PROJECT REPORT

SCHEDULING ALGORITHM FOR PARALLEL AND DISTRIBUTED SYSTEMS

Date : Friday, May 04, 2012

CSE 202
ALGORITHM DESIGN & ANALYSIS

Submitted by:

SAURABH JHA
10BCE1089
ACKNOWLEDGEMENT

I wish to express my sincere gratitude to Dr. L. Jeganathan for providing me the opportunity to study on the topic of my interest, scheduling algorithms for parallel and distributed computing.

I would also like to thank Devanjan Sikhdar for helping me understand some of the key concepts of the scheduling algorithms.
Problem Definition:

To schedule a bunch of inter-related activity (task) on to the parallel and distributed computers for high performance and high resource use.

Abstract:

To study various scheduling algorithms for parallel and distributed computing and to find out, if these existing algorithms can be modified for better performance. In this study I am mainly dealing with list and graph based scheduling approaches. In this paper I have first analysed some of the major techniques in this category. At the end of the paper I have proposed certain modifications for better performance of these algorithms. My study mainly focuses on two papers

1. Olga Rusanova and Alexandr Korochkin. Scheduling problems for parallel and distributed systems.

2. On the Complexity of List Scheduling Algorithms for Distributed-Memory Systems

Introduction

- Efficient scheduling is essential to obtain high performance from a parallel program. If the structure of the program and the target machine are known in advance, scheduling can be done automatically at compile time, thus saving considerable overhead at run-time.

- Task scheduling on distributed memory systems is a tradeoff between exploiting as much parallelism as possible while at the same time reducing communication, to minimize the parallel completion time of the program.
- Except for very restricted cases, the scheduling problem has been shown to be NP-complete. For realistic cases, scheduling is performed using heuristics. In order to be of practical use for large parallel applications, scheduling heuristics must have a low complexity. For shared-memory systems, it has been proven that even a low-cost scheduling heuristic is guaranteed to produce acceptable performance.

- In the distributed-memory case however, such a guarantee does not exist and the scheduling problem remains a challenge, especially for algorithms where low cost is of principal interest.

- Scheduling can be done either for a bounded or an unbounded number of processors. Although attractive from a cost perspective, scheduling for a unbounded number of processors is rarely of practical use, because the required number of processors is not usually available. Hence, their application is typically found within the multi-step scheduling method for a bounded number of processors. However, when compiling large programs for large systems, the complexity of current list scheduling approaches is still prohibitive.

**Task Representation**

- A parallel program can be modelled by a directed acyclic graph $G=(V,E)$ where $V$ is a set of nodes and $E$ is a set of edges.

- A node in the DAG represents a task, containing instructions that execute sequentially without pre-emption. Each task has a weight $p(t)$ associated with it, which represents the computation cost of executing the task. The edges correspond to task dependencies (communication messages or precedence constraints).

- Each edge has a weight associated with it, which represents the communication cost to satisfy the dependence. The communication to computation ratio (CCR) of a parallel program is defined as the ratio between its average communication cost and its average computation cost.
A task with no input edges is called an *entry* task, while a task with no output edges is called an *exit* task. A task is said to be *ready* if all its parents have finished their execution. A task can start its execution only after all its dependencies have been satisfied. If two tasks are mapped to the same processor, the communication cost between them is assumed to be zero.

As a distributed system, we assume a set of \( P \) processors connected in homogeneous clique topology. Inter-processor communication is performed without contention and tasks are executed without pre-emption. Once scheduled, a task \( t \) is associated with a processor has an *earliest start time* (\( TE \)) and a *latest execution time* (\( TL \)).

The objective of the scheduling problem is to find a scheduling of the tasks in \( V \) on the target processor such that the execution time is minimized.

**CLASSIFICATION OF STATIC SCHEDULING APPROACHES:**

**Optimal Scheduling Solutions**

1. Scheduling tree structured graph on \( n \) processor system
2. Scheduling arbitrary graph on 2 processor system
3. Scheduling interval order graph on \( n \) processor system

**Heuristic Sub Optimal Solutions**

**Genetic Algorithms:**

- This algorithm is applicable for easy problems.
- Disadvantage - When applied to problems where the ratio of the number of feasible solutions to the number of infeasible solutions is low, the genetic algorithm will act as a random search technique.
- Thus this algorithm is not suitable for scheduling purpose.
**Clustering:**

**Two approaches:**

- In the first approach, the tasks are assumed to be allocated to unlimited number of connected processors and then in group of tasks are merged and scheduled on particular number of processors.

- In the second approach, the tasks are assumed to be allocated to one processor for sequential computing. When more processors are available, some tasks are swapped out of the initial processor to other processors to do the computation in parallel and thus to reduce the computing time of the whole computation.

- Disadvantage – If there are large number of tasks and the number of processors available are less then this algorithm may fail.

**List Scheduling Technique:**

- Firstly we form the order of tasks according to their priorities. There are many approaches to determine priorities. Based on the priorities we assign tasks to processors.

- Assignment procedure can be performed with or without considering communication cost. So, list scheduling is the best suitable for distributed and parallel systems.

**Types of Communication Models:**

- **Model A** – In this model a processor cannot execute a task as well as communicate with another processor at the same time. Communication cost equals sum of edges weights in a given task graph.

- **Model-B** – In this model if the task has some pre-requirements of more tasks to be completed first and some of those allocated to the same
processor, their communication cost is counted only once but in model-A it is counted multiple times.

- **Model-C** - This model acts as an I/O processor. It implies that a processor can take the input data of a task and execute that task and at the same time it can communicate with another processor by transferring the output data of previously executed tasks.

- **Model-D** – In this model each data is divided into several units. Those units are transferred with the help of pipeline data transmission.

### Pipeline Data Transmission

### Timing of Pipeline
Some of the algorithms that we analysed for solving the given problem

**MCP ALGORITHM**

- The MCP algorithm is a list scheduling algorithm in which task priorities are based on the latest possible start time of the tasks.

- The latest possible start time is computed as difference between the critical path of the graph and the longest path from the current task to any exit task. A path length is the sum of the execution times and the communication costs of the tasks and edges belonging to the path.

- The critical path is the longest path in a graph. A task with the smallest latest possible start time has the highest priority. Ties are broken considering the priorities of the task’s descendents. The tasks are selected in the order of their priorities and assigned on the processor that can execute it the earliest.

**CPM ALGORITHM**

- Like MCP, the CPM algorithm also uses the *latest possible start time* as the task static priorities. However, the processor selection is different. The task with the highest priority is scheduled on the processor that becomes *idle* the earliest.

- CPM was originally designed without taking communication into consideration. In the case of non-zero communication delays, the scheduling performance of CPM decreases significantly. The reason is that CPM does not try to reduce communication costs, but only balances
the load of the processors. Therefore, CPM is closer to a load balancing scheme.

**FCP ALGORITHM**

- As mentioned earlier, list scheduling algorithms generally perform better compared to other scheduling algorithms for bounded number of processors. However, the list scheduling algorithms have a much higher complexity compared to multi-step scheduling algorithms.

Analyzing list scheduling algorithms, one can distinguish several steps:-

- The first step is the task priorities computation, since the whole task graph has to be traversed.
- The second step sorting tasks according to their priorities.
- The third step, task scheduling, schedules the sorted tasks one by one on the best processor. In list scheduling, usually the best processor is considered to be the processor on which the task to be scheduled starts the earliest.
- Computing the start times for all tasks requires traversing all tasks and edges, leading to a time complexity. As each task is tentatively scheduled to all processors to find the earliest start time, the processor selection takes $O((V+E)P)$ time. Finally, scheduling the task to a processor only takes $O(1)$ time because the start time of the task to the selected processor was already computed in the previous step.
- One first way to reduce the complexity of the task sorting step is not to sort all the tasks from the beginning, but to maintain only the ready tasks sorted throughout the scheduling process. However, despite the fact the sorting time is reduced, it still has the same complexity in the worst case.
- This complexity can be effectively reduced by maintaining only a constant size sorted list of ready tasks. The others are stored in an unsorted list which has an $O(1)$ time access. When a task becomes ready it is added to the sorted list when there is room to accommodate it, otherwise it is added to the unsorted list. For this reason, as long as the sorted list is not full, there can be no tasks in the unsorted list. The tasks are always dequeued from the sorted list. If the unsorted list is not empty, one task is moved to the sorted list. The time complexity of task sorting
using a list of size $H$ decreases to $O(V \log H)$ as all the tasks are enqueued and dequeued in the sorted list only once.

- A possible drawback using a fixed size for a sorted task list is that the task with the highest priority may not be included in the sorted list, but is temporarily stored in the unsorted list. The size of the sorted list must therefore be large enough not to affect the performance of the algorithm too much. At the same time, it should not be too large in view of the time complexity.

- A sorted list size of $P$ results in a task sorting complexity of $O(V \log P)$. The $O((E+V)P)$ time complexity of list scheduling processor selection can also be reduced by restricting the choice for the destination processor from all processors to only two processors. The two possible destination processors are either (a) the processor from which the last message to the given task arrives or (b) the processor which becomes idle the earliest. Considering only two candidate processors for the task to be scheduled, the selection time complexity decreases to $O(V \log (P) + E)$.

**Modification in FCP Algorithm**

The FCP algorithm maintains information of only 2 processor

1. The processor which just got free.

2. The processor which sent the last message.

Author of the algorithm argues that this approach will decrease the time complexity of allocation from $O((E+V)*P)$ to $O(V\log P + E)$ without degrading much of the performance.

Performance of the system running this algorithm decreases. Lets assume that there is a processor with high performance in the network but after this processor got freed there is one more processor that got freed with very less performance than the one just mentioned.

That is $P_i > P_j$

$j$ release time is greater than $i$ release time. Thus, there is a sure decrease in performance of overall system.
Continuing with the same idea of maintaining only information of two processor and having the time complexity of $O(V\log P + E)$, we can fo this without degrading the performance.

Instead of maintain the processor information of recently got freed we maintain information about the processor with highest performance and release time. But, maintaining this list will cost $O(P)$. So to reduce this complexity to the order of $O(\ln P)$ we maintain this list in the form of binary tree. Hence, this will bring the selection time of highest performance and released processor to $O(\ln V)$.

Therefore, maintaining the algorithm overall performance but at same time increasing the system performance.

SelectHigh(T)

1. Move left of parent while the node has no leaf.
2. Return the node

**Algorithm Proposed by Olga Rusanova and Alexender Korochkin**

**Phase 1**

Find the priority of the activities

formula, $p = \text{Sum weight of entering edges} - \text{real slack}$

**Modification proposed**

The formula given above does not give the correct priority of all the activities in the given graph. And hence I and Devanjan propose to modify this formula by the other formula as is used by MCP, and CPM algorithm.

Priority, $P_i = \text{critical path} - \text{longest path from node i to exit node.}$
Phase 2

Task allocation onto the processors can be done by using the following formula

Formulas

- Parallel
  \[ O_{m,j} = \max \{ T_m, \sum_{j=1}^{n} (T_f + e_{ji}) \} - T_{p,i} \]

- Distributed
  \[ O_{m,j} = \max \{ T_m, \sum_{j=1}^{n} (T_f + e_{ji}) - T_{p,j} \} + [W_i/p_m] \]

Again, the time complexity of the given algorithm is \( O((V+E)*P) \). But, this time complexity can be further brought down to \( O(\ln V+E) \).

Modification Proposed

we maintain information about the processor with attributes, performance and release time. Then we sort this list according to their performance. But, maintaining this list will cost \( O(P) \). So to reduce this complexity to the order of \( O(\ln P) \) we maintain this list in the form of binary tree. Hence, this will bring the selection time of highest performance and released processor to \( O(\ln V) \).

This modification allows us to calculate start delay for each task only once instead of calculating start delay of each task for each of the processor.

Therefore, the new formula becomes as following:

- Parallel
  \[ O_{m,j} = \max \{ T_{\min}, \sum_{j=1}^{n} (T_f + e_{ji}) \} - T_{p,i} \]

- Distributed
  \[ O_{m,j} = \max \{ T_{\min}, \sum_{j=1}^{n} (T_f + e_{ji}) - T_{p,j} \} + [W_i/p_m] \]
Here, $T_{\text{min}}$ is the released processor with highest performance. Therefore, the time complexity of the algorithm becomes $O(V \ln P + E)$

**EXAMPLE**

Critical Path ->1->2->4->5->6

Square denotes earliest occurrence time for event $j$

Triangle denotes Latest occurrence time for event $j$
Phase 1

**Priority Decision**

Calculate earliest time and latest time of event j

For each node

- Calculate latest possible start time [modified]
- The latest possible start time is computed as difference between the critical path of the graph and the longest path from the current task to any exit task

**Calculation**

- $T_1 = 0$
- $T_2 = 6$
- $T_3 = 13$
- $T_4 = 15$
- $T_5 = 16$
- $T_6 = 29$
- Hence, Priority = 1\(>\)2\(>\)3\(>\)4\(>\)5\(>\)6
Phase 2

Task Allocation

• Formulae

• Parallel

• $O_{m,j} = \max\{T_m, \sum_{j=1}^{n}(T_f + e_{ji})\} - T_{p,i}$

• Distributed

• $O_{m,j} = \max\{T_m, \sum_{j=1}^{n}(T_f + e_{ji}) - T_{p,j}\} + [W_i/p_m]$  

$m$ – denotes processor number

$j$ – denotes activity

$T_m$ – processor release time

$T_f$ – Data formation time

$T_{p,i}$ – Latest occurrence time for time $j$. 
CALCULATION

- $O_{0,1} = 0$ scheduled
- $O_{1,1} = 0$
- $O_{0,2} = -1$ scheduled [break arbitrarily]
- $O_{1,2} = -1$
- $O_{0,3} = -6$
- $O_{1,3} = -7$ scheduled [choose most negative]
- $O_{0,4} = -7$ scheduled
- $O_{1,4} = -4$
- $O_{0,5} = 0$
- $O_{1,5} = -5$ scheduled
- $O_{0,6} = -12$ scheduled
- $O_{1,6} = -12$
Processor time graph
Conclusion

I presented to you a study of list based static scheduling methods for parallel and distributed systems. My study shows

1. The existing algorithms can be modified for the best performance of scheduling algorithm and parallel and distributed system. That is we can achieve the both at same time.

2. Heuristic algorithms are better than optimal scheduling algorithms.

3. I also presented the modifications that needs to be done in order for the best performace of the distributed and parallel systems without degrading the performance of the scheduling algorithms.

Reference:

1. On the Complexity of List Scheduling Algorithms for Distributed-Memory Systems
2. Optimal Scheduling Algorithm for distributed memory machines [1998]
3. Coffman E., Graham R. Optimal scheduling for two-processor system
4. Operational Research – Hamdy A. Taha
6. Tracy D Braun. A comparison of eleven Static Heuristic for mapping class of independent Tasks onto hetrogenous distributed computing system.