

INSTITUTE OF MATHEMATICS AND STATISTICS

UNIVERSITY OF SAO PAULO

Scheduling in Grid Computing using Master-Slave Scheduling Model

Peter Ngugi Nyumu

Professor. Dr Alfredo Goldman

Mac-0461

# 1 Introduction

Master-Slave scheduling model, involves two sets of processors, the master processors that are responsible of preprocessing and postprocessing of work orders and the slave processors that are responsible for the actual execution of the orders. The number of slave processors is no less than the number of work orders.

## 2 Description and Applications

### Description

A set of job is to be processed by the system of master and slave processors. Each job has three tasks associated with it. The first is a preprocessing task ( $a_i$ ), the second is a slave task ( $b_i$ ), and the third a postprocessing task ( $c_i$ ). The available processors are divided into two categories: master and slave. If  $n$  denotes the number of jobs, then no schedule can use more than  $n$  slaves, Thus we can assume that there are exactly  $n$  slaves. The makespan of finish time of a schedule is the earliest time at which all tasks have been completed.

In this model we look at the following schedules;

A *no-wait-in schedule*, each slave task must be scheduled immediately after the corresponding preprocessing task finishes and each postprocessing task must be scheduled immediately after the corresponding slave task finishes. That means, once a job starts, it will not stop until it finishes[5].

A *canonical schedule*, satisfies the following properties: (1) There are no pre-emptions; (2) The preprocessing tasks begin on the master machine at time 0 and complete at time  $\sum a_i$ ; (3) The slave tasks begin as soon as their corresponding preprocessing tasks complete; (4) The postprocessing tasks are done in the same order as the slave tasks complete and as soon as possible. If two slave tasks complete at the same time, the postprocessing tasks are scheduled in the same order as the preprocessing tasks[5].

### Applications

In *Parallel computing*, the common paradigm involves the use of a single main computational thread that employs the fork and join operations to spawn parallel tasks or threads, and the synchronize following the completion of the tasks. Every spawned thread involves, preprocessing by main thread, work done in the thread and preprocessing by the main thread.

Another application is the *semiconductor testing* operation, use the master-slave paradigm. the chips are subjected to thermal stress for a duration of time to bring out latent defects leading to infinity mortality that might otherwise surface in the operating environments.

*Industrial applications* is another application. It includes the case of consolidators that receive orders to manufacture quantities of various items. The consolidator needs to assemble the raw material needed for each task, load the trucks that will deliver this material to slave processors and perform an inspection before the consignment leaves.

### 3 Single-Master Master-Slave Systems

As the name suggest, a single-master master-slave system is composed of one master machine and  $n$  slaves to process  $n$  jobs.

#### 3.1 Unconstrained Minimum Finish Time Problem (UMFT)

UMFT problem is NP-hard.

If schedule  $S$  is unconstrained, the we can rearrange the master tasks so that all preprocessing tasks complete before any postprocessing task starts. This can be done without increasing the makespan. In this case we apply the canonical schedule. It is clear that for every unconstrained schedule  $S$  there is a corresponding canonical schedule with better or same makespan.

The next theorem finds the worst case performance of an arbitrary canonical schedule  $S$ . Let  $C^S$  be the makespan of the canonical schedule  $S$  and  $C^*$  the optimal makespan of UMFT. We define  $i$  follows (precedes)  $j$  to mean  $i$  comes after (before)  $j$  in the permutation that defines the schedule.

*Theorem*

For any canonical schedule  $S$ ,  $\frac{C^S}{C^*} \leq 2$  and the bound is tight.

*Proof*

if  $C^S = \sum_i (a_i + c_i)$  the  $S$  is optimal and the error bound of 2 is valid. Contrary to that, it means that it exist idle time on the master processor and  $C^S > \sum_i (a_i + c_i)$ . Lets  $c_{i_0}$  be the last postprocessing task in  $S$  that starts immediately after its corresponding slave task  $b_{i_0}$ .  $i_0$  exists due to the fact we have an idle time on the master, it follows that,

$$C^S = \sum a_i(a_{i_0} + b_{i_0} + c_{i_0}) + \sum c_i \leq 2C^*$$

Lets consider an instance with  $k + 1$  jobs where  $k > 0$ , to see that the error bound is tight. The processing requirement is  $(1, \epsilon, \epsilon)$  for the first  $k$  jobs, while the one additional to the  $k$  jobs(the  $(k + 1)$ -st job) will require  $(\epsilon, k, \epsilon), \epsilon < \frac{1}{k}$ . The schedule  $S$  that process  $a_{k+1} = \epsilon$  last among all preprocessing tasks has makespan  $C^S = 2k + 2\epsilon$ . The schedule  $S^*$  that process  $a_{k+1}$  first among all preprocessing tasks has a makespan  $C^* = k + (k + 2)\epsilon$  and hence  $\frac{C^S}{C^*} \rightarrow 2$  as  $\epsilon \rightarrow 0$ .

Now we present a heuristic ( $H$ ), whose error bound is  $\frac{3}{2}$ . The following are steps to follow to execute this heuristic;

- Let  $S_1 = \{i : a_i \leq c_i\}$  and  $S_2 = \{i : a_i > c_i\}$
- Reorder the jobs in  $S_1$  according to nondecreasing order of  $b_i$ .
- Reorder the jobs in  $S_2$  according to nonincreasing order of  $b_i$ .
- Generate the cononical schedule in which the  $a$  tasks of  $S_1$  precede those of  $S_2$ .

*Theorem*

$$\frac{C^H}{C^*} \leq \frac{3}{2} \text{ and bound is tight.}$$

*Proof*

Let  $S^*$  be an optimal schedule for UMFT with a makespan  $C^*$ , based on the processing requirement on the processing requirements  $(a_i, b_i, c_i)$  of job  $i$ , we define an auxiliary problem  $P'$  with  $(a'_i, b'_i, c'_i)$  as its processing requirements.

In  $P'$  all preprocessing tasks in  $S_1$  are zero and hence they can precede all nonzero processing tasks, similarly to  $S_2$  but in this case follows the nonzero preprocessing tasks.  $S'$  is the schedule generated in step 4 of  $H$  if applied on  $P'$ . Let  $C'$  be the makespan of  $S'$ , by optimality of  $S'$  we can conclude that  $C' \leq C^*$ . From  $S'$  and  $P'$  we generate a schedule  $S^H$  for each original problem.

Consider an instance of  $k + 1$  jobs, where  $k > 0$ . The first  $k$  jobs require  $(1, \epsilon, 1)$  processing requirement, for the additional job, the processing requirement will be  $(\epsilon, 2k, \epsilon)$ . Jobs 1 to  $k$ , are processed in any order, lastly job  $k + 1$ , the makespan is  $3k + 2\epsilon$ , while an optimal solution with makespan is  $2k + 2\epsilon$ .

$$\frac{C^H}{C^*} = \frac{3k+2\epsilon}{2k+2\epsilon} \rightarrow \frac{3}{2}, \text{ when } \epsilon \rightarrow 0$$

## 3.2 Order Preserving Minimum Finish Time (OPMFT)

In this case we have same order of preprocessing and postprocessing. Some restrictions are considered in building the OPMFT algorithm, which include:

- the schedule are nonpreemptive
- slave tasks begin as soon as their corresponding preprocessing tasks are complete
- each postprocessing task begins as soon after the completion of its slave task as is consistent with the order preserving constraint.

With the above restrictions, its possible to come up with  $O(n \log n)$  algorithm, by defining a conical order preserving schedule (COPS) in an order preserving schedule in which (1) the master processor completes the preprocessing tasks of all jobs before beginning any of the postprocessing tasks, and (2) the processing tasks begin at time zero and complete at time  $\sum_{i=1}^n a_i$ .

Having the above information we can look at an OPMFT related *Theorem* which state that:

There is an OPMFT schedule which is a COPS in which the preprocessing order satisfies the following:

- jobs with  $c_j > a_j$  come first.
- jobs with  $c_j = a_j$  come next.
- jobs with  $c_j < a_j$  come last.

These can be seen through a *Lemma*:

Considering the COPS defined by a permutation  $\sigma$ . Assume that job  $j$  is preprocessed immediately before job  $j+1$ . If  $c_j \leq a_j$  and  $c_{j+1} \geq a_{j+1}$ , then the schedule length is no less than of the COPS obtained by interchanging  $j$  and  $j+1$  in  $\sigma$ .

Here follows the *Proof* of the Lemma:

Take a job  $j$  which start the preprocessing at time  $t$ , and it precedes job  $j+1$ . With  $A$  as the elapsed time between the completion of task  $a_{j+1}$  and the start of the postprocessing of job  $j$ , the time between the start of  $c_j$  and  $c_{j+1}$  is  $\Delta > 0$ , and  $\tau$  is  $c_{j+1}$  completion time.

Let  $\sigma'$  be the permutation obtained by interchanging jobs  $j$  and  $j + 1$  in  $\sigma$ . Let  $t'$  and  $\tau'$  be the finishing time of  $c_{j+1}$  and  $c_j$  respectively. If  $\Delta \geq a_j$ , then  $t' \leq \tau - a_j$ . As a result,  $b_j \leq a_{j+1} + A \leq c_{j+1} + A$ , hence,  $b_j$  finishes by  $t'$ . We have,  $\tau' = t' + c_j \leq \tau - a_j + c_j \leq \tau$ , this shows that the preprocessing tasks of the remaining jobs can be done so as to complete at or before the completion times in  $\sigma$  and the interchanging of  $j$  and  $j + 1$  does not increase the schedule length.

If  $\Delta < a_j$ , then  $c_{j+1}$  starts at time  $t + a_{j+1} + a_j + A$  in  $\sigma'$ . Then,  $t' = t + a_{j+1} + a_j + A + c_{j+1}$ . The time  $b_j$  finishes in  $\sigma'$  is  $t + a_{j+1} + a_j + b_j \leq t + 2a_{j+1} + a_j + A \leq t + a_{j+1} + a_j + A + c_{j+1} = t'$ . Hence,  $c_j$  finishes at  $t' + c_j = t + a_{j+1} + a_j + A + c_{j+1} + c_j \leq \tau$ . As a result, the order preserving schedule defined by  $\sigma'$  has a finish time that is  $\leq$  that of the order preserving schedule defined by  $\sigma$ .

### 3.3 Canonical Reverse Order Schedules (CROS)

This happens in construction of reverse order processing. For any given preprocessing permutation  $\sigma$ , this reverse order can be constructed in the following manner:

- the master preprocesses the  $n$  jobs in the order  $\sigma$
- slave  $i$  begins the slave processing of job  $i$  as soon as the master completes its preprocessing.
- the master begins the postprocessing of the last job in  $\sigma$  as soon as its slave task is complete
- the master begins the postprocessing of job  $j \neq k$  at the later of the two times (a) when it has finished the postprocessing of the successor of  $j$  in  $\sigma$ , and (b) when slave  $j$  has finished  $b_j$ .

It is possible to construct an  $O(n \log n)$  algorithm of an ROMFT schedule which is a CROS. The following *Theorem* help to construct this algorithm:

The CROS defined by the ordering  $b_1 \geq b_2 \geq \dots \geq b_n$  is an ROMFT schedule.

The proof follows the following *Lemma*:

Let  $\sigma = (1, 2, \dots, n)$  be a processing permutation. Let  $j < n$  be such that  $b_j < b_{j+1}$ . Let  $\sigma'$  be obtained from  $\sigma$  by interchanging jobs  $j$  and  $j + 1$ . Let  $\tau$  and  $\tau'$ , respectively, be the finish times of the CROSs  $S$  and  $S'$  corresponding to  $\sigma$  and  $\sigma'$ .  $\tau' \leq \tau$ .

Here follows the *Proof* of the Lemma above:

Let  $t$  be the time at which job  $j + 2$  finishes in  $S$  and  $S'$ . If  $j = n - 1$ , let  $t = 0$ . Let  $s_j(s'_j)$  be the time at which task  $b_j$  finishes in  $S(S')$ . Let  $s_{j+1}$  and  $s'_{j+1}$  be similarly defined. From the definition of a CROS, it follows that:

$$\begin{aligned} s_j &= \sum_1^j a_k + b_j ; s_{j+1} = \sum_1^{j+1} a_k + b_{j+1} \text{ (eq *)} \\ s'_j &= \sum_1^{j+1} a_k + b_j ; s'_{j+1} = \sum_1^{j+1} a_k - a_j + b_{j+1} \end{aligned}$$

Let  $q(q')$  be the time at which  $c_j(c_{j+1})$  finishes in  $\sigma(\sigma')$ . It is sufficient to show that  $q \leq q'$ :

$$q = \max\{\max\{t, s_{j+1} + c_{j+1}, s_j\} + c_j = \max\{t + c_j + c_{j+1}, s_{j+1} + c_j + c_{j+1}, s_j + c_j\}$$

(eq \*\*)

and

$$q' = \max\{\max\{t, s'_j + c_j, s'_{j+1}\} + c_{j+1}\} = \max\{t + c_j + c_{j+1}, s'_{j+1} + c_{j+1}, s'_j + c_j + c_{j+1}\}$$

From equations (eq \*) and (eq \*\*) and the inequality  $b_j < b_{j+1}$ , we obtain

$$s'_j + c_j + c_{j+1} = s_{j+1} + b_j - b_{j+1} + c_j + c_{j+1} < s_{j+1} + c_j + c_{j+1} \leq q$$

and

$$s'_{j+1} + c_{j+1} = s_{j+1} - a_j + c_{j+1} < s_{j+1} + c_{j+1} < s_{j+1} + c_{j+1} + c_j \leq q$$

which shows that  $q' \leq q$ . Like mentioned in the introduction there is another category, *No-Wait in Process*:

- The Minimize Finish Time (MFTNW), subject to the no-wait-in-process constraint. It is NP-hard problem.
- The Order-Preserving version of MFTNW. It is subject to the no-wait-in-process as well as order-preserving constraints. Its also NP-hard problem.
- The Reverse-Order version of MFTNW. It is subject to no-wait-in-process and reverse-order constraints. It is a polynomial problem.

## 4 Appendix

Grid computing has unique characteristics such as, in resource distribution, architectural aspect, job processing environment, just to name a few. To explain this unique characteristics we look at the taxonomies of grid computing.[1].

- **Local vs. Global**

The local scheduling use a single processor to allocate and execute the tasks, while in global scheduling the the system allocate processes to multiple processors to optimize a system wide performance objective. Grid scheduling falls in the category of global scheduling.

- **Static vs. Dynamic**

Under global scheduling we have a choice between static and dynamic scheduling, this choice indicates the time at which the scheduling or assignment decisions are made. In static scheduling, information regarding all resources in the grid as well as all the tasks in an application is assumed to be available by the time the application is scheduled. On the other hand, dynamic scheduling the basic idea is to perform task allocation as the application executes, which is important in real-time mode applications as well as in situation where you cannot determine the execution time.

- **Optimal vs. Suboptimal**

All information regarding the state of resources and the jobs is known in this case, hence an optimal assignment could be made based on some criterion function, such as minimum makespan and maximum resource utilization. The NP-Complete nature of scheduling algorithms and the difficulty in Grid scenarios to make reasonable assumptions which are usually required to prove the optimality of an algorithm, current research tries to find suboptimal solutions, which can be further divided into the following two general categories.

- **Approximate vs. Heuristic**

Suboptimal approximate is a sufficiently "good" solution taken, instead of searching the entire solution space for an optimal solution. The factors which determine whether this approach is worthy of pursuit include [2];

- Availability of a function to evaluate a solution.



- The time required to evaluate a solution.
- The ability to judge the value of an optimal solution according to some metric.
- Availability of a mechanism for intelligently pruning the solution space.

The other branch in the suboptimal category is called heuristic. This branch represents the class of static algorithms which make the most realistic assumptions about a priori knowledge concerning process and system loading characteristics. It also represents the solutions to the scheduling problem which cannot give optimal answers but only require the most reasonable amount of cost and other system resources to perform their function. The evaluation of this kind of solution is usually based on experiments in the real world or on simulation. Not restricted by formal assumptions, heuristic algorithms are more adaptive to the Grid scenarios where both resources and applications are highly diverse and dynamic.

- **Distributed vs. Centralized**

The responsibility for making global scheduling decisions may lie with one centralized scheduler, or be shared by multiple distributed schedulers. In a computational Grid, there might be many applications submitted or required to be rescheduled simultaneously. The centralized strategy has the advantage of easy implementation, but suffers from the lack of scalability, fault tolerance and the possibility of becoming a performance bottleneck. For example, Sabin et al [3] propose a centralized meta-scheduler which uses backfill to schedule parallel jobs in multiple heterogeneous sites. Arora et al [4] present a completely decentralized, dynamic and sender-initiated scheduling and load balancing algorithm for the Grid environment. A property of this algorithm is that it uses a smart search strategy to find partner nodes to which tasks can migrate.

- **Cooperative vs. Non-cooperative**

We considered whether the nodes involved in job scheduling are involved in cooperation between the distributed components (cooperatively), this is when, each grid scheduler has the responsibility to carry out its own portion of the scheduling task, but all schedulers are working toward a common system-wide goal. The other category occurs when the decision making process is done independently of other processors (non-cooperatively), in this case, individual schedulers act alone as autonomous entities and arrive at decisions regarding

their own optimum objects independent of the effects of the decision on the rest of system[2].

## References

- [1] T. Casavant, and J. Kuhl, A Taxonomy of Scheduling in General-purpose Distributed Computing Systems, in IEEE Trans. on Software Engineering Vol. 14, No.2, pp.141–154, February 1988.
- [2] Handbook of Scheduling, Joseph Y-T. Leung, paper 17
- [3] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan, Scheduling of Parallel Jobs in a Heterogeneous Multi-Site Environment, in the Proc. of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes In Computer Science; Vol. 2862, Washington, U.S.A, June 2003.
- [4] M. Arora, S.K. Das, R. Biswas, A Decentralized Scheduling and Load Balancing Algorithm for Heterogeneous Grid Environments, in Proc. of International Conference on Parallel Processing Workshops (ICPPW'02), pp.:499–505, Vancouver, British Columbia Canada, August 2002.
- [5] Scheduling problems in master-slave model Joseph Y.-T. Leung Hairong Zhao