J Inf Process Syst, Vol.10, No.2, pp.193~214, June 2014
http://dx.doi.org/10.3745/JIPS.01.0001

pISSN 1976-913X
eISSN 2092-805X

Duplication with Task Assignment in Mesh Distributed System

Rashmi Sharma* and Nitin*

Abstract—Load balancing is the major benefit of any distributed system. To facilitate this advantage, task duplication and migration methodologies are employed. As this paper deals with dependent tasks (DAG), we used duplication. Task duplication reduces the overall schedule length of DAG along-with load balancing. This paper proposes a new task duplication algorithm at the time of tasks assignment on various processors. With the intention of conducting proposed algorithm performance computation; simulation has been done on the Netbeans IDE. The mesh topology of a distributed system is simulated at this juncture. For task duplication, overall schedule length of DAG is the main parameter that decides the performance of a proposed duplication algorithm. After obtaining the results we compared our performance with arbitrary task assignment, CAWF and HEFT-TD algorithms. Additionally, we also compared the complexity of the proposed algorithm with the Duplication Based Bottom Up scheduling (DBUS) and Heterogeneous Earliest Finish Time with Task Duplication (HEFT-TD).

Keywords—Distributed System(DS), Task Assignment Heuristics, Task Duplication(TD), Directed Acyclic Graph(DAG)

1. INTRODUCTION

The Distributed System consists of numerous self-ruling processors that communicate via interconnection network. Each network follows different connectivity architectures, that are known as network topology. Mesh topology is one of the topologies [1] that are employed here for network connectivity. However, handling of mesh topology is very difficult because of the inter-connectivity between every node. Such network connectivity in-between processors can be of homogeneous or heterogeneous type. The homogeneous system shares identical architecture whereas the heterogeneous system shares diverse architecture. Therefore, task scheduling is complicated in the heterogeneous system due to non-uniform speed and communication bandwidth. List-based and cluster based are two important scheduling classes that help with task scheduling in the heterogeneous system [2]. This paper uses cluster based scheduling to solve the complication of heterogeneity of processors. On the basis of processor computational capacity [3] the entire system splits into three clusters (High, Medium, and Low).

Parallel task execution is the primary advantage of distributed system. Here, the independent subtasks of any task can be run correspondingly on various processors. These subtasks are generated from single task that is called--DAG (shows the interdependency in-between subtasks). In order to accomplish the complete task (DAG) as fast as possible, subtasks are allocated to

Manuscript received March 06, 2013 ; first revision September 11, 2013 ; accepted March 31, 2014.
Corresponding Author : Nitin (delnitin@ieee.org)
* Department of Computer Science and Information Technology, Jaypee University of Information Technology, Waknaghat, Solan, India (rashmi.nov30@gmail.com, delnitin@ieee.org)

separate processors of the same organization. These processors execute allocated tasks in parallel according to their computational speed. After achieving the results, the destination processor transmits it to the source processor (origin) of tasks. This paper explains the strategic duplication of tasks on the various processors that finally reduce the schedule length of the entire DAG.

The execution of any task passes through following two phases:

1. There is partitioning heuristic under which tasks split into dependent/independent tasks known as DAG [4]. DAG represents the size of each task along with the computational power consumption.
2. The allotment of processors to these distributed sub-tasks is another phase. First-Fit, Worst-Fit, Best–Fit, and Communication Aware Worst-Fit are some task assignment heuristics [4-6] that work with/ without task duplication.

These above-mentioned partitioning and assignment heuristics fall under the scheduling problem. This problem is also known as grain size determination [7], the clustering problem [8,9], or internalization pre-pass [10].

These above-mentioned First-Fit, Worst-Fit, and Best-Fit heuristics work in a sequential manner and the duplication of a task is not followed here. CAWF is designed for the reduction of communication costs in which two dependent tasks (predecessor-successor) can be allocated on the same processor, which reduces the communication cost between tasks. In the case of multiple successors of a single predecessor, CAWF assigns one of the successors to the same processor with its predecessor and the rest of the successors use the Worst-Fit heuristic for allocation. Hence, this is the downside of the CAWF algorithm.

This paper implements a new task duplication method that will overcome the limitation of CAWF. We have chosen the basic task assignment (duplication is not allowed here), CAWF and HEFT-TD algorithms to compare with the proposed algorithm. We chose these because these algorithms have their own properties, time complexities, and advantages during task assignment. There are many other algorithms that can be used for the execution of DAG in heterogeneous environment i.e. DBUS and HEFT-TD [2,11] algorithms (few properties are comparable to the proposed algorithm with a different approach).

In this work, we have proposed task duplication process at the time of its allocation before the execution. In the proposed algorithm, DAG is traversed by using bottom-up approach and we checked the dependencies of tasks with other tasks of DAG. If two independent tasks are found, then those tasks will execute independently (in parallel). The background and preliminaries are discussed in the following section. Additionally, Section III explains the proposed task duplication algorithm. Further on in the paper, results and discussions will be shown, followed by our conclusions and future work.

Table 1. Symbol Table

Symbol	Definition
G	DAG
V	Vertices of DAG
E	Edges of DAG
$CC(t_{i,k})$	Computation cost of task t_i on k^{th} processor
t_i	Task

p	Total number of processors
C_{t_i,t_j}	Communication cost b/w t_i and t_j
$C(t_i)$	Average computation cost of given task
$v(t_i, t_j)$	data sent from task t_i to t_j.
S_y	Start-up cost on given processor
$D_{x,y}$	Data transfer rate from processor p_x to p_y
TFT	Total Finish Time
$pred(T_i)$	Predecessor of task T_i
T	Set of tasks
P	Set of processors
ET	Execution Time
DP	Destination Processor

2. BACKGROUND AND PRELIMINARIES

Load balancing is the chief significance of the distributed system. This load balancing is accomplished by using task duplication or migration in-between processors. As we are dealing with dependent tasks, we employed the duplication of tasks here. Main role of task duplication is to reduce the communication cost, which helps in the reduction of overall schedule length of entire DAG. Many researchers have suggested various strategies for task duplication [11-14].

DAG is an arrangement of multiple tasks, out of which some tasks are dependent on previous tasks and some are independent. In the case of dependency, successor tasks couldn't execute before the execution of dependent predecessor tasks. On the other hand, independent tasks can execute in-parallel on several processors. In a DAG $G = (V, E)$, E is a link between two nodes that explains the communication cost between two dependent tasks. These sub-tasks (tasks) are assigned to various processors based on the features already discussed in our previous paper [15] and in many other papers [11,13].

Definition2.1: The computation cost of any task on a given processor is dependent on the computational capacity of a particular processor. The time taken by a processor to execute a particular task is known as the computation cost or execution time of a task on a given processor. Computation cost also depends on the size of a task as well.

Consider $CC(t_{i,k})$ is the computation cost of task $t_i$on k^{th} processor from p number of processors. Hence, the average computation cost of any task $C(t_i)$ is defined as:

$$C(t_i)=\sum_{k=1}^{p} CC(t_{i,k})/p \tag{1}$$

Definition2.2: Communication cost $(C_{t_{i,t_j}})$ is the time consumed by the processor in sending the data (results) of one task to another processor. This communication cost is dependent on the volume of communicating data (amount of data under communication) and data transfer rate from the source to the destination processor [11,14].

$$C_{t_{i,t_j}} = S_y + \frac{v(t_i,t_j)}{D_{x,y}} \tag{2}$$

If two jobs are assigned to the same processor then the communication cost, $C_{t_{i,t_j}} = 0$.
Definition2.3: Total Finish Time (TFT) [16]:
The TFT of k^{th} tasks on P_n processor is:

$$TFT(P_n) = \sum_{i=1}^{k}(new\ arrival\ (T_i) + \ Execution\ time(T_i)) \quad (3)$$

$$new\ arrival(T_i) = Execution\ time\big(pred(T_i)\big) + C_{pred(T_i),T_i} \quad (4)$$

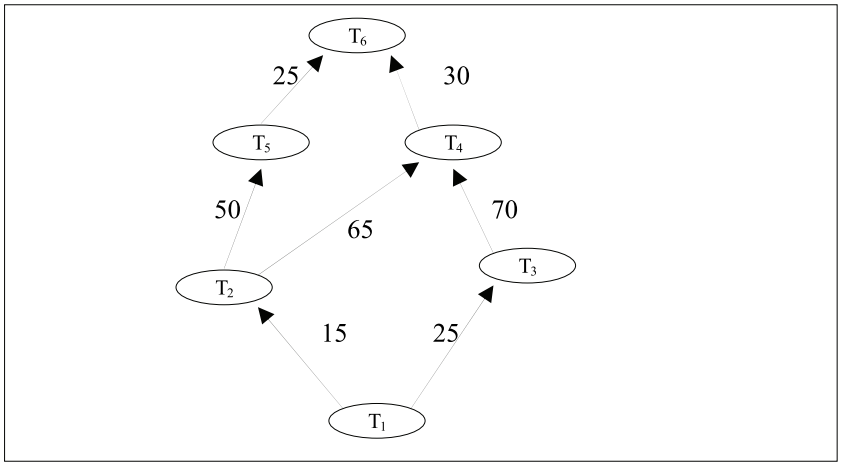


Fig. 1. Arbitrary DAG with Communication Cost

The above figure explains that the DAG contains tasks (subtasks) $\{T_1, T_2, T_3, T_4, T_5, T_6\}$ and $\{25,30,50,65,70,15,25\}$ are their respective communication costs in-between dependent tasks. Later on, the generation of random DAGs and subtasks (tasks of the DAG) will be assigned to the respective processors. Task assignment is the process of assigning multiple tasks to the numerous processors. Additionally, we used the parallel allocation and execution method for task assignment here [17].

In distributed system the selection of processors for task allocation can be sequential or parallel. For sequential task allocation First-Fit, Best-Fit, and Worst-Fit are well known. All of these mentioned sequential allocation heuristics focus on computation costs but not on communication costs. In [6] the author has discussed another assignment heuristic approach that focuses on communication cost along with computation cost. This heuristic is known as a Communication Aware Worst Fit (CAWF). According to CAWF, same processor is assigned to a pair of predecessor-successor sub-tasks that brings down the communication cost in-between the assigned pair. But if one predecessor has multiple successors, then the Worst-Fit algorithm is used for the rest successors. Although the sequential assignment of tasks is also present here but this algorithm seems helpful in reducing the communication cost.

Equation (3) calculates the TFT of a completed task on a particular processor. This TFT is dependent on the execution cost of every sub-task (task) on the respective processors and on the communication cost between dependent tasks (sub-tasks). The table below explains the execution time (computation cost) of tasks on respective processors:

Table 2. Execution Time of Tasks on Processors

T_i \ P_j	P_1	P_2	P_3	P_4
T_1	35	5	15	10
T_2	9	4	10	7
T_3	6	8	4	12
T_4	23	45	15	26
T_5	10	7	9	11
T_6	30	9	5	18

As we are working on distributed system, we considered the parallel execution and allocation of tasks here. Let us consider the case when processors are randomly selected for task assignment and execution as well. In Fig. 2, the P_3 processor is randomly selected for T_1, T_4 *and* T_6 tasks; P_2 is assigned for T_3 and T_5; similarly P_1 executes the T_2 task. Based on the execution cost and communication costs between the processors, the overall DAG schedule length has been calculated. In arbitrary selection, the DAG schedule length may vary because it is dependent on a preferred processor. There is no criterion for the processor selection for task execution in the arbitrary method.

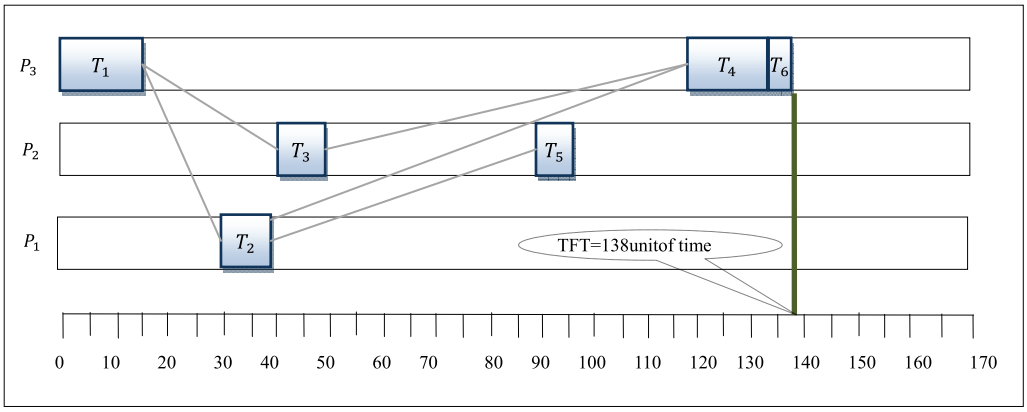


Fig. 2. The Arbitrary Allocation of Tasks on Processors in the Distributed System

$$DAG\ Schedule\ Length = \max_{1 \le n} TFT(P_n) \tag{5}$$

$$= max(TFT(P_1), TFT(P_2), TFT(P_3)) = max(39,96,138) = 138\ unit\ of\ time \tag{6}$$

Now, in Fig. 3, tasks are assigned according to the CAWF algorithm. Tasks that have a predecessor and successor are allocated to the same processor and another task will follow the Worst-Fit. (From figure 1) T_1 is the only predecessor of tasks T_2 & T_3. Similarly, T_2 is the predecessor of tasks T_5 & T_4. According to CAWF, one of the successors of these predecessors will allocate on the same CPU and other tasks will follow the Worst-Fit. Therefore, T_1 and T_3

(dependent tasks) are assigned on processor P_1. Similarly, T_2, T_4, T_5 and T_6 are interdependent tasks and are sequentially assigned to the next processor P_2. Lastly, on the basis of computation cost and communication cost, DAG schedule length has been calculated, which is lesser than the previous method due to a reduction in communication costs.

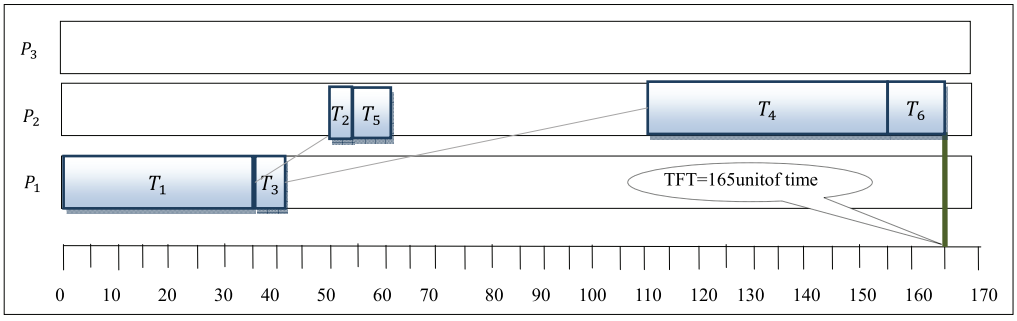


Fig. 3. DAG Execution Using the CAWF Heuristic

$$DAG\ Schedule\ Length = max(TFT(P_1), TFT(P_2)) = max(41,165)$$
$$= 165\ unit\ of\ time \quad (7)$$

Furthermore, the third type of allocation is our proposed task duplication algorithm that is essentially an advanced adaptation of CAWF. In this method, tasks that have lesser execution cost as compared to the communication cost become a duplicated task on a given processor.

From example above task T_1 duplicates on P_4 processor, because its communication costs towards dependent tasks T_2, and T_3 is greater than its computation cost on particular processors. Similarly, the computation costs of other dependent tasks are greater than their communication costs and therefore, those tasks will not duplicate on other processors. By applying this duplication technique the overall schedule length of the DAG is comparatively lower than the previous methods.

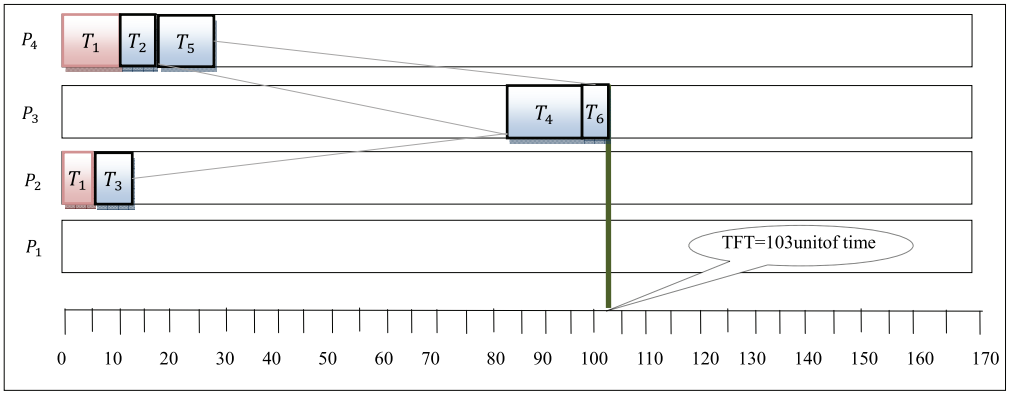


Fig. 4. Proposed Task Duplication Methodology

$$DAG\ Schedule\ Length = max(TFT(P_2), TFT(P_3), TFT(P_4)) = max(13,103,28)$$
$$= 103\ unit\ of\ time \quad (8)$$

The proposed duplication algorithm is somewhat similar to the HEFT-TD and DBUS algorithms. Additionally, the approach used here is different. The approach used in HEFT-TD is top-down and we are using the bottom-up approach. Therefore, the proposed algorithm gives similar or only slightly improved results than the existing ones. In the next section, we will explain the proposed algorithm for task duplication followed by explaining the simulation results.

3. THE PROPOSED TASK DUPLICATION ALGORITHM

There are many approaches that have been used for task assignment (i.e., First-Fit, Best-First, Worst-Fit and Communication Aware Worst Fit (CAWF) algorithm etc.). However, all of these heuristics select processors sequentially (the first processor assigns first and so forth) for the assignment of tasks without duplication. The CAWF algorithm reduces the communication cost by assigning the predecessor and successor on a single processor. This approach works fine if the predecessor has a single successor; therefore, the downside of the CAWF approach is the existence of multiple successors of a single predecessor task. We recognize that the primary motive of task duplication is to reduce the communication cost that affects the overall schedule length of DAG. Hence, in order to overcome the problem of CAWF, we used the task duplication methodology. These days, numerous researchers have designed many task duplication algorithms [2,11,14] with different approaches.

As we mentioned earlier, the topology we are using is a mesh that connects every processor with all of the other processors of the system. After the generation of DAG on the given processor, our proposed algorithm uses the bottom-up traversing of DAG, which is similar to the DBUS algorithm [2]. This approach determines the dependent and independent tasks of DAG. Independent tasks can execute in parallel and duplication is used for dependent tasks. Task assignments depend on the computational capacity of an assigned processor, because the job will execute on allotted processors. The duplication of a task is based on the communication cost and execution cost of processors. At the time of duplication, there are a few critical things that must be remembered. They are as listed below.

1) Limited number of duplicates: The algorithm must understand the number of duplications of any task (Successor/Predecessor). The algorithm should avoid useless duplication of tasks, consider that the C_{t_i,t_j} between i^{th} and j^{th} task is less than the $C(P_j)$ of j^{th} processor and then there is no need to duplicate a task.

2) While conducting the bottom-up traversing of DAG, all child tasks are executed first and then the parent tasks are. Due to which, parent task duplication decreases.

In the remainder of this section, the different modules of task duplication are elucidated.

3.1 Clustering of Heterogeneous Processors with Mesh Topology

We used mesh topology here for the interconnection of heterogeneous processors. Therefore, processor computational power shows incongruence. In order to handle this heterogeneous behavior of the system, the complete distributed system splits into three clusters (based on computational capacity i.e., High, Medium, and Low). For the grouping of processors, we have fixed some of the ranges that determine the efficiency level of processors. These ranges make a decision randomly from 0 to 10.

Fig. 5. Clustering of Processors

In the above figure, each cell represents a node (processor), and on the basis of efficiency range, the complete system is divided into three groups.

Blue represents "Low Efficiency," which comes under 0 to 4 ranges. Yellow represents "Medium Efficiency and this range lies between 5 to 7. Lastly, red is for "High Efficiency," and its range lies between 8 to 10.

Along with efficiencies, these nodes posses communication costs in-between their communication channels; and we represented that cost with the help of an adjacency matrix.

```
0 60 43 40 180 65 12 51
60 0 12 53 10 43 11 53
43 12 0 60 11 67 13 21
40 53 60 0 34 56 24 22
180 10 11 34 0 32 43 16
65 43 67 56 32 0 56 45
12 11 13 24 43 56 0 44
51 53 21 22 16 45 44 0
```

Fig. 6. Communication Costs between Nodes

The above image is a matrix of communication costs between several CPUs. For example: $C_{3,2}$.

3.2 Generation of a Task on Nodes

In a distributed heterogeneous system, DAG can be generated on any node at any time. In the above figure, task generation on a particular processor is indicated by the green color.

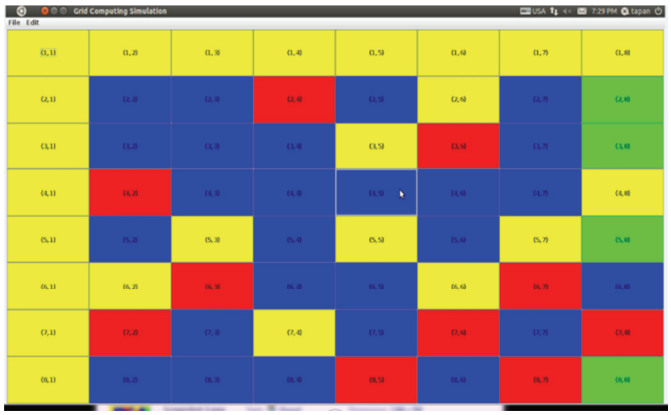


Fig. 7. Task Generation on any node of system randomly

This algorithm generates tasks randomly on any node and the getEfficiency() function retrieves the efficiency of a particular node.

BEGIN

TASKEXECUTION-ACTIONPERFORMED (java.awt.event.ActionEventevt)

1. Calendar c= Calendar.getInstance()
2. long m=c.getTimeInMillis()
3. Random r=new Random(m)
4. xcor=r.nextInt()
5. m=c.getTimeInMillis()
6. r.setSeed(m)
7. ycor=r.nextInt()
8. jbArray[Math.abs(xcor%5)][Math.abs(ycor%5)].setBackground(Color.GREEN)
9. group.getEfficiency(Math.abs(xcor%5),Math.abs(ycor %5))

END

After generation of DAG, the following algorithm retrieves efficiency of that particular node and its communication cost with near (other) nodes.

The getmatrix() function obtains communication costs from one processor (where the task generates) to other nodes. The gettaskmatrix() function sets the random DAG on a particular node.

SHOW-ACTION-PERFORMED (java.awt.event.ActionEventevt)

BEGIN

1. ndag.getmatrix()
2. tdag.gettaskmatrix()

END

3.3 The DAG Matrix and its Tracing

The above module is the basic framework for our simulation. This module explains the random DAG (in matrix format) of tasks; it shows the dependency/independence between tasks. In the DAG matrix, 0 represents an independent task and 1 represents a dependency between the two.

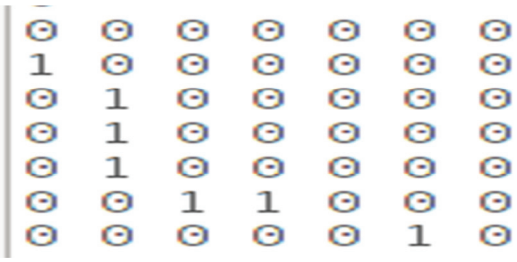


Fig. 8. DAG Representation in Terms of a Matrix

After the creation of DAG for its execution, the bottom-up approach is used here. Task T_7 is an independent task (Column of T_7 contains 0), T_6 is dependent on T_7 (T_6 column has 1 on T_7 row). Similarly, other dependencies have been made. For the traversing of this matrix of tasks (DAG); first, we check the dependencies (the occurrence of 1's in a column) and based on this occurrence a sorting of tasks is carried out. This computation takes $O(n^2)$ time.

Input: A sequence of n subtasks of DAG (t_1, t_2, t_3 … … … $t_{taskDAG}$).

Output: DAG in terms of matrix has been generated.

BEGIN	*cost*	*times*
1. count=0	c_1	1
2. **for** i=0 to taskDAG.length	c_2	$n+1$
3. **for** j=0 to taskDAG.length	c_3	n^2
4. **if** taskDAG[j][i]==1	c_4	$n-1$
5. count++	c_5	$n-1$
6. End **for**		
7. End **for**		
END		

Hence, we find that in the worst case, the running time of DAG generation is

$$T(n) = c_1.1 + c_2.(n+1) + c_3.n^2 + c_4.(n-1) + c_5.(n-1)$$
$$= c_1 + c_2.n + c_2 + c_3.n^2 + c_4.n - c_4 + c_5.n - c_5$$
$$= c_3.n^2 + (c_2 + c_4 + c_5)n + (c_1 + c_2 - c_4 - c_5) = O(n^2).$$

The running time of the algorithm is the sum of running times for each executed statement. We can express the above equation in the form of $an^2 + bn + c$ for constants a, b and c which again depend on the statement costs c_i; it is thus a quadratic function of n *i.e.* n^2.

After getting the dependent tasks we check whether this dependency is direct or indirect. For example, (Fig. 8) task directly dependent on task and is indirectly dependent on T_6 ($T_2 \rightarrow T_4 \rightarrow T_6$). These dependencies are determined by using Boolean Matrix Multiplication.

Input: Two copies of DAG for Boolean Matrix Multiplication.

Output: Dependency of tasks.

CHECK-INDIRECT-DEPENDENCY (matrixsize1 [][],matrixsize2 [][],Row, Column)

```
BEGIN
    m= ((matrixsize1.length)*(matrixsize1.length))/2
for   count=0 to m
ResultMatrix=new int[matrixsize1.length][matrixsize1.length]
      forI=Row to matrixsize1.length
int [] rowVector=getCurrentRow(matrixsize1, i)
        for   j=Column to matrixsize2.length
int[] columnVector=getCurrentColumn(matrixsize2, j)
              for   k=0 to matrixsize2.length
                    ifrowVector[k] == 1 &&columnVector[k]==1
ResultMatrix[i][j]=1
                        flag=true
                      break
                    End if
              End for
                    if !flag
ResultMatrix[i][j]=0
       End for
     End for
          forI=Row to matrixsize2.length
               for   j=Column to matrixsize2.length
               End for
          End for
     ifResultMatrix[Row][Column] == 1
        return true
     else
        matrixsize1 = ResultMatrix
End for
        return false
EndCHECK-INDIRECT-DEPENDENCY ()
END
```

Similar to the above algorithms running time, we examined the fact that all of the rows of the given matrix have $\log n$ elements, each of which is either 0 or 1. We conducted a similar examination with the each column of the given matrix. For the Boolean matrix multiplication problem, we divided the complete matrix into rows and columns and each row (column) have $\log n$ elements. Therefore, here the complexity is $O\left(\frac{1}{logn}\right)$. The first for loop of the algorithm calculates the number of multiplications (number of intermediate nodes from one task to another) and it is having $O(n^2)$ complexity. Hence, the overall running time here is $O(n^3/logn)$.

This traversing of DAG gives us a set of independent or dependent tasks. Furthermore, this set adjoins the queue of sets that works as a dispatcher. The purpose of a dispatcher is to discharge the tasks to the nodes. Task sets that come in front execute in parallel on different processors and the next set is dependent on that previous set. This operation dispatches sets one by one, so, that it is taking $O(1)$ time.

Input: Independent or dependent tasks added into a queue.

Output: Dispatch tasks for execution.

QUEUE<SET<STRING>>QUEUEOFSET ()

BEGIN

1. Set<String> s = Independenttaskset()
2. **if** (setqueue.isEmpty())
3. setqueue.add(s)
4. return setqueue

End **QUEUEOFSET ()**

QUEUE<SET<STRING>>TASKEXECUTION ()

1. Queue<Set<String>> q = queueofset()
2. **while** (q.iterator().hasNext())
3. Taskexecution(q.element())
4. return setqueue

End **TASKEXECUTION ()**

END

In the above function queueofset(), add the returned set of independent tasks and other function dispatches with the sets for execution. The tracing and dispatching of the tasks of DAG takes $O(n^3/logn)$ time in total.

3.4 Assignment without Duplication

The previous module is the actual backbone of the complete simulation experiment. The dispatcher dispatches the independent tasks to the nodes and the execution of the project will

continue in the assigned processor.

Fig. 5 shows the clusters of processors and Table 2 represents the computation cost of processors with respect to the tasks. The execution of tasks from the dispatcher depends on their priorities. Here, the queue for a set of tasks has been maintained, which follows FIFO criteria.

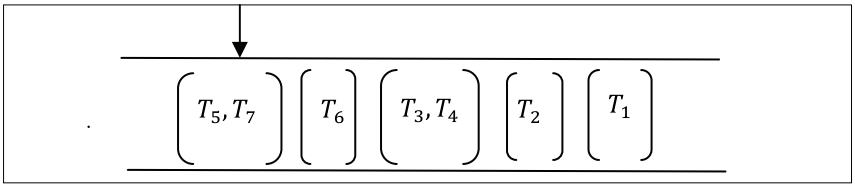


Fig. 9. Dispatcher Queue (FIFO)

The above dispatcher works on every processor separately. T_5, T_7 tasks will execute in parallel on different processors. Now T_6 is dependent on T_7, after getting the result from T_7 , T_6 assign to the other processor. T_3, T_4 are dependent on T_6. After getting the output from T_6; $T_3 and\ T_4$ can execute in parallel. Now T_2 requires output from $T_5, T_3\ and\ T_4$. Finally T_1executes on its own processor (source).

From Fig. 7 we notice that random tasks generate on four different processors that have different efficiencies. Let us take the DAG explained above, which is generated on a high efficiency processor. In arbitrary assignment heuristic algorithms, the dispatcher assigns tasks to other processors randomly. If the neighbor node is unable to execute extra given tasks, then the given task will switch to another processor.

3.5 Duplication Scheduling Explanation

This section explains the proposed duplication strategy that helps in reducing the schedule length of DAG. After the generation of random DAGs on particular nodes, its computational capability (efficiency) and communication cost of other processors is calculated (as shown in module B). We used the bottom-up approach of DAG. By using it we have designed a dispatcher queue that first allocates the processor to the first set of independent tasks. Those assigned independent tasks can execute in parallel on allocated processors. After the execution of these assigned tasks, the processor of dependent tasks starts with the implementation, because the output of the predecessor becomes the input for its successor.

Now, for the execution of such dependent task, task duplication is used. Our duplication approach is based on the following factors:

1. Communication cost: The time taken in the resettlement of the predecessor output towards its successor is the communication cost between them. If this data transfer rate is high then there is a requirement for the duplication of tasks to occur.
2. Computation cost: The time occupied by a processor to execute the specified task is the computation cost of the assigned tasks of the allotted processor.

In order to execute our approach, we first set the computation costs of a particular task (let us say $task_i$) on all of the processors in ascending order. Additionally, the communication costs between $task_i$ and its successors were arranged in descending order. Afterwards, the scheduler compares the successors computation cost in the source processor of $task_i$ and the

communication cost between tasks. If the computation cost is smaller than the $C_{task_i,successors}$ then the duplication of a successor task in the source processor of $task_i$ is achievable. Carrying out duplication in this way, along with the bottom-up approach, also decreases the number of duplications. The algorithm shown below explains the conditional duplication of our approach.

Input: The task with the Execution Time (ET) and communication cost (C_{t_i,t_j}) between connected tasks.
Output: Duplicate tasks to the Destination Processor (DP).

BEGIN

1. **IF** ($ET < C_{t_i,t_j}$)
2. DUPLICATE (T_i, DP)
3. **ELSE**
4. setqueue. TASKEXECUTION (T_i, P)

END

During the simulation of this duplication algorithm, we suspected that the number of processors affects the schedule length of the complete DAG with or without duplication. In it, we simulated one common DAG on two different distributed systems with or without duplication. The schedule length of DAG varies from the number of processors. We checked it for 5 and 10 processors.

Theorem 1: If we increase the number of processors in any distributed system then, will there be a need for task duplication?

Explanation: The addition of any processor in a system means accumulation of new computational power in the same. We can say that if we are increasing the number of processing powers, then schedule length of DAG should be small even without duplication.

Let us assume that the following common DAG and two different pairs of distributed systems exist. One system is a group of 5 processors. The other system is a group of 10 processors.

Fig. 10 explains the computation costs of tasks on the given processors of the system. This theorem explains the relationship between task duplication and schedule length. In order to establish the relationship between both, we have considered the two examples listed below.

1. A smaller number of processors with or without duplication:

Fig. 10(b) is a system of 5 processors with general computational capacity. If we execute the given DAG (Figure 10 (a)) on this system by using duplication, the overall schedule length of DAG is comparatively low (as shown in Fig. 11).

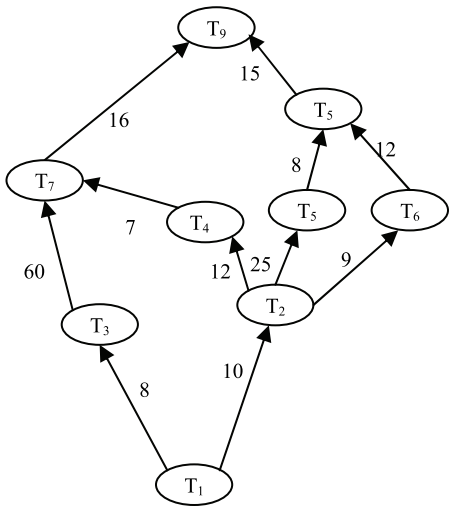
2. A greater number of processors with or without duplication:

After implementing a small system, we expanded the given system by the addition of 5 supplementary processors to increase computational capacity. Following the execution of the same DAG on this new arrangement, we again figured out that by using duplication, the schedule length of the DAG is less.

For task duplication we have used following criteria:

If ($ET < C_{t_i,t_j}$) then the duplication of a task occurs, but if the reverse occurs, then there is no need for duplication.

Other side of the coin is that when we increased the limit of processors by 5, then the DAG schedule length is increased as compared to the 5-processor system. Consequently, we cannot say that the schedule length is dependent upon the size of the system. By increasing the number of processors, the overall schedule length may or may not be reduced without duplication. The reason behind this is that the execution of a task is dependent on the computational capacity of any processor of the system and the usage of duplication is the best way to shorten the schedule distance.



(a)

P / T	P_1	P_2	P_3	P_4	P_5
T_1	25	10	6	24	12
T_2	18	15	16	20	10
T_3	9	11	8	16	14
T_4	8	9	7	10	3
T_5	12	13	15	10	8
T_6	7	13	15	12	11
T_7	19	15	20	12	10
T_8	13	20	17	14	12
T_9	16	11	15	10	12

(b)

P / T	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
T_1	25	10	6	24	12	15	11	6	10	14
T_2	18	15	16	20	10	21	17	19	14	16
T_3	9	11	8	16	14	15	13	7	10	9
T_4	8	9	7	10	3	9	6	11	4	5
T_5	12	13	15	10	8	7	11	9	14	16
T_6	7	13	15	12	11	8	14	16	13	15
T_7	19	15	20	12	10	21	16	18	11	17
T_8	13	20	17	14	12	14	21	16	13	15
T_9	16	11	15	10	12	17	10	14	11	9

(c)

Fig. 10. (a) Arbitrary DAG (b) Distributed System of 5 (c) 10 Processors

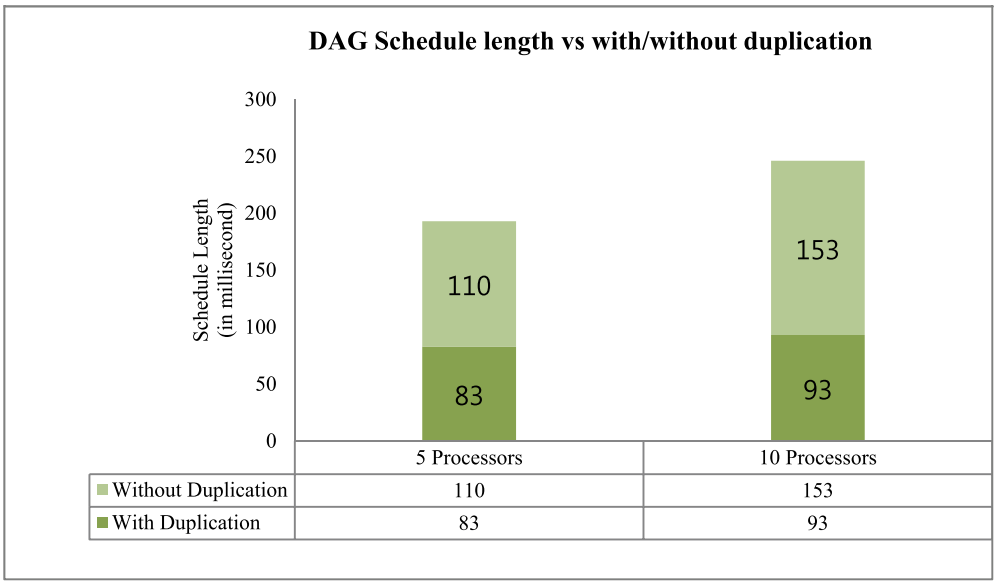


Fig. 11. Schedule Length vs. DAG Execution With or Without Duplication

4. Results and Comparisons

Here, the proposed algorithm for task duplication in heterogeneous systems with mesh topology was simulated. Simulation results for the bottom-up approach of random DAGs show that the makespan generated by the proposed algorithm is better than the existing compared algorithms i.e. arbitrary task assignment, CAWF and HEFT-TD algorithms. The concept of task duplication is used in the task assignment heuristic in mesh topology. Our proposed algorithm is named as Task Duplication Assisted Schedule Length Minimization Algorithm (TDASLM). The given example and simulations performed explain that the TFT can be cut down by reducing the communication cost because of duplication and that it can be done so by using optimal assignment (the communication cost must be greater than the execution time of related tasks on that processor).

4.1 Experimental Set-up and Test Bed

1. Topology

In a distributed system, the connectivity architecture that follows the processors of the entire system is known as topology. Some basic topologies that are followed by any network/ distributed system are bus, ring, star, and mesh. The implementation of bus, ring and star topologies are simpler when compared to mesh topology. In mesh topology each processor is associated with every other processor of the system. Due to the connectivity complexity of mesh topology, its handling is difficult to enforce. We simulated fully connected mesh topology in our proposed work.

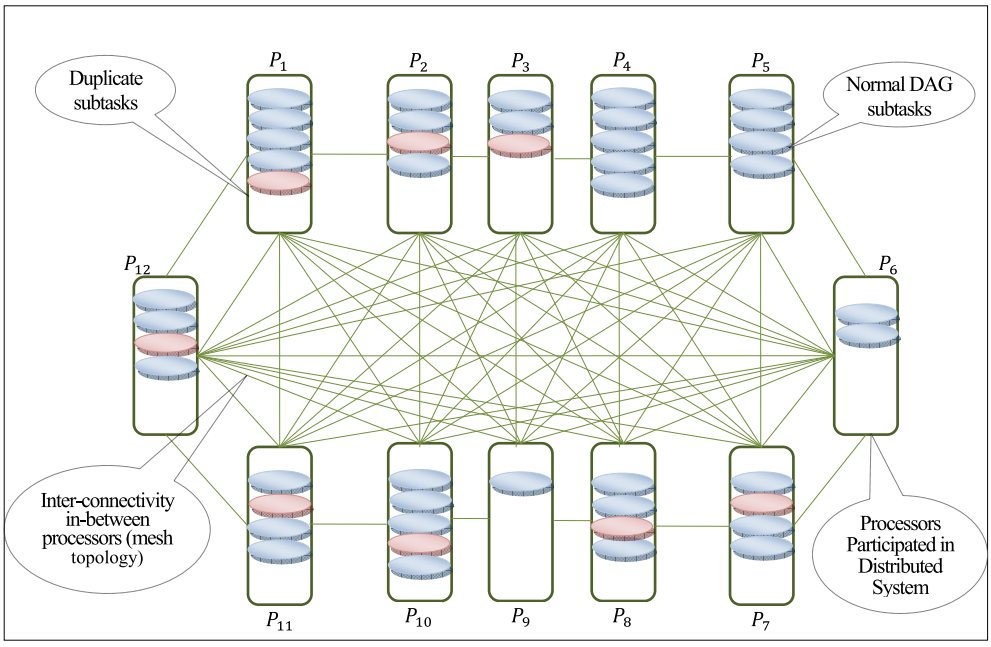


Fig. 12. explains the experimental setup of our proposed study. Listed below are some attributes that explain how the given set-up functions

2. Participating processors

Participating processors are the processors that belong to the distributed system. The participation of the processors devises an environment for the system that determines the overall performance of the system. Here, heterogeneous processors are utilized in this simulation. The term heterogeneous means that each processor of the system shares different architecture. Internal storage capacity and computational power are the main components of any architecture. Here every processor has different computational capacities. Hence, we have used a clustering method that splits the entire system into three clusters (i.e. Low, Medium, and High). All clusters have some fixed range of computational efficiency (Fig. 5).

3. Normal DAG sub-tasks

The proposed duplication algorithm works on DAG. As we discussed in the previous section, independent tasks will execute in parallel on different processors. Those assigned tasks behave like normal executable tasks.

4. Duplicate sub-tasks

We have divided the entire DAG into dependent or independent tasks. Duplication method is used for the decrease in communication cost between dependent tasks. There are various methods for task duplication, but the way of conducting processor selection for the execution of duplicate tasks/subtasks varies. In our method, we compared the computation and communication cost of the duplicated task in the destination processor. If the computation cost of processor is greater than the communication cost, then there is no requirement for duplication to occur.

These above techniques and all components of the framework are implemented in the Netbeans 6.9 IDE environment that runs on Ubuntu Version 11.10. We periodically generated random DAGs on any processor. The matrix is used to execute the DAG and queue data structure and it has been used to implement the dispatcher. Java threads are used to execute and communicate sub-tasks with each other. We continuously ran up to 100 DAG upto 30 times on 12 and 16 processors to compute the overall schedule length of DAG. We simulated our duplication algorithm along with the CAWF, arbitrary task assignment heuristics and HEFT-TD on the above designed framework.

Our proposed algorithm is the reproduction of HEFT-TD [11], but we have implemented it by using mesh topology and the bottom-up approach. Therefore, its complexity is a little bit higher.

Table 3. The Algorithmic Complexity of the Existing Duplication DBUS, HEFT-TD, and Proposed TDASLM Algorithms

Duplication Algorithms	Complexity
DBUS	$O(\|n^2\|\|P^2\|)$
HEFT-TD	$O(\|V^2\|(p+d))$
TDASLM(Proposed Algorithm)	$O(\frac{n^3}{logn})$

Mesh topology is good for a limited number of processors. As processors increase the connections between them also increase due to which system becomes more complex. It is the limitation of our algorithm that this algorithm is finer for inadequate size of distributed system.

4.2 Comparisons

4.2.1 Schedule Length

The schedule length (TFT) of DAG is computed by using Equation (3). The TFT of DAG without duplication (arbitrary processor selection method) is very high as compared to CAWF, where schedule length is decreased by cutting down the communication cost in-between tasks. When we used duplication, the resultant schedule length was very low as compared to CAWF and the arbitrary method, as well. The HEFT-TD method uses the top-down approach in DAG traversing and the proposed algorithm employs a bottom up approach. Therefore, the schedule length of our proposed algorithm gives similar or slightly better results than the other two algorithms.

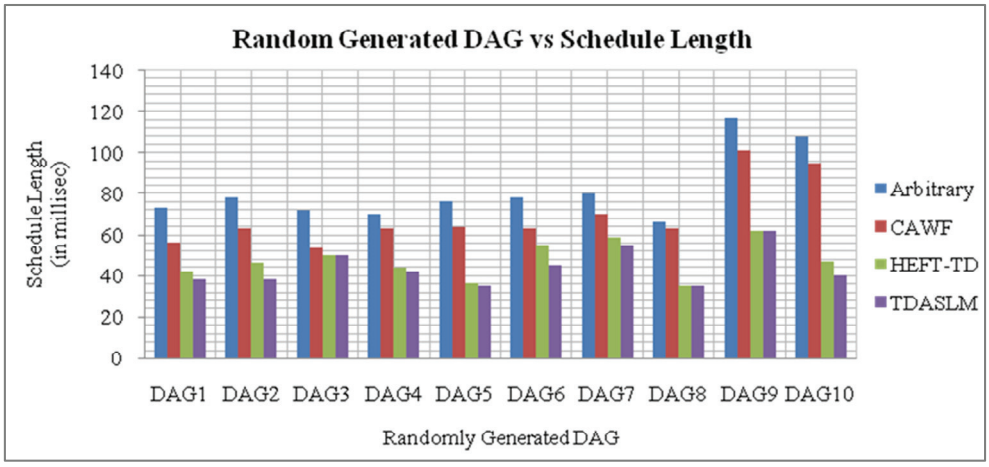


Fig. 13. Comparison of the Proposed Algorithm with Existing Assignment Algorithms

4.2.2 Computation to Communication Ratio (CCR):

The Computation to Communication Ratio (CCR) is the ratio of the number of calculations a process does to the total size of the messages it sends. This ratio depends upon the average communication volume and average task execution weight. The speed of the communication channel also affects the CCR and this speed depends on the computational speed of processors. In this paper, we used heterogeneous processors that had different computational speeds. It comes under mesh topology, so a high processing power processor connecting with a low processing power processor and vice versa is possible. Therefore, if any data moves from the higher efficiency processor to a less efficient processor and the speed of the communication channel is very fast, then the CCR will be higher. However, if the speed of the channel is high and the computational cost of processor is very low, then CCR will once again be affected. Therefore, CCR varies with both processor speeds, as well as with the communications channel, because we used mesh topology with heterogeneous processors here.

5. CONCLUSIONSAND FUTURE WORK

We employed the task duplication concept during the assignment procedure (before the implementation of tasks). This duplication reduced the total finish time of a task. According to Theorem1, we also explained that the TFT (schedule length) of a task is wholly dependent upon the execution power of the processor and if we apply duplication, then it will generate good results.This task duplication can overload a processor, in order to overcome the overload. In the future, we will extend this algorithm with task migration in the distributed or Real Time Distributed System (RTDS).

REFERENCES

[1] L. N. Bhuyan and D. P. Agrawal, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Transactions on Computers*, vol. C-33, no. 4, pp. 323-333, 1984.

[2] D. Bozdağ, U. Catalyurek, and F. Özgüner, "A task duplication based bottom-up scheduling algorithm for heterogeneous environments," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, 2006.

[3] Y. Jégou, "Runtime support for task migration on distributed memory architectures," in *Proceedings of the 11th International Conference on Parallel Processing Symposium*, Geneva, Switzerland, 1997.

[4] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia, "Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, 2000, pp. 25-33.

[5] A. Burchard, J. Liebeherr, O. Yingfeng, and S. H. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1429-1442, 1995.

[6] C. Wang, "Dynamic voltage scaling for priority-driven scheduled distributed real-time systems," Ph.D. dissertation, University of Kentucky, Lexington, KY, 2007.

[7] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: a survey," in *Approximation Algorithms for NP-Hard Problems*, D. S. Hochbaum, Ed. Boston, MA: PWS Pub. Co., 1997, pp. 46-93.

[8] A. Bashiry, S. A. Madaniy, J. H. Kazmiy, and K. Qureshix, "Task partitioning and load balancing strategy for matrix applications on distributed system," *Journal of Computers*, vol. 8, no. 3, pp. 576-584, 2013.

[9] J. Baxter and J. H. Patel, "The LAST algorithm: a heuristic-based static task allocation algorithm," in *Proceedings of the 1989 International Conference on Parallel Processing*, University Park, PA, 1989, pp. 217-222.

[10] B. Kruatrachue and T. Lewis, "Grain size determination for parallel processing," *IEEE Software*, vol. 5, no. 1, pp. 23-32, 1988.

[11] P. Chaudhuri and J. Elcock, "Process scheduling in heterogeneous multiprocessor systems using task duplication," *International Journal of Business Data Communications and Networking*, vol. 6, no. 1, pp. 58-69, 2010.

[12] S. Ranaweera and D. P. Agrawal, "A task duplication based scheduling algorithm for heterogeneous systems," in *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, Cancun, Mexico, 2000, pp. 445-450.

[13] S. Ranaweera and D. P. Agrawal, "A scalable task duplication based scheduling algorithm for heterogeneous systems," in *Proceedings of the International Conference on Parallel Processing*, Toronto, Canada, 2000, pp. 383-390.

[14] J. Singh and H. Singh, "Efficient tasks scheduling for heterogeneous multiprocessor using genetic algorithm with node duplication," *Indian Journal of Computer Science and Engineering*, vol. 2, no. 3, pp. 402-410, 2011.

[15] R. Sharma and N. Nitin, "Duplication with task assignment in mesh distributed system," in *World Congress on Information and Communication Technologies*, Mumbai, India, 2011, pp. 672-676.

[16] E. G. Coffman, Jr., G. Galambos, S. Martello, and D. Vigo, "Bin packing approximation algorithms: combinatorial analysis," in *Handbook of Combinatorial Optimization*, D. Du and P. M. Pardalos, Eds. Boston, MA: Kluwer Academic Publishers, 1998, pp. 151-207.

[17] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1384-1397, 1988.

APPENDIX

```
//Here is the main code for DAG tracing:

public static void main(String[] args) throws DAGCycleException
      {
CountOneFromMatrixcofm=new CountOneFromMatrix(taskDAG);
ArrayList<SortedTask>taskList= cofm.getColumnOneCountInSorted();
for(int i=0;i<taskList.size();i++)
          {
SortedTask t=taskList.get(i);
System.out.println("The       taskid="+t.getTaskid()+"and dependency="+t.getDependency());
          }
TaskCatagorization taskCatagorization=new TaskCatagorization();
taskCatagorization.setSortedTasks(taskList);
taskCatagorization.setTaskDAG(taskDAG);
taskCatagorization.constructTaskQueue();
      }
}

// Following code explains how we set the efficiency and communication cost

public static void main(String[] args) throws InterruptedException
      {
final Group g=new Group();
            Runnable r=new Runnable() {
public void run()
                  {
for(inti=0;i<8;i++)
                  {
for(int j=0;j<8;j++)
                        {
                              Calendar c=Calendar.getInstance();
long m=c.getTimeInMillis();
                               Random r=new Random(m*i*j);
int x=Math.abs(r.nextInt() %10);
g.setEfficiency(i, j, x);
                        }
                  }
                }
            };
          Thread t=new Thread(r);
t.start();
NewJFramejFrame=new NewJFrame(g);
jFrame.setTitle("Distributed Computing Simulation");
jFrame.setVisible(true);
jFrame.setSize(800, 800);
jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jFrame.setLocation(200, 50);
jFrame.setGroup(g);
}
```

Rashmi Sharma

She holds M.Sc (2008), M.Tech (2010) in computer science from Banasthali University, Rajasthan (India). Currently pursuing Ph.D in the field of Real Time Distributed system in computer science from Jaypee University of Information Technology, Waknaghat, Solan(H.P, India).

Dr. Nitin

He is Ex First Tier Bank Professor, University of Nebraska at Omaha, NE, USA. His permanent affiliation is with Jaypee University of Information Technology (JUIT), Waknaghat, Solan-173234, Himachal Pradesh, INDIA as a Associate Professor in the Department of Computer Science & Engineering and Information & Communication Technology. He joined Jaypee University of Information Technology in July 2003. He was born on October 06, 1978, in New Delhi, INDIA.

In July 2001, he received the B.Engg. in Computer Science & Engineering [Hons.] and M.Engg. in Software Engineering from Thapar Institute of Engineering and Technology, Patiala, Punjab, INDIA in March 2003. In 2008, he received his Ph.D. in Computer Science & Engineering from JUIT, INDIA. He has completed his Ph.D. course work from University of Florida, Gainesville, FL, USA.

He is a IBM certified engineer. He is a Life Member of IAENG, Senior Member IACSIT and Member of SIAM, IEEE and ACIS and has 121 research papers in peer reviewed International Journals & Transactions, Book Chapters, Symposium, Conferences and Position. His research interest includes Social Networks especially Computer Mediated Communications & Flaming, Interconnection Networks & Architecture, Fault-tolerance & Reliability, Networks-on-Chip, Systems-on-Chip, and Networks-in-Packages, Application of Stable Matching Problems, Stochastic Communication and Sensor Networks. Currently he is working on Parallel Simulation tools, BigSim using Charm++, NS-2 using TCL. He is referee for the Journal of Parallel and Distributed Computing, Elsevier Sciences, Computer Communications, Elsevier Sciences, Computers and Electrical Engineering, Elsevier Sciences, Mathematical and Computer Modelling, Elsevier Sciences. WSEAS Transactions, The Journal of Supercomputing, Springer and International Journal of System Science, Taylor &Francis.