Fault-Tolerant Dynamic Task Allocation

HingOn Miu

Carnegie Mellon University
hmiu@andrew.cmu.edu

ABSTRACT

If one process is assigned with m tasks, the analysis of completing all the tasks is fairly easy because all tasks must be done sequentially; however, it is much harder to analyze how to complete all the tasks with p concurrent processes using as little resources as possible. In this paper, we are interested in analyzing p asynchronous processes that cooperate to perform m tasks. We first introduce the problem statement and several important environmental factors. Then, we explain some interesting approaches to the problem. This allows us to gain a good understanding of how to allocate tasks to processes dynamically under adversarial settings.

KEYWORD

Dynamic task allocation, distributed computing, adversary, asynchronous, crashes, partitionable network, deterministic, randomized, To-Do Tree

INTRODUCTION

In modern days, distributed computing has been a major topic because the computational power of a single machine can no longer keep up with the complexity of the problems. Therefore, software developers and researchers study extensively how to distribute work among multiple machines. Hence, a huge emphasis lies on the efficiency of the task scheduler: how m tasks can be distributed among p processes efficiently. In an ideal environment with no adversity to interfere with the computation, a simple load balancer can do the job fairly well such that each processor is assigned with $\Theta(m / p)$ work and that each task is executed exactly once. However, given the realistic distributed platforms, the task scheduling algorithm must be able to deal with adverse conditions. Therefore, different adversarial environments of distributed platforms must be considered so that the task scheduling algorithm is sufficiently robust.

PROCESS FAILURES A processor may experience unexpected failure, such as a crash. So, the system must have mechanism to reassign the pending tasks among other processors such that all processes may not wait on the crashed processor indefinitely. Therefore, the system at most allows $p - 1$ crashes such that at least one process is running at any time. Also, if the processes are asynchronous such that each has various processing speed, some process may be arbitrarily slow. So, faster processors may remain idle for arbitrarily long time while they wait for the slow processor to complete the task. Hence, the system should be able to detect the abnormal behaviors of a processor, and redistributed the tasks among other processors.

COMMUNICATION Processors may experience intermittent connectivity such that information on the network may be lost. So, the system should develop mechanism to guarantee that the receiving processor gets the proper message from the sending processor. Also, the network may be fragmented such that some processor may only communicate with processors in its own partition, but not the processors in other partitions. Moreover, the network may have unpredictable delays, and so a processor may remain idle waiting for certain event acknowledgment. Thus, it is essential that the system should be able to

detect abnormal network problems, and attempt to resolve the issue by resending messages or reassigning tasks.

REAL LIFE APPLICATIONS The significance of fault-tolerant distributed computing lies in many real life applications. Transportation systems that schedule trains and planes have to be fault-tolerant, or else there are safety concerns. Banking and stock market systems have to be fault-tolerant as well, or else there are reliability concerns. Other real life applications include telecommunication systems, medical systems, etc.

ASSUMPTIONS The tasks are assumed to be similar, independent, and idempotent. By similarity of the tasks, it means that the task executions consume comparable resources. By independence of the tasks, the tasks can be executed in any order. By idempotence of the tasks, each task should be executed at least once. The processors are assumed to be either synchronous or asynchronous. Synchronous processes are governed by a global clock such that all processes run in the same processing speed. Asynchronous processes are governed by local clocks such that each run in arbitrary processing speed. The system is assumed to be either centralized or decentralized. One master process handles all the task allocations in a centralized system. All processes cooperate with each other and determine their next tasks to execute themselves in a decentralized system.

PROBLEM STATEMENT Given a set of m tasks, p processors perform all tasks under adversary A. This problem is solved when at least one of the p processors knows all m tasks are completed. To better analyze the total work of a task scheduling algorithm T, we define the total work S calculation as the following [1].

An execution context is defined as the environment for p processors to execute m tasks, and an adversary decides the execution context. Let ε be the execution condition given the adversary A. Let $\tau(\varepsilon)$ be the total time T spent on solving the problem under execution condition ε. Let $X_{i\,j}$ be the Boolean variable that indicates whether processor p_j executes work at time step i.

$$S(T, A) = \max_{\varepsilon \in A} \sum_{i=1}^{\tau(\varepsilon)} \sum_{j=1}^{p} X_{ij}(\varepsilon)$$

The obvious lower bound on total work is $\Omega(m)$ because each task must be completed at least once. A trivial solution is making each of the p processes to run all m tasks such that the system ensures that all m tasks are executed. Although this solution requires no communication among processors, each processor has $\Theta(m)$ work and so the total work is $\Theta(m\,p)$. Thus, researchers strive to develop a task scheduling algorithm that is much efficient than this trivial solution.

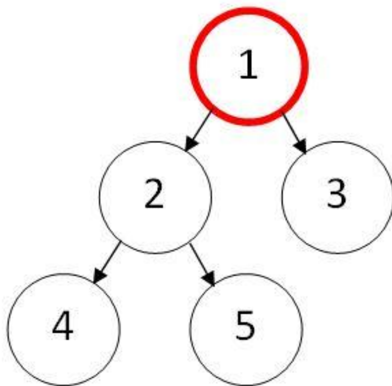
DETERMINISTIC SOLUTION

For deterministic approaches to this problem, we mainly investigate two types of adversarial settings, which are a system with maximum $f < p$ crashes and a system with maximum $f < p$ partitioned networks.

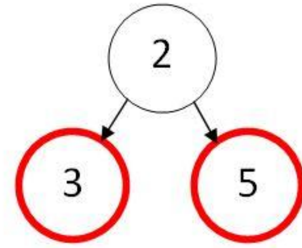
CRASHES First of all, we assume that once a processor is crashed, it does not restart. The system allows at most $p - 1$ crashes, but we assume reliable multicast is available such that either a message is successfully delivered to all processors or no processor receives the message. The system assumes reliable multicast communication because many realistic distributed platforms have reliable message broadcasting, such as Ethernet and bypass rings. It is important that we separate the concerns between handling processor failures and handling network failures since we can then construct solutions with a higher modularity.

The main idea of algorithm *AN* is that all processors are workers and some of them are coordinators at various iterations of the algorithm. Coordinators collect the progress of all processors, while workers report their progress to coordinators. For simplicity, coordinators report progress to themselves as well. The number of coordinators is determined by the martingale principle: if all coordinators crash in a iteration, the number of coordinators is doubled for the next iteration; if at least one coordinator does not crash in a iteration, the number of coordinators is reduced to one for the next iteration.

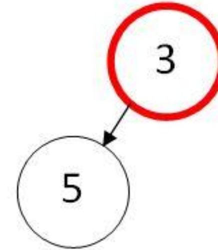
Therefore, each processor maintains a balanced binary tree *B* of the id's of online processors, such that the number of processors in each level is double of the number of processors in previous level (except the last level). All online processors update their own local *B* at the end of each iteration so that crashed processors are removed from *B*. Initially, processor in level 0 of *B* is assigned to be coordinator. If no coordinator survives after the iteration, processors in level 1 are assigned to be coordinators for the next iteration. Therefore, processors in level i are assigned coordinators if all 2^{i-1} processors in level $i - 1$ crash in previous iteration.



(a) *B* at 1st iteration



(b) *B* at 2nd iteration



(c) *B* at 3rd iteration

Figure 1: Example of a local *B* getting updated in each iteration. In (a), root of *B* is initially assigned as the coordinator. In (b), processor 1 and 4 crashed. Since the only coordinator crashed, so the next level in *B* is assigned as coordinators. In (c), processor 2 crashed. Since no coordinator crashed, so the root of *B* is again assigned as coordinator.

Also, each processor maintains a local list *U* for all unaccounted tasks. In a iteration, the load balancing rule is that each worker with its processor id in level *h* of its local *B*, executes the *j*th task in its local *U*, where $j = (h \bmod |U|)$. Then, all workers report their progress to coordinators. Here is the implementation of the algorithm *AN* [1]:

def *AN*:

while *U* is not empty:

1. executes task *j*
2. report task *j* to coordinators
3. coordinators gather reports
4. coordinators update *U* and *B*
5. coordinators multicast *U* and *B* to all
6. gathers summary from coordinators
7. updates local *U* and *B*

To analyze the complexity of algorithm *AN*, we define the total work $S = S_a + S_b$, where S_a accounts for the iterations that the processors survive and S_b accounts for iterations that the processors crash. Let $|B_i|$ be the number of remaining online processors and $|U_i|$ be the number of remaining unaccounted tasks in iteration i.

First, we consider S_a with two cases: the iterations with $|B_i| \leq |U_i|$ and the iterations with $|B_i| > |U_i|$. In the first case, the load balancing rule allows at most one processor to be assigned to each of the unaccounted tasks, and so the total processor work S_1 is equivalent to total tasks executed, which is $O(m + p)$.

In the second case, we define a constant d to be $\log p / \log \log p$. Then, we consider two sub-cases: $|U_{i+1}| < |U_i| / d$ and $|U_{i+1}| \geq |U_i| / d$. In the first sub-case, it is obvious that there is at most of $O(\log_d p)$ such iterations because $|U_{i+1}| < |U_i| < p$. Since $d^d = \Theta(p)$, we know that $O(\log_d p) = O(d)$. Because there is at most p processors with such iterations, the total work $S_{2.1}$ is $O(p\, d)$.

In the second sub-case, the load balancing rule ensures at least $\lfloor |B_i| / |U_i| \rfloor$ processors are assigned to each of the unaccounted tasks since $|B_i| > |U_i|$. Since $|U_{i+1}|$ unaccounted tasks remained, then the number of crashed processor in the ith iteration is at least $|U_{i+1}| * \lfloor |B_i| / |U_i| \rfloor \geq (|U_i| / d) * (|B_i| / 2|U_i|) = |B_i| / 2d$. So, the number of online processors for the next iteration is at most $|B_i| - |B_i| / 2d$, which is $O(p\,(1 - 1/2d))$. Let there be k such iterations in this sub-case so that the first iteration of these has at most p online processors, and so each of these subsequent iterations j should then have at most $p\,(1 - 1/2d)^j$ online processors. Since the number of online processors in a iteration equals to the number of unit work done in that iteration, we conclude the total work $S_{2.2}$ as following:

$$S_{2.2} \leq \sum_{j=0}^{k} p(1 - \frac{1}{2d})^j \leq \frac{p}{1 - (1 - \frac{1}{2d})} = O(pd)$$

$$\therefore S_a = S_1 + S_{2.1} + S_{2.2} = O(m + pd) = O(m + p\frac{\log p}{\log \log p})$$

Next, we consider S_b. Say the system allows f crashes, where $f < p$. The number of iterations in which processors crash is at most $\log f$ [1]. The total work S_b should be the sum of the number of online processors in all iterations with processors crash, since each online processor in a iteration represent one unit of work, and so we have the following:

$$S_b \leq \log f * \sum |U_i| \leq (p + S_a) \log f = O(S_a \log p)$$

$$\therefore S = S_a + S_b = O(m \log p + p\frac{\log^2 p}{\log \log p})$$

PARTITIONABLE NETWORKS First of all, we again assume reliable multi-cast is available. We assume the network is not fragmented initially. However, the network can be partitioned progressively during the task executions. Once the network is fragmented, processors of distinct fragments cannot communicate, while processors in the same fragment can communicate. Since there are p processors, we know the number of fragmentation failures f allowed is at most $p - 1$. This is the case when there are p fragments in the network, each contains one processor. Since processors in different fragments cannot communicate, some processor may learn that all tasks are executed but it cannot inform processors in other fragments to terminate. So, it is essential for each processor to learn that all tasks are executed before it terminates.

The basic idea of algorithm AF is that each processor executes one of the remaining unaccounted tasks according to its load balancing rule until it learns all tasks have been executed. The set T of the initial m tasks are known to all processors. Each processor maintains a local set D of tasks that are known as already executed. So, each processor knows the set U of remaining unaccounted tasks. Each processor also maintains a local set F of processors in the current fragment. Since all tasks and all processors have unique id's, we can assume U and F are sorted. The load balancing rule is defined as the following: say the processor i has rank k in F, then it executes the task with rank $k \bmod |U|$. Hence, in each iteration, each processor broadcasts and updates its local U and F, and then it executes a task according to its load balancing rule. At any point of execution, if a processor finds its local U is empty, it terminates. Therefore, this algorithm terminates when all processors terminate.

Now, we analyze the work complexity of AF. Since each processor keeps track of tasks executed in local D, it never executes more than m tasks. Since each processor executes at least one unaccounted task in each iteration, there are at most m iterations. Since there are at most $f + 1$ fragments and each fragmentation failure causes each processor to become a member of at most one new fragment, we know that any processor is a member of at most $f + 1$ fragments during the computation. Since there are p processors in total, we can conclude the total work of the iterations with new fragmentation failures is $O(p * (f + 1)) = O(p * f)$.

To compute the total work of the iterations without new fragmentation failures, we consider two cases: $|U| < |F|$ and $|U| \geq |F|$. In the first case, all remaining unaccounted tasks can be completed in one iteration since the number of processors in current fragment is more than the number of remaining accounted tasks. Since each group of processors of distinct fragments has at least one such iteration and there are at most $f + 1$ fragments, there are $O(f + 1)$ such iterations. Since the number of tasks executed in such iteration is bounded by $|F|$, the number of tasks executed in such iteration is $O(p)$. Therefore, the total work of these iterations is $O(p) * O(f + 1) = O(p * f)$.

In the second case, the number of processors in current fragment is less than or equal to the number of remaining accounted tasks. Since each iteration completes $|F|$ unaccounted tasks, we know that there can be no more than $\lfloor |U| / |F| \rfloor$ iterations in each fragment. Because the number of tasks executed in any of these iterations is bounded by $|F|$, the work of these iterations in one fragment is then $O(\lfloor |U| / |F| \rfloor * |F|) = O(|U|) = O(m)$. Hence, since there are at most $f + 1$ fragments, we know that the total work of these iterations is $O(m) * O(f + 1) = O(m * f)$. Therefore, we can conclude the total work of AF is the following [4]:

$$O(pf) + O(pf) + O(mf) = O(f * (m + p))$$

RANDOMIZED SOLUTION

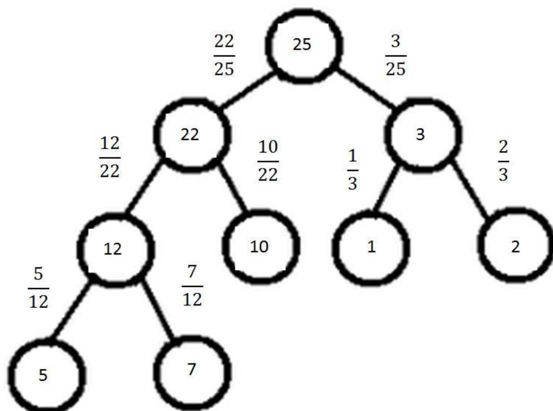
For randomized approaches to this problem, we investigate a recent algorithm proposed by Dan Alistarh, Michael Bender, Seth Gilbert, and Rachid Guerraoui [2].

TO-DO TREE The choice of which task each processor execute next is guided by a balanced binary tree called a To-Do tree. Each leaf represents a fixed number of tasks, so there are m leaves if each leaf represents one task. Each node also contains a min-register, which is a storage unit to record the number of unaccounted tasks in the sub-tree of this node. The min-register is a concurrent data structure that supports write and read operations that returns the smallest value previously written. A min-register with value v

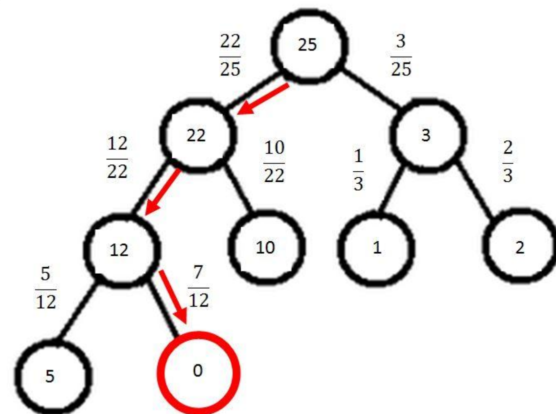
stored within takes O(log v) work for each read and write operations [3], and so it is clear that the work is O(log m) in the case of storing m tasks in the tree.

Each processor decides which task to execute next by performing a tree-walk. The tree-walk begins at the root node of the To-Do tree. The processor checks the min-register at the root node, and so it knows all tasks are executed when it reads 0. Otherwise, the processor traverses the tree by reading the left child's min-register value a and the right child's min-register value b. The processor then computes a random real number r between 0 and 1. If $r < a / (a + b)$, the processor proceeds to the left child. Otherwise, the processor proceeds to the right child. So, it is obvious to see that a processor is more likely to execute task in the sub-tree with more unaccounted tasks.

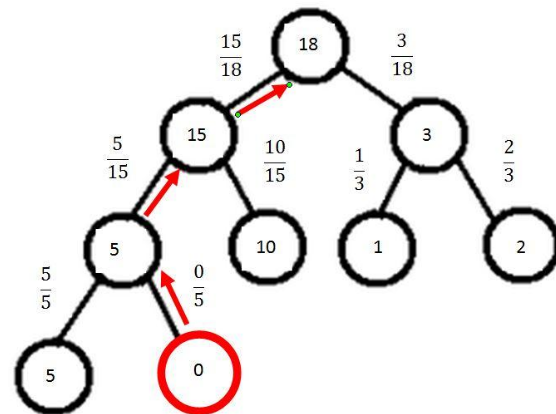
The traversal ends when either the processor reaches a leaf node or reads both 0 from left and right children's min-register. In the first case, the processor executes all the tasks on the leaf node and writes 0 to the min-register of the leaf node when all tasks on the leaf node are executed. Then, the processor proceeds to its parent node and update the parent node's min-register. The procedure terminates after the root node's min-register is updated.



(a) Original To-Do Tree



(b) Traversing To-Do Tree



(c) Updating To-Do Tree

Figure 2: Example of a processor executing tasks in To-Do Tree. In (a), the leaves contain the number of unaccounted tasks, and the internal nodes contain the sum of unaccounted tasks from both children. The labels on the edges are the probability of a processor proceeding from those edges. In (b), a processor reaches a leaf and executes all unaccounted tasks in that leaf. In (c), the processor updates each parent nodes until it reaches the root node.

Since the To-Do tree is a concurrent data structure, all processors are accessing the tree at the same time. It is obvious that all m tasks are guaranteed to be executed eventually after some tree-walks. To analyze the total work of completing m tasks with p processors, we split the execution into two epochs: the total work when the remaining unaccounted tasks $\geq 2p$ and the total work when the remaining unaccounted tasks $< 2p$. Each epoch is then split into phases. The number of remaining leaves with min-register value ≥ 1 decreases by p in each phase of the first

epoch, while the number of remaining leaves with min-register value ≥ 1 decreases by a factor of 2 in each phase of the second epoch. Therefore, we conclude that there are at most $m/p + \log_2 2p$ = O(m/p + log p) phases to complete executing all m tasks.

Knowing that there are O(m/p + log p) phases in total, there may be more than p tree-walks in some phase by bad luck if p is much smaller than m. By Chernoff bound, we can conclude that each phase has O(p) tree-walks [2]. Hence, we then have O(p) * O(m/p + log p) = O(m + p log p) tree-walks in total. Since traversing the To-Do tree takes O(log m) and reading/writing each min-register takes O(log m), we know that each tree-walk takes O($\log^2 m$) steps. Therefore, we conclude that the total work is O(m + p log p) * O($\log^2 m$) = O($m \log^2 m + p \log p \log^2 m$). Moreover, if we set the number of tasks in each leaf node as $\log^2 m$, the total work can be minimized to the following [2]:

$$S = O(m + p\log p\, log^2 m)$$

CONCLUSION

In this paper, we investigate multiple approaches of fault-tolerant dynamic task scheduling that deal with various adversarial settings. While these algorithms can handle faults in realistic distributed platforms very well, we must ask ourselves whether there is some algorithm that can do the same job with a lower work complexity. Hence, many ongoing researches are concerned with improving the current best algorithms and achieving a lower cost bound.

REFERENCES

[1] C. Georgiou and A. Shvartsman, Do-all computing in distributed systems. New York: Springer, 2008.

[2] D. Alistarh, M. Bender, S. Gilbert and R. Guerraoui, 'How to Allocate Tasks Asynchronously', in Foundations of Computer Science, New Brunswick, NJ, 2012, pp. 331 - 340.

[3] J. Aspnes, H. Attiya and K. Censor-Hillel, 'Polylogarithmic concurrent data structures from monotone circuits', JACM, vol. 59, no. 1, pp. 1-24, 2012.

[4] S. Dolev, R. Segala and A. Shvartsman, 'Dynamic load balancing with group communication', Theoretical Computer Science, vol. 369, no. 1-3, pp. 348-360, 2006.