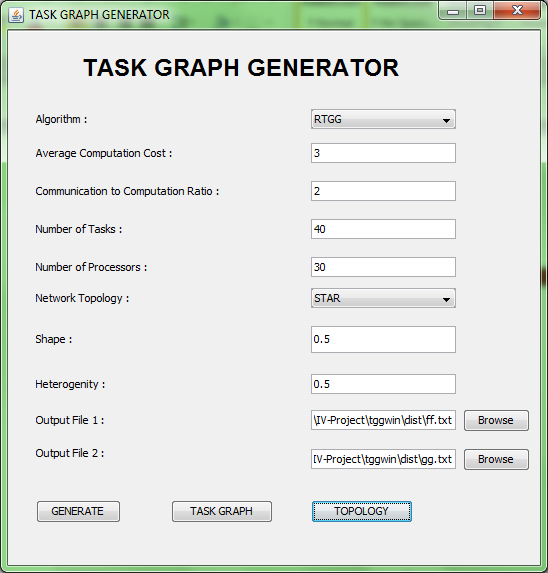
The purpose of the project is to explore the failure and its recovery in heterogeneous distributed systems. Since failure causes a lot of delay in completing the tasks assigned to processor, this project concentrates on decreasing the make span.

**1.3 SCOPE**

The scope of the project is to efficiently schedule the tasks and decrease the make span of the system. This can be done by replicating the tasks in a wise manner by following certain algorithms discussed below.

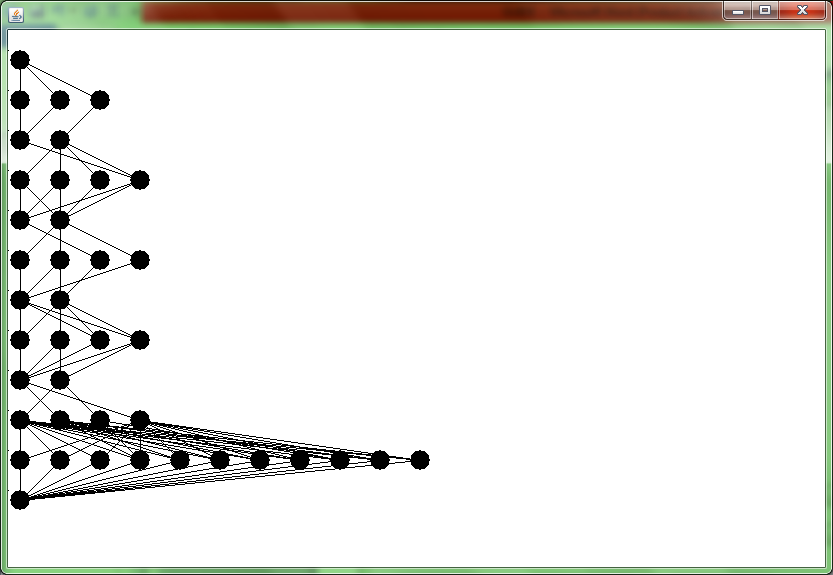
**1.4 REQUIREMENTS**

One of the main requirements is the TASK GRAPH GENERATOR (TGG) which generates Task Graph based on certain algorithms like RTGG, FFT, etc. On giving input as certain parameters (No. of tasks, No. of processors, Topology, etc.,), the TGG will give two output files.



**Fig. 1.1 Input window of Task Graph Generator**

The first file contains number of tasks and number of edges in its first and second lines respectively. It is followed by a tabular form of task and it’s adjacent with communication between them.



**Fig 1.2 Topology of the randomly generated graph**

**Fig. 1.3 Output File 1**

The second file contains the number of processors followed by a matrix like representation which shows the processing time of each task on each processor.

**Fig. 1.4 Output File 2**

**1.5 SUMMARY**

The rest of the document is organized as follows:

Section 2 briefly describes the related works and existing methods that have been performed for the proposed method.

Section 3 briefly describes the Workflow, Inputs and Outputs for each phase of the system.

Section 4 briefly describes the system architecture and flow of the system.

Section 5 briefly describes the implementation details of the system.

Section 6 briefly describes the results of the processes used in this system.

Section 7 contains conclusion and further enhancement.

**LITERATURE REVIEW**

1. **LITERATURE REVIEW**
   1. **OVERVIEW**

This chapter clearly explains about the existing system for the proposed algorithm in the project. The section 2.2 deals with dynamic level scheduling algorithm and section 2.3 deals with Levelized Min Time Algorithm. The section 2.4 deals with HEFT algorithm and section 2.5 deals with CPOP. The section 2.6 deals with Reliability Aware Scheduling Algorithm.

* 1. **DYNAMIC LEVEL SCHEDULING ALGORITHM**

Dynamic-level scheduling is an effective compile-time scheduling technique which is used for inter processor communication overhead when mapping precedence-constrained, communicating tasks onto arbitrarily interconnected processor networks. Scheduling and routing are performed simultaneously to account for limited interconnections between processors, and communications are scheduled along with computations to eliminate shared-resource contention. It encompasses heterogeneous processing environments, and presents two techniques designed to enhance scheduling performance: forward/backward scheduling, and precedence constraint appendage.

* 1. **LEVELIZED MIN TIME ALGORITHM**

The LMT algorithm is a two phase scheduling algorithm. The first phase groups the tasks that can be executed in parallel using the level attribute. The second phase is a greedy method that assigns each task to the fastest available processor. A task at a lower level has higher priority for scheduling than a task at a higher level. Within the same level, the task with the highest average computation cost has the highest priority. If the number of tasks in a level is greater than the number of available processors, the fine-grain task, arc merged into a coarse-grain task until the number of tasks is equal to the number of processors. Then the tasks are sorted in reverse order (largest task first) based on average computation time. Beginning from the largest task, each task will be assigned to the processor that minimizes the sum of computation cost of the task and the communication costs with tasks in the previous layers and does not have any scheduled task at the same level.

* 1. **HETEROGENEOUS EARLIEST-FINISH-TIME ALGORITHM**

HEFT is a simple and best scheduling technique in static task scheduling in heterogeneous as well as homogeneous environment for limited number of processors. HEFT has two stages:

* Prioritization phase
* Processor selection stage.

HEFT calculates the priority using upward ranking (ranku). An application is traversed in upward direction and the rank of all nodes in a list with the help of mean communication and mean computation cost are found. Generated list is arranged in decreasing order of ranku. HEFT uses a Tie breaking policy for selecting the nodes, which node or successor selects whose rank value is highest. Upward rank of task ni is described as:

Rank(ni) = Wi+maxnjϵsucc(ni) (cij+ranku(nj))

where Wi is the mean computation cost, succ (ni) is the immediate child of node ni, ci,j is the mean communication cost of node (i,j). In case of two nodes have equal rank value selects randomly. In upward ranking, graph is traversed from exit node to entry node. Highest rank is same with exit node:

ranku(nexit) = W exit ranku(ni)

is total critical path from source node to exit node including communication and computation cost of tasks.

* 1. **CRITICAL PATH ON A MACHINE**

CPOP computes the rank value of each node by adding both the techniques ranku+ rankd. An application is traversed from entry node (ni) to exit node (nj) is called downward ranking and traverse exit node to entry node is called upward ranking. CPOP has two steps:

* Task prioritization phase
* Task allocation phase.

**Task Prioritization Phase**

Tasks are prioritized according to their rank value (ranku+ rankd) with the help of communication and computation costs of DAG then set into a queue (decreasing order). CPOP used critical path (CP) of an application to find the longest path starting from entry node to exit node.

**Task allocation Phase**

Tasks are selected according to higher rank value and selects for scheduling to best suitable processor which minimizes the execution time of task. CP nodes are scheduled on a processor which has less mean computation cost then other processors.

* 1. **RELIABILITY AWARE SCHEDULING ALGORITHM**

Reliability Aware Scheduling Algorithm effectively measures system reliability, based on an optimal reliability communication path search algorithm. Reliability priority rank (RRank) is used to estimate the task's priority by considering reliability overheads. Based on directed acyclic graph (DAG) a reliability-aware scheduling algorithm for precedence constrained tasks, which can achieve high quality of reliability for application is proposed.

**SYSTEM STUDY**

1. **SYSTEM STUDY**
   1. **OVERVIEW**

This section discusses the overall flow of the proposed system. Proposed system describes a way to schedule tasks to processor with minimum make span. This is done by Heterogeneous Allotment Aware Scheduling which schedules tasks to processors based on RRank.

* 1. **PROPOSED SYSTEM**

The input to the system is a set of files which contains communication cost between adjacent tasks and processing time of tasks on processors. The proposed algorithm is a two-phase algorithm. The ﬁrst phase ﬁnds a favourable number of replication of tasks. The second phase applies a scheduling method that is based on the expected executed time and the communication time.

* + 1. **WORK FLOW**

The work flow of the system proceeds as follows:

* The Task Graph Generator accepts inputs as certain parameters like No. of tasks, no. of processors, topology etc., and produces two files as output which contains communication cost between adjacent tasks and processing time of each task on processor.
* With the output files generated by TGG, the data enters phase I which is **Allotment Lower Bound.**
* Then allotment upper bound is found.
* Then the data enters phase II which is **Allotment Aware Scheduling.**

Finally expected time is calculated using certain rounding procedures.

* + 1. **INPUTS AND OUTPUTS**

**Phase I**

|  |  |
| --- | --- |
| **Input** | Mean processing time  Mean recovery time  Function with arguments task and no. of processors |
| **Output** | Allotment lower bound for each task i.e., minimum number of replications for each task |

**Phase II**

|  |  |
| --- | --- |
| **Input** | List of tasks  List of processors |
| **Output** | Tasks scheduled to processors |

**3.3. SUMMARY**

This chapter describes the entire workflow of our system. Inputs, outputs of each and every phase are described in detail. The overall functions of the system are studied.

SYSTEM DESIGN

1. **SYSTEM DESIGN**

**4.1 OVERVIEW**

This section discuss about the design of our system. The modules of the system are described in detail. The various steps involved in each module are specified.

* 1. **SYSTEM ARCHITECTURE**

Finding lower and upper bound

Calculation of expected time

Scheduling Tasks

TG

Generation

**Fig 4.1 Architecture of the System**

* + 1. **DESCRIPTION**

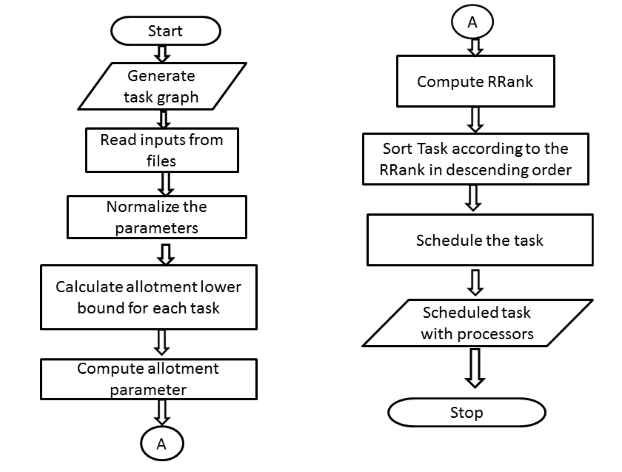
**Task Graph Generation:** Task graph is generated using certain parameters like no. of edges, no. of tasks, no. of processors, topology, algorithm etc. Then two output files are generated containing communication cost between adjacent tasks and processing time of each task on processors. Certain parameters like no. of edges, no. of tasks, no. of processors, communication costs between adjacent tasks, precedence tasks, processing time of each task on processors are read from the file.

**Finding lower and upper bound:** In this phase, an algorithm is proposed which determines the allotment lower and upper bound i.e., the least number of replications a task can have and the maximum number of replications a task can have. Allotment parameter is used.

**Scheduling Tasks:** In this phase, normalization has to be done to convert the heterogeneous distributed system into homogeneous distributed system. Then RRank is calculated, based on RRank tasks are scheduled in decreasing order.

**Calculation of expected time:** In this last phase, expected time i.e., the computation time for each task is calculated.

* 1. **PROJECT WORKFLOW**



**Fig. 4.2 Work Flow of the project**

* 1. **SUMMARY**

This section discussed the entire design of our system. The modules used and their functionalities are described in detail.

**IMPLEMENTATION METHODOLOGY**

1. **IMPLEMENTATION METHODOLOGY**
   1. **PREREQUISITES**

The input for these algorithms are in the form of graph. Hence with the help of Task Graph Generator tool we should generate the graphs, which gives us two output files. One file contains the task graph and its adjacency relationship and the other file contains processor graph having execution time of each task in every processor.

* 1. **ALGORITHMS**

**THE ALLOTMENT LOWER BOUND**

**In the Strategy I**

AL(*i*) = 1 for all 1  *i n*

**In the Strategy II**

for *i* 1 to n do

if f1(vi ,1) then

AL(*i*) = 1

else if < f1(vi ,m - 1) then

AL(*i*) = m

else

Find a number *j* ϵ [1, m – 2] such that

f1(vi , j + 1) <= < f1(vi , j)

AL(i) = j + 1

Algorithm 1 is used to find the allotment lower bound which is the minimum number of replications for the tasks to be done. For Strategy I it is always 1. For Strategy II by considering the communication time the lower bound varies.

* + 1. **THE ALLOTMENT AWARE SCHEDULING**

1. Normalize the heterogeneous system into a homogeneous system.
2. Compute allotment parameter α’ by the linear program and the rounding procedure.
3. Compute αi  = max {min {α’i , µ}, AL (i)} for vi ϵ V.
4. Compute RRank for all tasks by traversing graph from the exit task.
5. Sort the tasks in a scheduling list by non – increasing order of RRank
6. while the scheduling list is not empty do
   1. Remove the first task vi from the scheduling list and set Si = Ø and CT = ∞.
   2. while αi > |Si | do
      1. Set Lmin = ∞ and pc = null
      2. for each pk ϵ P \ Si do

11. if Lmin > L(vi , pk , Si ) and CT > EFT (vi , pk , Si )

then

12. Set pc = pk and Lmin = L(vi ,pk , Si )

* + 1. If pc ≠ null then

14. Si = Si U {pc }

15.CT=

16. Else

No replication of task improve the make span, break the loop.

* + 1. **THE EXPECTED TIME PROCEDURE**

1. Compute the reliability probability R = e-λtp(v,p)  for the task v on the processor p and set t = tp(v,p).
2. Get a number n ε [0,1] uniformly at random.
3. While n > R do
4. Get a number p ε [0,1] uniformly at random
5. t = t+p \* tp(v,p) + 𝞱
6. Get a number n ε [0,1] uniformly at random.
   1. **CODING**

**For Strategy I**

package projects1;

import java.io.\*;

import java.text.DecimalFormat;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashMap;

import java.util.Iterator;

import java.util.LinkedHashMap;

import java.util.LinkedHashSet;

import java.util.LinkedList;

import java.util.List;

import java.util.Map;

import java.util.Set;

public class ProjectS1 {

static int [] task;

static int noOfProcessor,noOfTask,noOfEdges;

static int[][] proTime;

static double [] failureRate;

static double [] meanProTime;

static double meanRecTime;

static double [] recTime;

static double [][] proCommTime;

static double [] x;

static double beta;

static double [] gamma;

static double meanGamma;

static double [] alphadash;

static double [] w;

static Set<Integer>[] Si;

static Set<Integer>[] SSj;

static double[][] commfailureRate;

static Set<Integer> pk=new LinkedHashSet<>();

static Set<Double> newset=new LinkedHashSet<>();

static LinkedList<Node>[] list;

static ArrayList<Predecessor> predList;

static double allotpar;

static int[] ai;

static int[] rank;

static double[] avaipro;

static int max;

static double Ct;

static double lMin;

static int pc;

static double fin=0;

static int resrbar;

public static void main(String[] args) throws IOException {

**/\*Reading data from the file**

**\*/**

Predecessor pred;

DataInputStream in = null;

DecimalFormat newFormat = new DecimalFormat("#.#");

int avgProTime;

int avgCommTime;

int i,j;

int count=0;

int sum=0;

double ro=0.9999;

double sumf=0.0,meanValue;

int[] aL;

double m1;

String c;

String[] str;

j=1;

try {

in = new DataInputStream(new FileInputStream("C:\\Users\\DELL\\Documents\\IV-Project\\tggwin\\dist\\f1.txt"));

noOfTask=Integer.parseInt(in.readLine());

noOfEdges=Integer.parseInt(in.readLine());

System.out.println("No of Task:"+noOfTask);

System.out.println("No of Edges:"+noOfEdges);

meanProTime=new double[noOfTask+1];

task = new int[noOfTask+1];

predList=new ArrayList();

for(i=1;i<=noOfTask;i++)

task[i]=i;

list = new LinkedList[noOfTask+1];

aL=new int[noOfTask+1];

for(i=1;i<=noOfTask;i++){

list[i]=new LinkedList<>();

}

in.readLine();

c=in.readLine();

while (c!=null) {

str=c.split("\t");

int index;

index=Integer.parseInt(str[0]);

Node node=new Node(Integer.parseInt(str[1]),Integer.parseInt(str[2]));

list[index].add(node);

pred=new Predecessor(Integer.parseInt(str[1]),Integer.parseInt(str[0]),Integer.parseInt(str[2]));

predList.add(pred);

c=in.readLine();

}

}

finally

{

if (in != null)

{

in.close();

}

}

try {

in = new DataInputStream(new FileInputStream("C:\\Users\\DELL\\Documents\\IV-Project\\tggwin\\dist\\f2.txt"));

in.readLine();

noOfProcessor=Integer.parseInt(in.readLine());

System.out.println("No of Processor:"+noOfProcessor);

proTime=new int[noOfTask+1][noOfProcessor+1];

commfailureRate=new double[noOfProcessor+1][noOfProcessor+1];

failureRate=new double[noOfProcessor+1];

gamma=new double[noOfProcessor+1];

in.readLine();

c=in.readLine();

i=1;

while (c!=null) {

str=c.split("\t");

for(int k=0;k<str.length;k++,j++){

proTime[i][j]=Integer.parseInt(str[k]);

}

j=1;

i++;

c=in.readLine();

}

}

finally {

if (in != null) {

in.close();

}

}

proCommTime = new double[noOfProcessor+1][noOfProcessor+1];

for(int i1=1;i1<=noOfProcessor;i1++)

{

for(int j1=i1;j1<=noOfProcessor;j1++)

{

if(i1==j1)

{

proCommTime[i1][j1]=0;

}

else

{

proCommTime[j1][i1]=proCommTime[i1][j1]=Math.random()\*(1.25-0.75)+0.75;

}

}

}

m1=noOfProcessor;

allotpar=((m1\*(2.0+ro))-(Math.sqrt(((2.0+(2.0\*ro)+(ro\*ro))\*(m1\*m1))-((2.0\*m1)\*(1+ro)))))/2.0;

System.out.println("Allotment Paramter : "+allotpar);

for(int ii=1;ii<=noOfTask;ii++){

for(int jj=1;jj<=noOfProcessor;jj++){

sum+=proTime[ii][jj];

}

meanProTime[ii]=sum/noOfProcessor;

sum=0;

}

w=new double[noOfProcessor+1];

recTime=new double[noOfProcessor+1];

for(i=1;i<=noOfProcessor;i++)

{

w[i]=(double) Math.random()\*(1.25-0.75)+0.75;

recTime[i]=1/w[i];

sumf+=recTime[i];

}

meanRecTime=sumf/noOfProcessor;

System.out.println("Mean Recovery Time "+meanRecTime);

for(i=1;i<=noOfProcessor;i++)

{

failureRate[i]=Math.random()\*(0.01-0.001)+0.001;

}

for(i=1;i<=noOfProcessor;i++)

{

for(j=i+1;i<=noOfProcessor;i++)

{

commfailureRate[i][j]=Math.random()\*(0.01-0.001)+0.001;

}

}

int sumgamma=0;

for(int vii=1;vii<=noOfProcessor;vii++)

{

gamma[vii]=Math.random()\*(0.9-0.5)+0.5;

sumgamma+=gamma[vii];

}

meanGamma=sumgamma/noOfProcessor;

**//Algorithm 1**

**//Strategy I**

for(i=1;i<=noOfTask;i++)

aL[i]=1;

i=1;

int []xnew;

int mj;

xnew=new int[noOfTask+1];

System.out.println("Finding Upper Bound");

x = new double[noOfTask+1];

alphadash = new double[noOfTask+1];

for(int vi=1;vi<=noOfTask;vi++){

alphadash[vi]=aL[vi]+1;

for(int j1=aL[vi];j1<noOfProcessor;j1++){

if(j1==noOfProcessor || proTime[vi][j1]!= proTime[vi][j1+1]){

xnew[vi]=proTime[vi][j1];

}

x[vi]=Math.ceil(Math.random()\*(xnew[vi]-0)+0);

if((int)expectedProTime(vi,j1)==x[vi])

{

alphadash[vi]=j1;

}

}

}

**/\*algrithm 2\*/**

**//step3**

ai=new int[noOfTask+1];

for(int vi=1;vi<=noOfTask;vi++)

{

ai[vi]=alpha(alphadash[vi],allotpar,aL[i]);

}

**//step 4**

Map<Integer, Integer> lMap=new HashMap<Integer, Integer>();

rank = new int[noOfTask+1];

for(int vi=noOfTask;vi>=1;vi--)

{

beta=Math.random()\*(5-0)+0;

rank[vi]=(int) rRank(vi,ai[vi]);

lMap.put(vi,rank[vi]);

}

**//step 5**

int[] sortedTask=new int[noOfTask+1];

int x=1;

Map<Integer, Integer> map = sortByValues((HashMap) lMap);

Set set2 = map.entrySet();

Iterator iterator2 = set2.iterator();

while(iterator2.hasNext()) {

Map.Entry me2 = (Map.Entry)iterator2.next();

sortedTask[x]=(int) me2.getKey();

x++;

}

System.out.println("Task\tMeanProTime\tLB\tUB\tRank\tSortedTask\tAP");

for(i=1;i<=noOfTask;i++)

{

System.out.println(i+"\t"+meanProTime[i]+"\t\t"+aL[i]+"\t"+alphadash[i]+"\t"+rank[i]+"\t"+sortedTask[i]+"\t\t"+ai[sortedTask[i]]);

}

Si=new LinkedHashSet[noOfTask+1];

SSj=new LinkedHashSet[noOfTask+1];

avaipro=new double[noOfProcessor+1];

Ct=99999;

double soh,eft;

for(int vi=1;vi<=noOfTask;vi++)

{

Si[vi]=new LinkedHashSet();

SSj[vi]=new LinkedHashSet();

}

for(int vi=1;vi<=noOfTask;vi++){

while(ai[sortedTask[vi]]>Si[sortedTask[vi]].size()){

lMin=9999;

pc=0;

for(int pk=1;pk<=noOfProcessor;pk++)

{

if(!Si[sortedTask[vi]].contains(pk))

{

soh=sumOfoh(sortedTask[vi],pk,Si[sortedTask[vi]]);

eft=earliestFinTime(sortedTask[vi],pk,Si[sortedTask[vi]]);

if((lMin>soh)&& (Ct>eft))

{

pc=pk;

lMin=soh;

}

}

}

if(pc!=0){

Si[sortedTask[vi]].add(pc);

eft=earliestFinTime(sortedTask[vi],pc,Si[sortedTask[vi]]);

avaipro[pc]=eft;

for(Iterator it=Si[sortedTask[vi]].iterator();it.hasNext();)

{

newset.add(earliestFinTime(sortedTask[vi], (int) it.next(),Si[sortedTask[vi]]));

}

double maxx=Double.MIN\_VALUE;

for(double val: newset)

{

if(val>maxx)

{

maxx=val;

}

}

Ct=maxx;

newset.clear();

}

else

{

break;

}

}

System.out.println("Allocated processors for the task "+sortedTask[vi]+" : "+Si[sortedTask[vi]]+"\nCompletion Time : "+Ct);

}

**//algorithm 3**

for(int v=1;v<=noOfTask;v++){

for(int p=1;p<=noOfProcessor;p++){

double r;

double t;

r=reliability(v,p);

t=proTime[v][p];

double n;

n=(Math.random()\*(1-0)+0);

while(n>r){

p=(int) (Math.random()\*(1-0)+0);

t=t+p\*proTime[v][p]+meanRecTime;

n=(Math.random()\*(1-0)+0);

}

}

}

}

public static double function1(int vi,int ai)

{

double res;

double temp1,temp2;

double mf;

DecimalFormat newFormat = new DecimalFormat("#.###");

mf=meanFailure(vi);

temp1=(double) (2\*(Math.pow(mf,2\*ai+1)-(ai+1)\*Math.pow(mf,ai+1)+ai\*Math.pow(mf,ai)));

temp2=(double) (2+Math.pow(mf,2\*ai+1)+(ai-1)\*Math.pow(mf,ai+1)-(ai+2)\*Math.pow(mf,ai));

res=temp1/temp2;

res = Double.valueOf(newFormat.format(res));

return res;

}

public static int alpha(double alphadash1,double allot,int al)

{

return (int) Math.max(Math.min(alphadash1, allot),al);

}

public static double meanFailure(int vi)

{

return 1-meanReliability(vi);

}

public static double meanReliability(int vi)

{

int i,j;

double sum=0;

for(i=1;i<=noOfProcessor;i++)

{

sum+=Math.exp(-failureRate[i]\*proTime[vi][i]);

}

sum=sum/noOfProcessor;

return sum;

}

public static double failure(int vi,int px)

{

// System.out.println("Failure");

return 1-reliability(vi,px);

}

public static double reliability(int vi,int px)

{

//System.out.println("Reliabbility");

return Math.exp(-failureRate[px]\*proTime[vi][px]);

}

public static double expectedProTime(int vi,int ai)

{

double mef=meanFailure(vi);

return (meanProTime[vi]+(((Math.pow(mef, ai))/(1-(Math.pow(mef, ai))))\*((meanProTime[vi]/2)+meanRecTime)));

}

public static double earliestFinTime(int vi,int pk,Set si)

{

double v1,v2;

SSj[vi].add(pk);

double fail=1.0;

for(Object r : SSj[vi])

{

fail\*=failure(vi, (int) r);

}

v1=earliestStaTime(vi,pk);

v2=((proTime[vi][pk])+(fail/(1-fail))\*(((proTime[vi][pk])/2))+recTime[pk]);

return v1+v2;

}

public static double meanTaskCommTime(int i)

{

int sum=0,count=0;

LinkedList<Node> head;

head = list[i];

for(Node ele :head)

{

sum+=ele.getCost();

count++;

}

return sum/count;

}

public static int rRank(int vi,int aii)

{

max=0;

int resrank;

double mtct,rr;

for(Node ele:list[vi])

{

mtct=meanTaskCommTime(vi);

rr=rRank(ele.getAdj(),aii);

if(max<(mtct+rr))

{

max=(int) (mtct+rr);

}

}

resrank= (int) (expectedProTime(vi,aii)\*(1+beta\*Math.pow((1-rbar(vi)), aii)));

return resrank;

}

public static double rbar(int vi)

{

double result;

int k=0;

double mr;

Iterator itr=predList.iterator();

while(itr.hasNext()){

Predecessor pred=(Predecessor)itr.next();

if(pred.getTask()==vi){

k=pred.getPred();

resrbar\*=(1-(1-Math.pow(meanCommReliability(k,vi),ai[k])));

}

}

mr=meanReliability(vi);

result=(mr)/(1-meanGamma+meanGamma\*mr)\*resrbar;

return result;

}

public static double earliestStaTime(int vj,int px)

{

double rrr;

rrr= Math.max(dataReadyTime(vj,px),available(vj,px));

return rrr;

}

public static double dataReadyTime(int vj,int px)

{

Set s=new LinkedHashSet();

Iterator itr=predList.iterator();

while(itr.hasNext()){

Predecessor pred=(Predecessor)itr.next();

if(pred.getTask()==vj)

{

s.add(earliestCommFinTime(pred.getPred(),vj,px,Si[pred.getPred()]));

}

}

double maxValue = 0;

for (Object value : s) {

if ((double)value > maxValue) {

maxValue = (double)value;

}

}

return maxValue;

}

public static double earliestCommFinTime(int i,int j,int x,Set ss)

{

double size=0,ef,ctp;

if(Si[i].size()!=0){

size=1/Si[i].size();

int pp;

for(Iterator k=Si[i].iterator();k.hasNext();)

{

pp=(int)k.next();

ef=earliestFinTime(i,pp,ss);

ctp=commtaskproTime(i,j,pp,x);

fin=fin+ef+ctp;

}

}

return size\*fin;

}

public static double commtaskproTime(int i,int j,int k,int x)

{

if(k==x)

{

return 0;

}

else

{

LinkedList<Node> head;

head = list[i];

double weij = 0,wlkx;

for(Node ele :head)

{

int adj=ele.getAdj();

if(adj==j)

{

weij=ele.getCost();

}

}

wlkx=proCommTime[k][x];

return weij/wlkx;

}

}

public static double sumOfoh(int vj,int pk,Set Si)

{

double rj;

double F;

double vv,xx;

F=fail(vj,pk,Si);

rj=funR(vj,pk,Si);

vv=(beta\*(1-rj)\*proTime[vj][pk])+(beta\*(1-rj)\*F/(1-F)\*((proTime[vj][pk]/2)+recTime[pk]));

xx=earliestFinTime(vj,pk,Si);

return vv+xx;

}

public static double fail(int vj,int pk,Set sj)

{

double failpro=1;

SSj[vj].add(pk);

for(Object r : SSj[vj])

{

failpro\*=failure(vj, (int) r);

}

return failpro;

}

public static double funR(int vj,int pk,Set sj)

{

double rj;

double meanR;

double value;

double fails=1;

value=1;

Set s=new LinkedHashSet();

Iterator itr=predList.iterator();

for(Object r : sj)

{

while(itr.hasNext()){

Predecessor pred=(Predecessor)itr.next();

if(pred.getTask()==vj)

{

fails\*=commReliability(vj,pred.getPred(),(int)r,pk);

}

}

value\*=(1-fails);

}

meanR=meanReliability(vj);

rj=(meanR/(1-gamma[pk]+gamma[pk]\*meanR))\*value;

return rj;

}

public static double commReliability(int i,int j,int k,int x)

{

return Math.exp(-commfailureRate[k][x]\*commtaskproTime(i,j,k,x));

}

public static double meanCommReliability(int i,int j)

{

double val=0.0;

for(int k1=1;k1<=noOfProcessor;k1++)

{

for(int k2=k1;k2<=noOfProcessor;k2++)

{

val+=(2\*commReliability(i,j,k1,k2))/(noOfProcessor\*(noOfProcessor-1));

}

}

return val;

}

public static double commFailure(int i,int j,int k,int x)

{

return 1-commReliability(i,j,k,x);

}

public static double expecWorkSpan(int vi,int ai)

{

double meaf=meanFailure(vi);

return ((ai\*meanProTime[vi])+(((ai\*Math.pow(meaf,ai))/(1-Math.pow(meaf,ai)))\*((meanProTime[vi]/2)+meanRecTime)));

}

public static double available(int vj,int px)

{

return avaipro[px];

}

private static HashMap sortByValues(HashMap map) {

List list = new LinkedList(map.entrySet());

Collections.sort(list, new Comparator() {

public int compare(Object o1, Object o2) {

return ((Comparable) ((Map.Entry) (o2)).getValue())

.compareTo(((Map.Entry) (o1)).getValue());

}

});

HashMap sortedHashMap = new LinkedHashMap();

for (Iterator it = list.iterator(); it.hasNext();) {

Map.Entry entry = (Map.Entry) it.next();

sortedHashMap.put(entry.getKey(), entry.getValue());

}

return sortedHashMap;

}

}

class Node

{

int adj;

int cost;

public Node(int adj,int cost)

{

this.adj=adj;

this.cost=cost;

}

public int getAdj()

{

return adj;

}

public int getCost()

{

return cost;

}

}

class Predecessor

{

int task,pred,cost;

Predecessor(int task,int pred,int cost)

{

this.task=task;

this.pred=pred;

this.cost=cost;

}

public int getTask()

{

return task;

}

public int getCost()

{

return cost;

}

public int getPred()

{

return pred;

}

}

**For Strategy II**

package projects2;

import java.io.\*;

import java.text.DecimalFormat;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashMap;

import java.util.Iterator;

import java.util.LinkedHashMap;

import java.util.LinkedHashSet;

import java.util.LinkedList;

import java.util.List;

import java.util.Map;

import java.util.Set;

public class ProjectS2 {

static int [] task;

static int noOfProcessor,noOfTask,noOfEdges;

static int[][] proTime;

static double [] failureRate;

static double [] meanProTime;

static double meanRecTime;

static double [] recTime;

static double [][] proCommTime;

static double [] x;

static double beta;

static double [] gamma;

static double meanGamma;

static double [] alphadash;

static double [] w;

static Set<Integer>[] Si;

static Set<Integer>[] SSj;

static double[][] commfailureRate;

static Set<Integer> pk=new LinkedHashSet<>();

static Set<Double> newset=new LinkedHashSet<>();

static LinkedList<Node>[] list;

static ArrayList<Predecessor> predList;

static double allotpar;

static int[] ai;

static int[] rank;

static double[] avaipro;

static int max;

static double Ct;

static double lMin;

static int pc;

static double fin=0;

static int resrbar;

private static double[] wotask;

private static double exectime;

public static double eeft;

private static double t;

public static int[] sortedTask;

public static void main(String[] args) throws IOException {

/\*Reading data from the file

\*/

Predecessor pred;

DataInputStream in = null;

DecimalFormat newFormat = new DecimalFormat("#.#");

int avgProTime;

int avgCommTime;

int i,j;

int count=0;

int sum=0;

double ro=0.9999;

double sumf=0.0,meanValue;

int[] aL;

double m1;

String c;

String[] str;

j=1;

try {

in = new DataInputStream(new FileInputStream("C:\\Users\\DELL\\Documents\\IV-Project\\tggwin\\dist\\f1.txt"));

noOfTask=Integer.parseInt(in.readLine());

noOfEdges=Integer.parseInt(in.readLine());

System.out.println("No of Task:"+noOfTask);

System.out.println("No of Edges:"+noOfEdges);

meanProTime=new double[noOfTask+1];

task = new int[noOfTask+1];

predList=new ArrayList();

for(i=1;i<=noOfTask;i++)

task[i]=i;

list = new LinkedList[noOfTask+1];

aL=new int[noOfTask+1];

for(i=1;i<=noOfTask;i++){

list[i]=new LinkedList<>();

}

in.readLine();

c=in.readLine();

while (c!=null) {

str=c.split("\t");

int index;

index=Integer.parseInt(str[0]);

Node node=new Node(Integer.parseInt(str[1]),Integer.parseInt(str[2]));

list[index].add(node);

pred=new Predecessor(Integer.parseInt(str[1]),Integer.parseInt(str[0]),Integer.parseInt(str[2]));

predList.add(pred);

c=in.readLine();

}

}

finally

{

if (in != null)

{

in.close();

}

}

try {

in = new DataInputStream(new FileInputStream("C:\\Users\\DELL\\Documents\\IV-Project\\tggwin\\dist\\f2.txt"));

in.readLine();

noOfProcessor=Integer.parseInt(in.readLine());

System.out.println("No of Processor:"+noOfProcessor);

proTime=new int[noOfTask+1][noOfProcessor+1];

commfailureRate=new double[noOfProcessor+1][noOfProcessor+1];

failureRate=new double[noOfProcessor+1];

gamma=new double[noOfProcessor+1];

in.readLine();

c=in.readLine();

i=1;

while (c!=null) {

str=c.split("\t");

for(int k=0;k<str.length;k++,j++){

proTime[i][j]=Integer.parseInt(str[k]);

}

j=1;

i++;

c=in.readLine();

}

}

finally {

if (in != null) {

in.close();

}

}

proCommTime = new double[noOfProcessor+1][noOfProcessor+1];

for(int i1=1;i1<=noOfProcessor;i1++)

{

for(int j1=i1;j1<=noOfProcessor;j1++)

{

if(i1==j1)

{

proCommTime[i1][j1]=0;

}

else

{

proCommTime[j1][i1]=proCommTime[i1][j1]=Math.random()\*(1.25-0.75)+0.75;

}

}

}

m1=noOfProcessor;

allotpar=((m1\*(2.0+ro))-(Math.sqrt(((2.0+(2.0\*ro)+(ro\*ro))\*(m1\*m1))-((2.0\*m1)\*(1+ro)))))/2.0;

System.out.println("Allotment Paramter : "+allotpar);

for(int ii=1;ii<=noOfTask;ii++){

for(int jj=1;jj<=noOfProcessor;jj++){

sum+=proTime[ii][jj];

}

meanProTime[ii]=sum/noOfProcessor;

sum=0;

}

w=new double[noOfProcessor+1];

recTime=new double[noOfProcessor+1];

for(i=1;i<=noOfProcessor;i++)

{

w[i]=(double) Math.random()\*(1.25-0.75)+0.75;

recTime[i]=1/w[i];

sumf+=recTime[i];

}

meanRecTime=sumf/noOfProcessor;

System.out.println("Mean Recovery Time "+meanRecTime);

for(i=1;i<=noOfProcessor;i++)

{

failureRate[i]=Math.random()\*(0.01-0.001)+0.001;

}

for(i=1;i<=noOfProcessor;i++)

{

for(j=i+1;i<=noOfProcessor;i++)

{

commfailureRate[i][j]=Math.random()\*(0.01-0.001)+0.001;

}

}

int sumgamma=0;

for(int vii=1;vii<=noOfProcessor;vii++)

{

gamma[vii]=Math.random()\*(0.9-0.5)+0.5;

sumgamma+=gamma[vii];

}

meanGamma=sumgamma/noOfProcessor;

**//Algorithm 1**

**//In Strategy II**

for(i=1;i<=noOfTask;i++)

{

if((meanProTime[i]/meanRecTime)>=function1(i,1))

aL[i]=1;

else if((meanProTime[i]/meanRecTime)<function1(i,noOfProcessor-1))

aL[i]=noOfProcessor;

else

{

for(j=1;j<=noOfProcessor-2;j++)

{

if(function1(i,j+1)<=(meanProTime[i]/meanRecTime) &&(meanProTime[i]/meanRecTime) <function1(i,j))

aL[i]=j+1;

}

}

}

i=1;

int []xnew;

int mj;

xnew=new int[noOfTask+1];

System.out.println("Finding Upper Bound");

//finding upper bound

x = new double[noOfTask+1];

alphadash = new double[noOfTask+1];

**//CVTA(Constructing Virtual Task Algorithm)**

for(int vi=1;vi<=noOfTask;vi++){

alphadash[vi]=aL[vi]+1;

for(int j1=aL[vi];j1<noOfProcessor;j1++){

if(j1==noOfProcessor || proTime[vi][j1]!= proTime[vi][j1+1]){

xnew[vi]=proTime[vi][j1];

}

x[vi]=Math.ceil(Math.random()\*(xnew[vi]-0)+0);

if((int)expectedProTime(vi,j1)==x[vi])

{

alphadash[vi]=j1;

}

}

}

**/\*algrithm 2\*/**

**//step3**

ai=new int[noOfTask+1];

wotask=new double[noOfTask+1];

for(int vi=1;vi<=noOfTask;vi++)

{

wotask[vi]=Math.random()\*(59-1)+1;

}

for(int vi=1;vi<=noOfTask;vi++)

{

ai[vi]=alpha(alphadash[vi],allotpar,aL[i]);

}

**//step 4**

Map<Integer, Integer> lMap=new HashMap<Integer, Integer>();

rank = new int[noOfTask+1];

for(int vi=noOfTask;vi>=1;vi--)

{

beta=Math.random()\*(5-0)+0;

rank[vi]=(int) rRank(vi,ai[vi]);

lMap.put(vi,rank[vi]);

}

**//step 5**

sortedTask=new int[noOfTask+1];

int x=1;

Map<Integer, Integer> map = sortByValues((HashMap) lMap);

Set set2 = map.entrySet();

Iterator iterator2 = set2.iterator();

while(iterator2.hasNext()) {

Map.Entry me2 = (Map.Entry)iterator2.next();

sortedTask[x]=(int) me2.getKey();

x++;

}

System.out.println("Task\tMeanProTime\tLB\tUB\tRank\tSortedTask\tAP");

for(i=1;i<=noOfTask;i++)

{

System.out.println(i+"\t"+meanProTime[i]+"\t\t"+aL[i]+"\t"+alphadash[i]+"\t"+rank[i]+"\t"+sortedTask[i]+"\t\t"+ai[sortedTask[i]]);

}

Si=new LinkedHashSet[noOfTask+1];

SSj=new LinkedHashSet[noOfTask+1];

avaipro=new double[noOfProcessor+1];

double soh,eft;

for(int vi=1;vi<=noOfTask;vi++)

{

Si[vi]=new LinkedHashSet();

SSj[vi]=new LinkedHashSet();

}

for(int vi=1;vi<=noOfTask;vi++){

Ct=99999;

exectime=0;

eeft=1.0;

while(ai[sortedTask[vi]]>Si[sortedTask[vi]].size() && exectime<eeft){

lMin=9999;

pc=0;

for(int pk=1;pk<=noOfProcessor;pk++)

{

if(!Si[sortedTask[vi]].contains(pk))

{

soh=sumOfoh(sortedTask[vi],pk,Si[sortedTask[vi]]);

eft=earliestFinTime(sortedTask[vi],pk,Si[sortedTask[vi]]);

if((lMin>soh)&& (Ct>eft))

{

pc=pk;

lMin=soh;

}

}

}

if(pc!=0){

Si[sortedTask[vi]].add(pc);

eeft=earliestFinTime(sortedTask[vi],pc,Si[sortedTask[vi]]);

avaipro[pc]=eeft;

for(Iterator it=Si[sortedTask[vi]].iterator();it.hasNext();)

{

newset.add(earliestFinTime(sortedTask[vi], (int) it.next(),Si[sortedTask[vi]]));

}

double maxx=Double.MIN\_VALUE;

for(double val: newset)

{

if(val>maxx)

{

maxx=val;

}

}

Ct=maxx;

newset.clear();

}

else

{

break;

}

exectime=exectime+2;

}

System.out.println("Allocated processors for the task "+sortedTask[vi]+" : "+Si[sortedTask[vi]]+"\nCompletion Time : "+Ct);

}

**//algorithm 3**

for(int v=1;v<=noOfTask;v++){

for(int p=1;p<=noOfProcessor;p++){

double r;

r=reliability(v,p);

t=proTime[v][p];

double n;

n=(Math.random()\*(1-0)+0);

while(n>r){

p=(int) (Math.random()\*(1-0)+0);

t=t+p\*proTime[v][p]+meanRecTime;

n=(Math.random()\*(1-0)+0);

}

}

System.out.println("processing time of task "+v+" is:" +t);

}

}

public static double function1(int vi,int ai)

{

double res;

double temp1,temp2;

double mf;

DecimalFormat newFormat = new DecimalFormat("#.###");

mf=meanFailure(vi);

temp1=(double) (2\*(Math.pow(mf,2\*ai+1)-(ai+1)\*Math.pow(mf,ai+1)+ai\*Math.pow(mf,ai)));

temp2=(double) (2+Math.pow(mf,2\*ai+1)+(ai-1)\*Math.pow(mf,ai+1)-(ai+2)\*Math.pow(mf,ai));

res=temp1/temp2;

res = Double.valueOf(newFormat.format(res));

return res;

}

public static int alpha(double alphadash1,double allot,int al)

{

return (int) Math.max(Math.min(alphadash1, allot),al);

}

public static double meanFailure(int vi)

{

return 1-meanReliability(vi);

}

public static double meanReliability(int vi)

{

int i,j;

double sum=0;

for(i=1;i<=noOfProcessor;i++)

{

sum+=Math.exp(-failureRate[i]\*proTime[vi][i]);

}

sum=sum/noOfProcessor;

return sum;

}

public static double failure(int vi,int px)

{

// System.out.println("Failure");

return 1-reliability(vi,px);

}

public static double reliability(int vi,int px)

{

//System.out.println("Reliabbility");

return Math.exp(-failureRate[px]\*proTime[vi][px]);

}

public static double expectedProTime(int vi,int ai)

{

double mef=meanFailure(vi);

return (meanProTime[vi]+(((Math.pow(mef, ai))/(1-(Math.pow(mef, ai))))\*((meanProTime[vi]/2)+meanRecTime)));

}

public static double earliestFinTime(int vi,int pk,Set si)

{

double v1,v2;

//System.out.println("Earliest");

SSj[vi].add(pk);

double fail=1.0;

for(Object r : SSj[vi])

{

fail\*=failure(vi, (int) r);

}

v1=earliestStaTime(vi,pk);

v2=((proTime[vi][pk])+(fail/(1-fail))\*(((proTime[vi][pk])/2))+recTime[pk]);

return v1+v2;

}

public static double meanTaskCommTime(int i)

{

int sum=0,count=0;

LinkedList<Node> head;

head = list[i];

for(Node ele :head)

{

sum+=ele.getCost();

count++;

}

return sum/count;

}

public static int rRank(int vi,int aii)

{

max=0;

int resrank;

double mtct,rr;

for(Node ele:list[vi])

{

mtct=meanTaskCommTime(vi);

rr=rRank(ele.getAdj(),aii);

if(max<(mtct+rr))

{

max=(int) (mtct+rr);

}

}

resrank= (int) (expectedProTime(vi,aii)\*(1+beta\*Math.pow((1-rbar(vi)), aii)));

return resrank;

}

public static double rbar(int vi)

{

double result;

int k=0;

double mr;

Iterator itr=predList.iterator();

while(itr.hasNext()){

Predecessor pred=(Predecessor)itr.next();

if(pred.getTask()==vi){

k=pred.getPred();

resrbar\*=(1-(1-Math.pow(meanCommReliability(k,vi),ai[k])));

}

}

mr=meanReliability(vi);

result=(mr)/(1-meanGamma+meanGamma\*mr)\*resrbar;

return result;

}

public static double earliestStaTime(int vj,int px)

{

double rrr;

rrr= Math.max(dataReadyTime(vj,px),available(vj,px));

return rrr;

}

public static double dataReadyTime(int vj,int px)

{

Set s=new LinkedHashSet();

Iterator itr=predList.iterator();

while(itr.hasNext()){

Predecessor pred=(Predecessor)itr.next();

if(pred.getTask()==vj)

{

s.add(earliestCommFinTime(pred.getPred(),vj,px,Si[pred.getPred()]));

}

}

double maxValue = 0;

for (Object value : s) {

if ((double)value > maxValue) {

maxValue = (double)value;

}

}

return maxValue;

}

public static double earliestCommFinTime(int i,int j,int x,Set ss)

{

double size=0,ef,ctp;

if(Si[i].size()!=0){

size=1/Si[i].size();

int pp;

for(Iterator k=Si[i].iterator();k.hasNext();)

{

pp=(int)k.next();

ef=earliestFinTime(i,pp,ss);

ctp=commtaskproTime(i,j,pp,x);

fin=fin+ef+ctp;

}

}

return size\*fin;

}

public static double commtaskproTime(int i,int j,int k,int x)

{

if(k==x)

{

return 0;

}

else

{

LinkedList<Node> head;

head = list[i];

double weij = 0,wlkx;

for(Node ele :head)

{

int adj=ele.getAdj();

if(adj==j)

{

weij=ele.getCost();

}

}

wlkx=proCommTime[k][x];

return weij/wlkx;

}

}

public static double sumOfoh(int vj,int pk,Set Si)

{

double rj;

double F;

double vv,xx;

F=fail(vj,pk,Si);

rj=funR(vj,pk,Si);

vv=(beta\*(1-rj)\*proTime[vj][pk])+(beta\*(1-rj)\*F/(1-F)\*((proTime[vj][pk]/2)+recTime[pk]));

xx=earliestFinTime(vj,pk,Si);

return vv+xx;

}

public static double fail(int vj,int pk,Set sj)

{

double failpro=1;

SSj[vj].add(pk);

for(Object r : SSj[vj])

{

failpro\*=failure(vj, (int) r);

}

return failpro;

}

public static double funR(int vj,int pk,Set sj)

{

double rj;

double meanR;

double value;

double fails=1;

value=1;

Set s=new LinkedHashSet();

Iterator itr=predList.iterator();

for(Object r : sj)

{

while(itr.hasNext()){

Predecessor pred=(Predecessor)itr.next();

if(pred.getTask()==vj)

{

fails\*=commReliability(vj,pred.getPred(),(int)r,pk);

}

}

value\*=(1-fails);

}

meanR=meanReliability(vj);

rj=(meanR/(1-gamma[pk]+gamma[pk]\*meanR))\*value;

return rj;

}

public static double commReliability(int i,int j,int k,int x)

{

return Math.exp(-commfailureRate[k][x]\*commtaskproTime(i,j,k,x));

}

public static double meanCommReliability(int i,int j)

{

double val=0.0;

for(int k1=1;k1<=noOfProcessor;k1++)

{

for(int k2=k1;k2<=noOfProcessor;k2++)

{

val+=(2\*commReliability(i,j,k1,k2))/(noOfProcessor\*(noOfProcessor-1));

}

}

return val;

}

public static double commFailure(int i,int j,int k,int x)

{

return 1-commReliability(i,j,k,x);

}

public static double expecWorkSpan(int vi,int ai)

{

double meaf=meanFailure(vi);

return ((ai\*meanProTime[vi])+(((ai\*Math.pow(meaf,ai))/(1-Math.pow(meaf,ai)))\*((meanProTime[vi]/2)+meanRecTime)));

}

public static double available(int vj,int px)

{

return avaipro[px];

}

private static HashMap sortByValues(HashMap map) {

List list = new LinkedList(map.entrySet());

Collections.sort(list, new Comparator() {

public int compare(Object o1, Object o2) {

return ((Comparable) ((Map.Entry) (o2)).getValue())

.compareTo(((Map.Entry) (o1)).getValue());

}

});

HashMap sortedHashMap = new LinkedHashMap();

for (Iterator it = list.iterator(); it.hasNext();) {

Map.Entry entry = (Map.Entry) it.next();

sortedHashMap.put(entry.getKey(), entry.getValue());

}

return sortedHashMap;

}

}

class Node

{

int adj;

int cost;

public Node(int adj,int cost)

{

this.adj=adj;

this.cost=cost;

}

public int getAdj()

{

return adj;

}

public int getCost()

{

return cost;

}

}

class Predecessor

{

int task,pred,cost;

Predecessor(int task,int pred,int cost)

{

this.task=task;

this.pred=pred;

this.cost=cost;

}

public int getTask()

{

return task;

}

public int getCost()

{

return cost;

}

public int getPred()

{

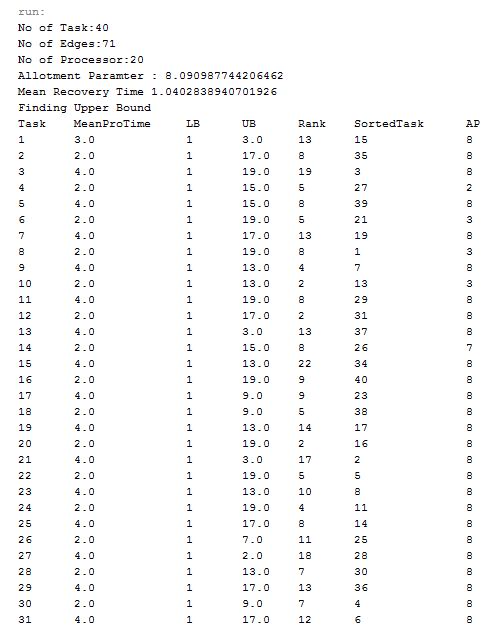
return pred;

}

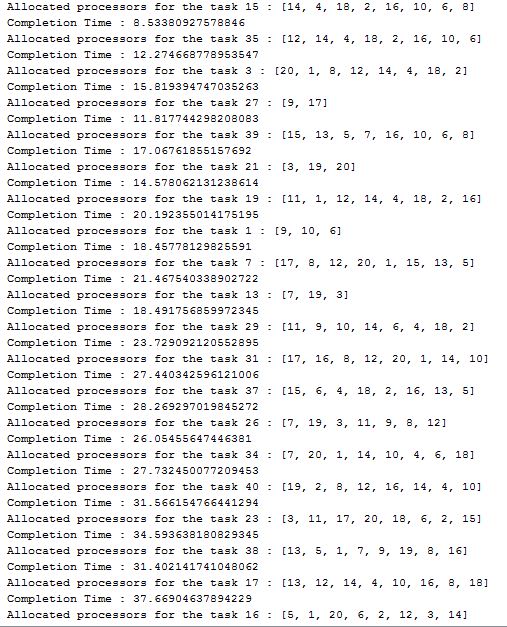
}

* 1. **SCREENSHOT**

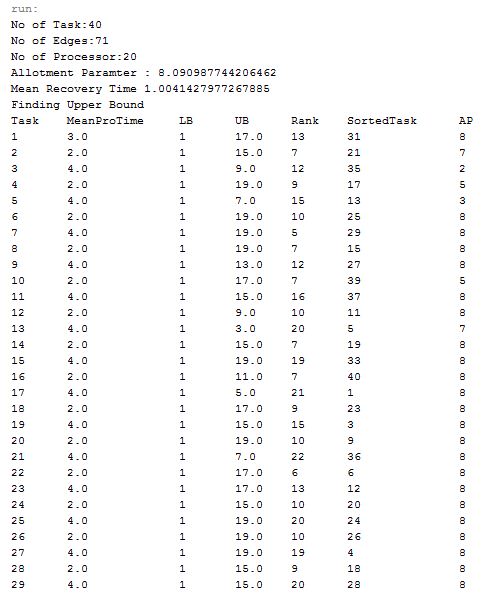
**For Strategy I:**

******

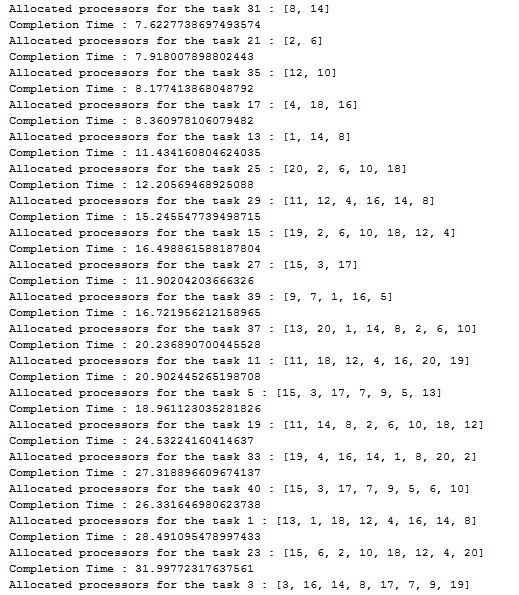
**For Strategy I:**

******

**For Strategy II:**

****

**For Strategy II:**

****

CONCLUSION

**6 CONCLUSION**

A heterogeneous allotment-aware scheduling algorithm is proposed. This is a two-phase algorithm. In the first phase, a favorable allotment is determined using a linear program formulation and a rounding procedure. In the second phase, a scheduling method that is based on the expected execu- tion time and the communication time is implemented. Two strategies I and II are considered. In Strategy I, no rep- lication of a task can be stopped. In Strategy II, once a replica- tion of a task is completed, the other replications of a task are immediately aborted. Strategy I is preferred for communica- tion-intensive applications and Strategy II is preferred for computation-intensive applications. The HAAS algorithm significantly outperformed the other algorithms in terms of the SLR, reli- ability and speedup.

* xi\*
* Allotted processors can be identified such that Tp(vi, ) = xi
* f1(vi,)=
* Allotment Parameter:

μ=

* Reliability Aware Rank:

RRank(vi)=Tp(vi,αi)(1+β(1-)αi)+

* The mean processing time is computed using

(vi)=

* The mean communication time of edge is computed using

(ei,j)=

* The mean reliability of task is computed using

=

* Mean Communication reliability probability between two tasks.

[Eei,j] =

* Mean Overhead of a task on processor.

=

=

* Earliest Computation Finish Time

EFT(vi,pk,Si)=EST(vi,pk)+tp(vi,k)+

* Earliest Communication Finish Time

EFT(ei,j,Si,px)=