1. INTRODUCTION

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

2. BACKGROUND AND RELATED WORK

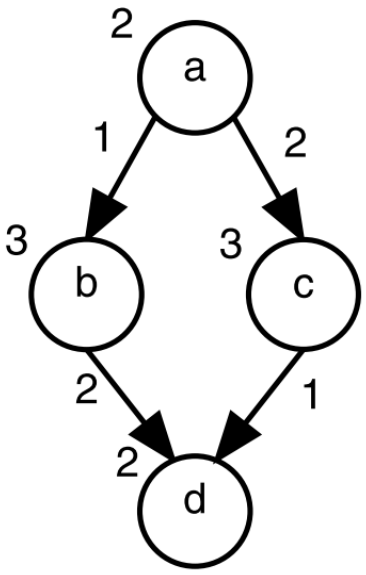
BASIC ASSUMPTIONS

The following classical assumptions are made about the target system

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

TASK GRAPH

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



TASK SCHEDULING MODEL

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

RELATED WORK

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

3. AO MODEL

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

ALLOCATION PHASE

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

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

The function for Communication cost is as follows:

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

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

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

ORDERING PHASE

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

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

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

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

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

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

4. PRUNING TECHNIQUES

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

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

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

NOVEL PRUNING TECHNIQUES

GRAPH REVERSAL

After observing research papers and

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

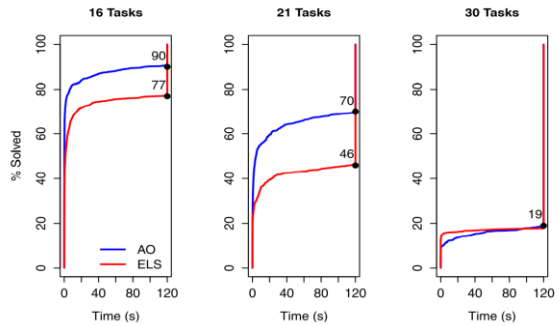
5. EVALUATION

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

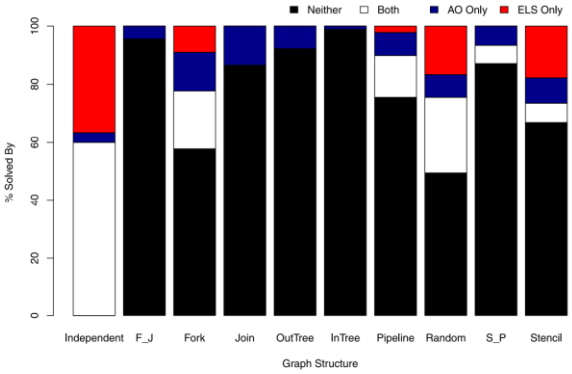
Range of task graphs in the experimental data set.

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

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



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



(c) 30 Tasks

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

6. CONCLUSION

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

CODE FOR ELS

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

using namespace std;

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

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

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

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

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



void ELS(){

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

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



    }

int heuristic(){

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

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

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

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

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



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


    // Neighbors of current node
```

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

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

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




}

};

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

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

bool dulplicates(state s1,state s2){

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




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

```
            }

          }
        }
        return true;

}


// Functor for comparision in fork graph maintaining ascending order

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

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

// Functor for maintaining descending order in join graph

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

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




void pruning(){

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

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

    // if graph is join
```

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




}

map<int,int> computation_cost;
 int no_of_processors;

int main() {

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

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


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

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

```
    */



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


    // Finding no. of free tasks



    return 0;
}
```

CODE FOR AO

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

using namespace std;

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

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

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

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

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



}

int heuristic(){



}

};

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

    }

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

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

  }
};

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

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

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

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

}



map<int,int> computation_cost;
 int no_of_processors;


int main() {

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

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


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

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

```
    }
    */



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


    // Finding no. of free tasks



    return 0;
}
```