Journal of Engineering Science and Technology
Vol. 5, No. 3 (2010) 332 - 341
© School of Engineering, Taylor's University College

SCALABLE LOW COMPLEXITY TASK SCHEDULING ALGORITHM FOR CLUSTER OF WORKSTATIONS

S. PADMAVATHI*, S. MERCY SHALINIE

Department of Computer Science and Engineering,
Thiagarajar College of Engineering, Madurai-625 015, TamilNadu, India
*Corresponding Author: spmcse@tce.edu

Abstract

Static scheduling is the temporal and spatial mapping of a program to the resources of parallel system. Scheduling algorithms use the Directed Cyclic Graph (DAG) to represent sub-tasks and the precedence constraints of the program to be parallelized. It has been shown to be NP-Complete in general as well as in several restricted cases. This paper proposes a scalable, low complexity performance effective task scheduling algorithm whose time complexity is $O((e)\ (p + \log v))$. It provides effective result for applications represented by DAGs. Experiments have been conducted based on randomly generated graphs which show that the proposed algorithm outperforms the existing algorithms.

Keywords: DAG, Task graph, Cluster computing system, Scheduling scalability, Speedup, Efficiency.

1. Introduction

Many cluster computing application like weather modeling, image processing, real-time and distributed database systems require efficient scheduling of tasks. Parallel execution can improve overall efficiency. To take the full advantage of the computing power provided by these systems, an easy and efficient scheduling of tasks is required. Program performance critically depends on the partitioning of the program and the scheduling of the resulting tasks onto the physical processors. Task scheduling algorithm takes care of above said process and the application is represented by a Directed Cyclic Graph (DAG) which can be performed at compile-time or run-time. The objective of this paper is to map the tasks on the processors, order their execution so that task precedence requirements are satisfied and a minimum overall completion time is obtained. It has been proven to be NP- Complete [1-3], for which optimal solutions can be found only after an exhaustive search [4, 5].

Nomenclatures

E	Set of directed edges
e	Number of edges
L	Schedule length
O	Time complexity
p	Number of processors
t	Running time
V	Set of tasks
v	Task

Abbreviations

AFT	Actual finish time of task
AST	Actual start time of task
CC	Computation cost
CCR	Communication-to-computation ratio
CPOP	Critical path on a processor
DAG	Directed cyclic graph
DRC	Data receiving cost
DTC	Data transfer cost
EFT	Earliest finish time of task
EST	Earliest start time of task
LMT	Levelized min time
NSL	Normalized schedule length
SLETS	Scalable low complexity, effective task scheduling
SS	Scheduling scalability

Efficient application scheduling is critical for achieving high performance in cluster computing systems. Because of its key importance on performance, the task scheduling problem in general has been extensively studied and various heuristics have been proposed [1-16]. These heuristics are classified into a variety of categories such as list scheduling algorithms, clustering algorithms, guided random search methods and task duplication based algorithms. In list scheduling algorithms [8 and 14] ordered lists of tasks was constructed by assigning priority to each task and are selected based on their priority. List scheduling is generally preferred since they are generating good quality schedules with less complexity. List scheduling algorithms such as Mapping Heuristics (MH) [8], Levelized Min Time (LMT) [10], Critical Path on a Processor (CPOP) [14], are well known scheduling algorithm. Clustering algorithms [11, 14] also known as three phase scheduling. It tries to schedule heavily communicating tasks onto the processor, even if other processors are available, thereby trading off parallelism with interprocess communication.

Among the various scheduling algorithms, list scheduling algorithms are generally preferred for task scheduling, since they produce good schedule with less time. But, the reported scheduling algorithms like LMT, CPOP are complex in nature and take higher complexity. Moreover the LMT algorithm does not utilize the earliest idle time slot between the scheduled tasks on a processor. Because of this, the schedule length generated by the LMT is not always minimal. The rank of the task in the CPOP algorithm is calculated in the reverse fashion i.e., traversing the task graph downwards from the entry task. The rank of a task is

the length of the critical path from the entry task to itself. The rank computation is recursive procedure and also complex. The motivation behind this work is to develop a new task scheduling algorithm which deliver a high performance, low complexity and scalable on cluster of workstations. A new algorithm known as scalable low complexity task scheduling algorithm is proposed which gives best performance in terms of speedup, efficiency with low complexity.

This paper is organized as follows: The Scheduling system model is described in Section 2. Proposed Algorithm for task scheduling is explained in Section 3. The results and discussions are presented in Section 4 and the conclusions are presented in Section 5.

2. Task Scheduling Problems

A scheduling system model consists of an application, a target computing system and Criteria for scheduling. An application program is represented by a DAG, $G = (V,<,E)$, where $V=\{v_i, i=1...n\}$ is the set of n tasks, '$<$' represents a partial order on V. For any two tasks v_i and v_k the existence of the partial order $v_i < v_k$ means v_k cannot be scheduled until task v_i will be completed. Hence v_i is a predecessor of v_k and v_k is a successor of v_i. The task executions of a given application are assumed to be non-preemptive. E is the set of directed edges. Data is an $n \times n$ matrix of communication data, where $data_{i,k}$ is the amount of data required to be transmitted from tasks v_i to v_k. In Fig. 1 task graph, a task without any predecessor is called an *entry task* and a task without any child is called an *exit task*. Without loss of generality, it is assumed that there is one entry task to the DAG and one exit task from the DAG. In an actual implementation, a pseudo entry task and pseudo exit task can be created with zero computation time and communication time. A task graph is shown in Fig. 1, and its computation cost matrix is given in Table 1.

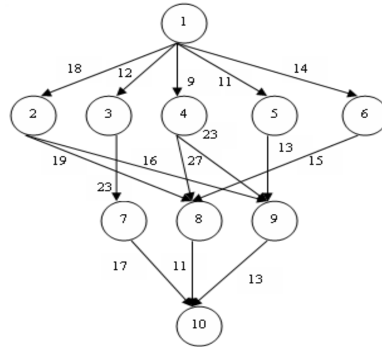

Fig. 1. Task Graph Represented by DAG.

Table 1. Computation Cost Matrix.

Task	Cost	Task	Cost
1	13	6	13
2	17	7	11
3	14	8	10
4	13	9	17
5	12	10	15

Cluster computing system consists of a set $P=\{p_j: j=0, m-1\}$ of 'm' processors fully interconnected by a high-speed arbitrary network. The data transfer rate is represented by an $m\times m$ matrix, $R_{m\times m}$. W is a $n\times m$ computation cost matrix in which each W_{ij} gives the *Estimated Computation Time* (ECT) to complete task v_i on processor p_j where $0 \leq i < n$ and $1 \leq j \leq m$. The communication cost between two processors p_x and p_y, depends on the channel initialization at both sender processor p_x and receiver processor p_y in addition to the communication time on the channel. This is a dominant factor and can be assumed to be independent of the source and destination processors. The channel initialization time is assumed to be negligible. The communication cost of the *edge* (i,k), is for transferring data from task v_i (scheduled on processor p_x) to task v_k (scheduled on processor p_y) is defined by Eq. (1)

$$C_{i,k}= data_{i,k}/R_{xy} \quad (1)$$

$C_{i,k} = 0$ when both the tasks v_i and v_k scheduled on the same processor. Assuming the data transfer rate for each link as 1.0, the communication cost and amount of data to be transferred will be the same. The communication-to-computation ratio (CCR) of a parallel program is defined as its average communication cost divided by its average computation cost on a given system.

Let $EST(v_i, p_j)$ and $EFT(v_i, p_j)$ are the Earliest Start Time and Earliest Finish Time of task v_i on p_j, respectively. For the entry task v_{entry}, $EST(v_{entry}, p_j) = 0$, and for the other tasks, the *EST* and *EFT* values are computed recursively using Eqs. (2) and (3) starting from the entry task. In order to compute the *EFT* of a task v_i, all immediate predecessor tasks of v_i must have been scheduled.

$$EST(v_i,p_j) = \max\{avail[j], \max(AFT(v_t + C_{t,i}))\} \quad (2)$$

where $v_t \in pred(v_i)$

$$EFT(v_i,p_j) = Wij + EST(v_i,p_j) \quad (3)$$

where $pred(v_i)$ is the set of immediate predecessor tasks of task v_i and $avail[j]$ is the earliest time at which processor p_j is ready for task execution. If v_k is the last assigned task on processor p_j, then $avail[j]$ is the time that processor p_j completed the execution of the task v_k and it is ready to execute another task if it is based on non-insertion scheduling policy. The inner max block in the *EST* equation returns the ready time, i.e., the time when all the data needed by v_i has arrived at processor p_j.

After a task v_t is scheduled on a processor p_j, the earliest start time and the earliest finish time of v_t on processor p_j is equal to the actual start time $AST(v_t)$ and the actual finish time $AFT(v_t)$ of task v_t, respectively. After all tasks in a graph are scheduled, the schedule length (i.e., the overall completion time) will be the actual finish time of the exit task v_{exit}. Finally the schedule length is defined in Eq. (4)

$$\text{Schedule Length} = \max\{AFT(v_{exit})\} \quad (4)$$

The objective function of the task-scheduling problem is to schedule the tasks of an application to machines such that its schedule length will be minimized.

3. Scalable Low complexity, Effective Task Scheduling (SLETS) Algorithm

The proposed algorithm consists of three phases; level sorting, task prioritization and processor selection. The detailed explanation of each phase of the algorithm is given below:

3.1. Level sorting phase

In the level sorting phase, the given DAG is traversed in a top-down fashion to sort task at each level in order to group the tasks that are independent of each other. As a result, tasks in the same level can be executed in parallel. In a given DAG, level 0 contain entry task. Level i consist of all tasks v_k such that, for all *edges*(v_j,v_k), task v_j is in a level less than i and there exists at least one *edge*(v_j,v_k) such that v_j is in level i-1. The last level comprises of some of the exit tasks. For the task graph given in Fig. 1, there are 4 levels; *level 1* consists of task 1 (*entry task*), *level 2* consists of task 2, 3, 4, 5 and 6, *level 3* consists of task 7, 8 and 9, and *level 4* consists of task 10 (*exit task*).

3.2. Task prioritization phase

In the task prioritization phase, priority is computed and assigned to each task. For assigning priority to a task, three attributes were defined namely, *Computation Cost* (*CC*), *Data Transfer Cost* (*DTC*) and the *Data Receiving Cost* (*DRC*). The *CC* of a task v_j is the computation cost on any processor and is computed by the $C_{j,k}$ of any processor p_k.

DTC of a task is the amount of cost incurred to transfer the data to all its immediate successor tasks. The *DTC* for the exit task is 0, for all other tasks at level i, computed using Eq. (5)

$$DTC(v_i) = \sum_{i=1}^{x} C_{i,j} \tag{5}$$

where x is the number of immediate successors of v_i.

DRC of a task is the cost incurred to receive data from its immediate predecessor tasks. The *DRC* for the entry task is 0 and for all other tasks at level i, it is computed using Eq. (6)

$$DRC(v_j) = \max\{rank(v_i)\} \tag{6}$$

where v_i is the predecessor of v_j.

Rank of a task v_i($rank(v_i)$) is the sum of its *DTC, DRC* and *CC* values of that task. For the every task at each level i, $rank(v_i)$ is computed using Eq. (7)

$$rank(v_i) = DTC(v_i) + DRC(v_i) + CC(v_i) \tag{7}$$

At each level i, priority is assigned to all the tasks based on its rank value. The task with highest rank value receives the highest priority followed by task with next highest rank value and so on is shown in Table 2.

Table 2. The Computed CC, DTC, DRC, Rank and Priority for Task Graph in Fig. 1.

Level	Task	*CC*	*DTC*	*DRC*	Rank	Priority
1	1	13	64	0	77	1
2	2	17	35	77	129	2
2	3	14	23	77	114	3
2	4	13	50	77	140	1
2	5	12	13	77	102	5
2	6	13	15	77	105	4
3	7	11	17	114	142	6
3	8	10	11	140	161	2
3	9	17	13	140	170	1
4	10	15	0	170	185	1

3.3. Processor selection phase

In the processor selection phase, the processor, which gives minimum *EFT* for a task is selected and the task is assigned to that processor. It has an insertion-based policy, which considers the possible insertion of a task in an earliest idle time slot between two already scheduled tasks on a processor. At each level, the *EST* and *EFT* value of each task on every processor is computed using Eqs. (2) and (3).

The tasks are selected for execution based on their priority value. Task with highest priority is selected and scheduled on its favourite processor (processor which gives the minimum *EFT*) for execution followed by the next highest priority task. Similarly all the tasks in each level are scheduled on to the suitable processors is shown in Table 3. The proposed algorithm is given in Fig. 2.

```
1.Read the DAG, associated attributes values, and the number  of processor P;
2. For all tasks at each level Li do
3. Begin
4.  Compute DRC, DTC and CC;
5.  Compute rank(vk )=DTC(vk)+DRC(vK)+ CC(vK);
6.  Construct a priority queue using ranks;
7.  While there are unscheduled tasks in the queue
     do
8.   Begin
9.     Select the first task, vk  from the queue for scheduling;
10.    For each processor pk  in the processor set P
        do
11.       Begin
12.        Compute EFT value using insertion based
                scheduling policy;
13.        Assign the task vk to processor pk , which
                 minimizes the EFT;
14.        End;
15.   End;
16. End.
```

Fig. 2. The SLETS Algorithm.

The time complexity of other algorithm reported in literature is equal to $O((v+e)\ (p+log\ v))$, where v is the number of tasks, e is number of edges and p is nmber of processors. The breadth first search is used for level sorting whose time complexity is $O(e)$, since the implementation is done by adjacency lists. If the adjacency matrix is used for implementation, the complexity will be $O(v^2)$. A

binary heap was used to implement the priority queue, which has time complexity $O(log\ v)$. Each task in the priority queue is checked with all the p processors in order to select a processor that gives the earliest finish time. Hence the overall time complexity is $O(e)\ (p + log\ v)$.

Table 3. The Computed *EST, EFT* Values on Processors p_0, p_1, p_2, for the Tasks in Fig. 1.

Ordered tasks	Processors						Prede-cessor	Processor selected
	p_0		p_1		p_2			
	EST	*EFT*	*EST*	*EFT*	*EST*	*EFT*		
1	0	13	0	13	0	13	Null	P_0
4	22	35	22	35	22	35	1	P_0
2	35	52	31	48	31	48	1	P_1
3	35	49	48	62	25	39	1	P_2
6	35	48	48	61	39	52	1	P_0
5	48	60	48	60	39	51	1	P_2
9	64	81	64	81	64	81	2,4,5	P_0
8	81	91	81	91	63	73	2,4,6	P_2
7	73	84	81	92	73	84	3	P_0
10	94	109	94	109	94	109	7,8,9	P_0

4. Performance Analysis and Discussions

Randomly generated task graphs and the graphs that represent some of the numerical real world problems are the workload for testing the proposed algorithm. Cluster consists of HP P-IV processors with 32 nodes were used for experiments.

Performance metrics

The following four metrics were used to evaluate the performance of the proposed algorithm over different CCRs. The metrics are:

Speedup: The speedup is the ratio of the sequential execution time to the parallel execution time.

***Efficiency*:** The efficiency is the ratio of the speedup value to the number of processor used to schedule the graph.

Normalized Schedule length* (*NSL*)*: NSL is the ratio of the parallel time to the sum of weights of the critical path tasks. The main performance measure of an algorithm is the schedule length of its output schedule.

$$NSL = \frac{L}{\sum_{n\,\varepsilon CP} w(n_i)} \quad (8)$$

where L is the schedule length.

***Scheduling Scalability* (*SS*):** This metric is a collective indicator of whether an algorithm can scale its performance well for larger problem sizes

$$SS = \left(\frac{\sum_{i=1}^{v} w(n_i)}{L} \right) \cdot \left(\frac{v}{p} \right) \cdot \left(\frac{\log v}{\log t} \right) \tag{9}$$

where t is the running time of the algorithm. It is an indicator of its solution quality in terms of number of processors used and its running time in finding the solutions.

5. Experimental Results

The quality of schedules generated by the SLETS algorithm was evaluated for a large set of graphs with different CCRs in the range {0.1, 0.2, 0.5, 1, 2}and scheduled these task graphs on to a homogeneous cluster computing system consists of 32 nodes. Figure 3 represents the performance of the algorithm in terms of speed-up by varying the CCR values. The speed-up increases as the number of processors gets increased. Also as the CCR gets increased, (i.e., when the communication cost is more compared to the computation cost), the speed-up gets decreased. Another important performance factor for any type of scheduling is, Efficiency. Figure 4 represents the relation between efficiency of the algorithm and number of processors used.

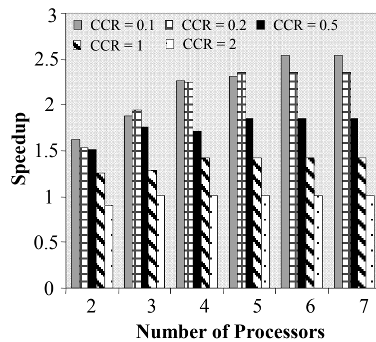

Fig. 3. Speedup vs. Number of Processors.

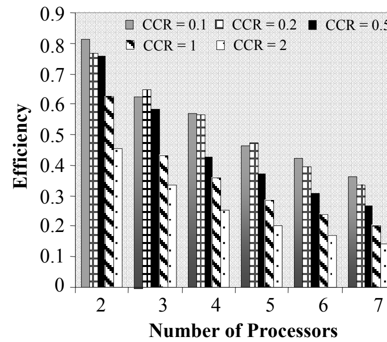

Fig. 4. Efficiency vs. Number of Processors.

Figure 5 represents the Normalized Schedule Length for DAGs with various CCR values. The Schedule Length goes on increasing when the average communication cost is very high compared to the average computation cost. NSL increases slightly when CCR increases from small to medium range. This indicates that when edge weights are large, then the algorithm can make more mistakes in scheduling the tasks. Figure 6 represents the Scheduling Scalability characteristics of DAGs with different CCR values. When 'p' value is low, the scalability factor is high. From the graphs, it is clear that for the applications

whose average communication cost is less than the average computation cost, the proposed algorithm outperforms in optimal scheduling.

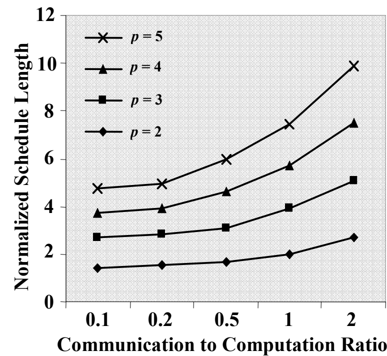

Fig. 5. Normalized Schedule Length vs. Communication to Computation Ratio.

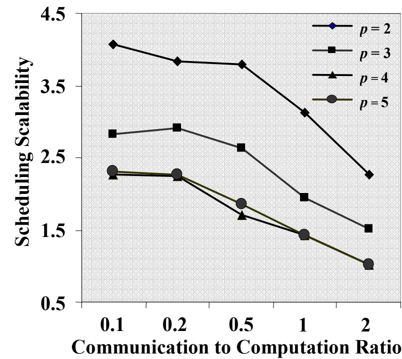

Fig. 6. Scheduling Scalability vs. Communication to Computation Ratio.

6. Conclusions

Tackling the scheduling problem in an efficient manner is imperative for achieving from message-passing parallel computer systems. The task scheduling algorithm SLETS proposed here has been proven to be better for scheduling DAG structured applications onto homogeneous cluster computing system in terms of scheduling scalability, Normalized schedule length, speedup and efficiency results. The performance of the SLETS has been observed experimentally by using large set of randomly generated task graphs with various characteristics. The simulation results confirm that SLETS is substantially better. The complexity of SLETS algorithm is $O((e)(p + \log v))$, which is less when compared to other scheduling algorithms reported. The scheduling accuracy can be still improved by using stochastic techniques and intelligent scheduling heuristics.

References

1. Ahmed, I.; and Kwok, Y.K. (1998). On exploiting task duplication in parallel program scheduling. *IEEE Trans on Parallel and Distributed Systems*, 9(9), 872-892.
2. Baskiyar, S.; and SaiRanga, P.C. (2003). Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length. *International Conference on Parallel Processing Workshop*, 97-103.
3. Braun, T.D.; Siegel, H.J.; Beck, N.; Boloni, L.L.; Maheswaran, M.; Reuther, A.I.; Robertson, J.P.; Theys, M.D.; Hensgen, B.Y.D.; and Freund, R.F. (1999). A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. *Proc. 8th Workshop on Heterogeneous Computing*, 15-29.

4. Casavant, T.L.; and Kuhl, J.G. (1988). Taxonomy of scheduling in general purpose distributed memory systems. *IEEE Transactions on Software Engineering*, 14(2), 141-154.
5. Graham, R.L.; Lawler, L.E.; Lenstra, J.K.; and Rinnooy Kan, A.H.G. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5, 287-326.
6. Boeres, C.; Filho, J.V.; and Rebello, V.E.F. (2004). A Cluster-based strategy for scheduling task on heterogeneous processors. *Proc. 16th Symp. On Computer Architecture and High Performance Computing*, 214-221.
7. Dhodhi, M.K.; Ahmad, I.; Yatama, A.; and Ahmad, I. (2002). An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 62(9), 1338-1361.
8. EI-Rewini, H.; and Lewis, T.G. (1990). Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2), 138-153.
9. Hui, C.C.; and Chanson, S.T. (1997). Allocating task interaction graphs to processors in heterogeneous networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(9), 908-926.
10. Iverson, M.; Ozguner, F.; and Follen, G.J. (1995). Parallelizing existing applications in a distributed heterogeneous environment. *Proceeding of 4th Heterogeneous Computing Workshop*, 93-100.
11. Kafil, M.; and Ahmed, I. (1998). Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency*, 6(3), 42-51.
12. Kim, S.C.; and Lee, S. (2005). Push-pull: guided search DAG scheduling for heterogeneous clusters. *Proceeding of International Conference on Parallel Processing*, 603-610.
13. Ranaweera, S.; and Agarwal, D.P. (2000). A task duplication based algorithm for heterogeneous systems. *Proceeding of 14th International Parallel and Distributed Processing Symposium*, 445-450.
14. Topcuoglu, H.; Hariri, S.; and Wu, M.Y. (2002). Performance effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(2), 260-274.
15. Wang, L.; Siegel, H.J.; Rowchoudhry, V.P.; and Maciejewski, A.A. (1997). Task matching and scheduling in heterogeneous computing environments using a genetic algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1), 8-22.
16. Kwok, Y.K.; and Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4), 406-471.