Distributed Transactional Contention Management as the Traveling Salesman Problem

Bo Zhang, Binoy Ravindran, and Roberto Palmieri

Virginia Tech, Blacksburg VA 24060, USA,
{alexzbzb,binoy,robertop}@vt.edu

Abstract. In this paper we consider designing contention managers for distributed software transactional memory (DTM), given an input of n transactions sharing s objects in a network of m nodes. We first construct a dynamic ordering conflict graph $G_c^*(\phi(\kappa))$ for an offline algorithm (κ, ϕ_κ). We show that finding an optimal schedule is equivalent to finding the offline algorithm for which the weight of the longest weighted path in $G_c^*(\phi(\kappa))$ is minimized. We further illustrate that when the set of transactions are dynamically generated, processing transactions according to a $\chi(G_c)$-coloring of G_c does not lead to an optimal schedule, where $\chi(G_c)$ is the chromatic number of G_c. We prove that, for DTM, any online work conserving deterministic contention manager provides an $\Omega(\max[s, \frac{s^2}{D}])$ competitive ratio in a network with normalized diameter $\overline{D}$. Compared with the $\Omega(s)$ competitive ratio for multiprocessor STM, the performance guarantee for DTM degrades by a factor proportional to $\frac{s}{D}$. To break this lower bound, we present a randomized algorithm Cutting, which needs partial information of transactions and an approximate algorithm A for the traveling salesman problem with approximation ratio ϕ_A. We show that the average case competitive ratio of Cutting is $O\big(s \cdot \phi_A \cdot \log^2 m \log^2 n\big)$, which is close to $O(s)$.

Keywords: Synchronization, Distributed Transactional Memory

Transactional Memory is an alternative synchronization model for shared memory objects that promises to alleviate the difficulties of manual implementation of lock-based concurrent programs, including composability. The recent integration of TM in hardware by major chip vendors (e.g., Intel, IBM), together with the development of dedicated GCC extensions for TM (i.e., GCC-4.7) has significantly increased TM's traction, in particular its software version (STM). Similar STM, distributed STM (or DTM) is motivated by the difficulties of lock-based distributed synchronization.

In this paper we consider the data-flow DTM model [4], where transactions are immobile, and objects are migrated to invoking transactions. In a realization of this model [8], when a node initiates a transaction that requests a read/write operation on object o, it first checks whether o is in the local cache; if not, it invokes a *cache-coherence* protocol to locate o in the network. If two transactions access the same object at the same time, a contention manager is required to handle the concurrent request. The performance of a contention manager is often

evaluated quantitatively by measuring its *makespan* — the total time needed to complete a finite set of transactions [1]. The goal in the design of a contention manager is often to minimize the makespan, i.e., maximize the throughput.

The first theoretical analysis of contention management in multiprocessors is due to Guerraoui *et. al.* [3], where an $O(s^2)$ upper bound is given for the Greedy manager for s shared objects, compared with the makespan produced by an optimal clairvoyant offline algorithm. Attiya *et. al.* [1] formulated the contention management problem as a *non-clairvoyant* job scheduling problem and improved this bound to $O(s)$. Furthermore, a matching lower bound of $\Omega(s)$ is given for any deterministic contention manager in [1]. To obtain alternative and improved formal bounds, recent works have focused on randomized contention managers [9, 10]. Schneider and Wattenhofer [9] presented a deterministic algorithm called CommitRounds with a competitive ratio $\Theta(s)$ and a randomized algorithm called RandomizedRounds with a makespan $O(C \log M)$ for M concurrent transactions in separate threads with at most C conflicts with high probability. In [10], Sharma *et. al.* consider a set of M transactions and N transactions per thread, and present two randomized contention managers: Offline-Greedy and Online-Greedy. By knowing the conflict graph, Offline-Greedy gives a schedule with makespan $O(\tau \cdot (C + N \log MN))$ with high probability, where each transaction has the equal length τ. Online-Greedy is only $O(\log MN)$ factor worse, but does not need to know the conflict graph. While these works have studied contention management in multiprocessors, no past work has studied it for DTM, which is our focus.

In this paper we study contention management in DTM. Similar to [1], we model contention management as a non-clairvoyant scheduling problem. To find an optimal scheduling algorithm, we construct a dynamic ordering conflict graph $G_c^*(\phi(\kappa))$ for an offline algorithm (κ, ϕ_κ), which computes a k-coloring instance κ of the dynamic conflict graph G_c and processes the set of transactions in the order of ϕ_κ. We show that the makespan of (ϕ, κ) is equivalent to the weight of the longest weighted path in $G_c^*(\phi(\kappa))$. Therefore, finding the optimal schedule is equivalent to finding the offline algorithm (ϕ, κ) for which the weight of the longest weighted path in $G_c^*(\phi(\kappa))$ is minimized. We illustrate that, unlike the one-shot scheduling problem (where each node only issues one transaction), when the set of transactions are dynamically generated, processing transactions according to a $\chi(G_c)$-coloring of G_c does not lead to an optimal schedule, where $\chi(G_c)$ is G_c's chromatic number.

We prove that for DTM, an online, work conserving deterministic contention manager provides an $\Omega(\max[s, \frac{s^2}{\overline{D}}])$ competitive ratio for s shared objects in a network with normalized diameter $\overline{D}$. Compared with the $\Omega(s)$ competitive ratio for multiprocessor STM, the performance guarantee for DTM degrades by a factor proportional to $\frac{s}{\overline{D}}$. This motivates us to design a randomized contention manager that has partial knowledge about the transactions in advance.

We thus develop an algorithm called CUTTING, a randomized algorithm based on an approximate TSP algorithm A with an approximation ratio ϕ_A. CUTTING divides the nodes into $O(C)$ partitions, where C is the maximum degree in the conflict graph G_c. The cost of moving an object inside each partition is at most

$\frac{\text{ATSP}_A}{C}$, where ATSP_A is the total cost of moving an object along the approximate TSP path to visit each node exactly once. CUTTING resolves conflicts in two phases. In the first phase, a binary tree is constructed inside each partition, and a transaction always aborts when it conflicts with its ancestor in the binary tree. In the second phase, CUTTING uses a randomized priority policy to resolve conflicts. We show that the average case competitive ratio of CUTTING is $O\big(s \cdot \phi_A \cdot \log^2 m \log^2 n\big)$ for s objects shared by n transactions invoked by m nodes, which is close to the multiprocessor bound of $O(s)$ [1].

CUTTING is the first ever contention manager for DTM with an average-case competitive ratio bound, and constitutes the paper's contribution.

1 Preliminaries

DTM model. We consider a set of *distributed transactions* $\mathcal{T} := \{T_1, T_2, \ldots, T_n\}$ sharing up to s objects $\mathcal{O} := \{o_1, o_2, \ldots, o_s\}$ distributed on a network of m nodes $\{v_1, v_2, \ldots, v_m\}$, which communicate by message-passing links. For simplicity of the analysis, we assume that each node runs only a single thread, i.e., in total, there are at most m threads running concurrently.[1] A node's thread issues transactions sequentially. Specifically, node v_i issues a sequence of transactions $\{T_1^i, T_2^i, \ldots\}$ one after another, where transaction T_j^i is issued once after T_{j-1}^i has committed.

An execution of a transaction is a sequence of timed operations. There are four operation types that a transaction may take: *write*, *read*, *commit*, and *abort*. An execution ends by either a commit (success) or an abort (failure). When a transaction aborts, it is restarted from its beginning immediately and may access a different set of shared objects. Each transaction T_i has a *local execution duration* τ_i, which is the time T_i executes locally without contention (or interruption). Note that τ_i does not include the time T_i acquires remote objects. In our analysis, we assume a fixed τ_i for each transaction T_i. Such a general assumption is unrealistic if the local execution duration depends on the properties of specific objects. In that case, when a transaction alters the set of requested objects after it restarts, the local execution duration also varies. Therefore, if the local execution duration varies by a factor of c, then the performance of our algorithms would worsen by the same factor c.

A transaction performs a read/write operation by first sending a read/write access request through CC. For a read operation, the CC protocol returns a read-only copy of the object. An object can thus be read by an arbitrary number of transactions simultaneously. For a write operation, the CC protocol returns the (writable) object itself. At any time, only one transaction can hold the object exclusively. A contention manager is responsible for resolving the conflict, and does so by aborting or delaying (i.e., postponing) one of the conflicting transactions.

[1] When a node runs multiple threads, our analysis can still be adopted by treating each thread as an individual node. This strategy overlooks the possible local optimization for the same threads issued by the same node. Therefore, multiprocessor contention management strategy can be used to improve performance.

A CC protocol moves objects via a specific path (e.g., the shortest path for Ballistic [4], or a path in a spanning tree for Relay [11]). We assume a fixed CC protocol with a *moving cost* d_{ij}, where d_{ij} is the communication latency to move an object from node v_i to v_j under that protocol. We can build a complete *cost graph* $G_d = (V_d, E_d)$, where $|V_d| = m$ and for each edge $(v_i, v_j) \in E_d$, the weight is d_{ij}. We assume that the moving cost is bounded: we can find a constant D such that for any d_{ij}, $D \geq d_{ij}$.

Conflict graph. We build the *conflict graph* $G_c = (\mathcal{T}_c, E_c)$ for the transaction subset $\mathcal{T}_c \subseteq \mathcal{T}$, which runs concurrently. An edge $(T_i, T_j) \in E_c$ exists if and only if T_i and T_j conflict. Two transactions conflict if they both access the same object and at least one of the accesses is a write. Let N_T denote the set of neighbors of T in G_c. The degree $\delta(T) := |N_T|$ of a transaction T corresponds to the number of neighbors of T in G_c. We denote $C = \max_i \delta(T_i)$, i.e., the maximum degree of a transaction. The graph G_c is dynamic and only consists of live transactions. A transaction joins $\mathcal{T}_c$ after it (re)starts, and leaves $\mathcal{T}_c$ after it commits/aborts. Therefore, N_T, $\delta(T)$, and C only capture a "snapshot" of G_c at a certain time. More precisely, they should be represented as functions of time. When there is no ambiguity, we use the simplified notations. We have $|\mathcal{T}_c| \leq \min\{m, n\}$, since there are at most n transactions, and at most m transactions can run in parallel. Then we have $\delta(T) \leq C \leq \min\{m, n\}$.

Let $\boldsymbol{o}(T_i) := \{o_1(T_i), o_2(T_i), \ldots\}$ denote the sequence of objects requested by transaction T_i. Let $\gamma(o_j)$ denote the number of transactions in $\mathcal{T}_c$ that concurrently writes o_i and $\gamma_{max} = \max_j \gamma(o_j)$. Let $\lambda(T_i) = \{o : o \in \boldsymbol{o}(T_i) \wedge (\gamma(o) \geq 1)\}$ denote the number of transactions in $\mathcal{T}_c$ that conflict with transaction T_i and $\lambda_{max} = \max_{T_i \subset \mathcal{T}_c} \lambda(T_i)$. We have $C \leq \lambda_{max} \cdot \gamma_{max}$ and $C \geq \gamma_{max}$.

2 The DTM Contention Management Problem

2.1 Problem measure and complexity

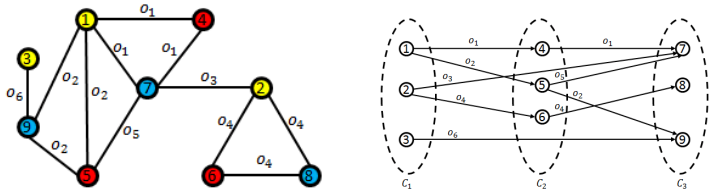
A contention manager determines when a particular transaction executes in case of a conflict. To quantitatively evaluate the performance of a contention manager, we measure the *makespan*, which is the total time needed to complete a set of transactions $\mathcal{T}$. Formally, given a contention manager A, makespan_A denotes the time needed to complete all transactions in $\mathcal{T}$ under A.

We measure the contention manager's quality, by assuming Opt, the optimal, centralized, clairvoyant scheduler which has the complete knowledge of each transaction (requested objects, locations, released time, local execution time). The quality of a contention manager A is measured by the ratio $\frac{\text{makespan}_A}{\text{makespan}_{\text{Opt}}}$, called the *competitive ratio* of A on $\mathcal{T}$. The competitive ratio of A is $\max_{\mathcal{T}} \frac{\text{makespan}_A}{\text{makespan}_{\text{Opt}}}$, i.e., the maximum competitive ratio of A over all possible workloads.

An ideal contention manager aims to provide an optimal schedule for any given set of transactions. However, it is shown in [1] (for STM) that if there exists an adversary to change the set of shared objects requested by any transaction arbitrarily, no algorithm can do better than a simple sequential execution. Furthermore, even if the adversary can only choose the initial conflict graph and

does not influence it afterwards, it is NP-hard to get a reasonable approximation of an optimal schedule [9].

We can consider the transaction scheduling problem for multiprocessor STM as a subset of the transaction scheduling problem for DTM. The two problems are equivalent as long as the communication cost (d_{ij}) can be ignored, compared with the local execution time duration (τ_i). Therefore, extending the problem space into distributed systems only increases the problem complexity.



(a) Conflict graph G_c. (b) Ordering conflict graph $G_c(\phi_\kappa)$.

Fig. 1.

We depict an example of a conflict graph G_c in Figure 1(a), which consists of 9 write-only transactions. Each transaction is represented as a numbered node in G_c. Each edge (T_i, T_j) is marked with the object which causes T_i and T_j to conflict (e.g., T_1 and T_4 conflict on o_1). We can construct a coloring of the conflict graph $G_c = (\mathcal{T}_c, E)$. A 3-coloring scenario is illustrated in Figure 1(a). Transactions are partitioned into 3 sets: $C_1 = \{T_1, T_2, T_3\}, C_2 = \{T_4, T_5, T_6\}, C_3 = \{T_7, T_8, T_9\}$. Since transactions with the same color are not connected, every set $C_i \subset \mathcal{T}_c$ forms an independent set and can be executed in parallel without facing any conflicts. With the same argument of [1], we have the following lemma.

Lemma 1 *An optimal offline schedule* Opt *determines a k-coloring κ of the conflict graph G_c and an execution order ϕ_κ such that for any two sets $C_{\phi_\kappa(i)}$ and $C_{\phi_\kappa(j)}$, where $i < j$, if (1) $T_1 \in C_{\phi_\kappa(i)}$, $T_2 \in C_{\phi_\kappa(j)}$, and (2) T_1 and T_2 conflict, then T_2 is postponed until T_1 commits.*

In other words, Opt determines the order in which an independent set C_i is executed. Generally, for a k-coloring of G_c, there are $k!$ different choices to order the independent sets. Assume that for the 3-coloring example in Figure 1(a), an execution order $\phi_\kappa = \{C_1, C_2, C_3\}$ is selected. We can construct an *ordering conflict graph* $G_c(\phi_\kappa)$, as shown in Figure 1(b).

Definition 1 (Ordering conflict graph) *For the conflict graph G_c, given a k-coloring instance κ and an execution order $\{C_{\phi_\kappa(1)}, C_{\phi_\kappa(2)}, \ldots, C_{\phi_\kappa(k)}\}$, the ordering conflict graph $G_c(\phi_\kappa) = (\mathcal{T}_c, E(\phi_\kappa), w)$ is constructed. $G_c(\phi_\kappa)$ has the following properties:*

1. *$G_c(\phi_\kappa)$ is a weighted directed graph.*
2. *For two transactions $T_1 \in C_{\phi_\kappa(i)}$ and $T_2 \in C_{\phi_\kappa(j)}$, a directed edge (or an arc) $(T_1, T_2) \in E(\phi_\kappa)$ (from T_1 to T_2) exists if: (i) T_1 and T_2 conflict over*

object o; (ii) $i < j$; and (iii) $\nexists T_3 \in C_{\phi_\kappa(j')}$, where $i < j' < j$, such that T_1 and T_3 also conflict over o.

3. *The weight $w(T_i)$ of a transaction T_i is τ_i; the weight $w(T_i, T_j)$ of an arc (T_i, T_j) is d_{ij}.*

For example, the edge (T_1, T_4) in Figure 1(a) is also an arc in Figure 1(b). However, the edge (T_1, T_7) in Figure 1(a) no longer exists in Figure 1(b), because C_2 is ordered between C_1 and C_3, and T_1 and T_4 also conflict on o_1.

Hence, any offline algorithm can be described by the pair (κ, ϕ_κ), and the ordering conflict graph $G_c(\phi_\kappa)$ can be constructed. Given $G_c(\phi_\kappa)$, the execution time of each transaction can be determined.

Theorem 2 *For the ordering conflict graph $G_c(\phi_\kappa)$, given a directed path $P = \{T_{P(1)}, T_{P(2)}, \ldots, T_{P(L)}\}$ of L hops, the weight of P is defined as $w(P) = \sum_{1 \le i \le L} w(T_{P(i)}) + \sum_{1 \le j \le L-1} w(T_{P(j)}, T_{P(j+1)})$. Then transaction $T_0 \in \mathcal{T}_c$ commits at time: $\max_{P=\{T_{P(1)},\ldots,T_0\}} t_{P(1)} + w(P)$, where $T_{P(1)}$ starts at time $t_{P(1)}$.*

Proof. We prove the theorem by induction. Assume $T_0 \in C_{\phi_\kappa(j)}$. When j=1, T_0 executes immediately after it starts. At time $t_0+\tau_0$, T_0 commits. There is only one path that ends at T_0 in $G_c(\phi_\kappa)$ (which only contains T_0). The theorem holds.

Assume that when $j = 2, 3, \ldots, q-1$, the theorem holds. Let $j = q$. For each object $o_i \in \boldsymbol{o}(T_0)$, find the transaction $T_{0(i)}$ such that $T_{0(i)}$ and T_0 conflict over o_i, and $(T_{0(i)}, T_0) \in E(\phi_\kappa)$. If no such transaction exists for all objects, the analysis falls into the case when $j = 1$. Otherwise, for each transaction $T_{0(i)}$, from Definition 1, no transaction which requests access to o_i is scheduled between $T_{0(i)}$ and T_0. The offline algorithm (κ, ϕ_κ) moves o_i from $T_{0(i)}$ to T_0 immediately after $T_{0(i)}$ commits. Assume that $T_{0(i)}$ commits at $t^c_{0(i)}$. Then T_0 commits at time: $\max_{o_i \in \boldsymbol{o}(T_0)} t^c_{0(i)} + w(T_{0(i)}, T_0) + w(T_0)$. Since $(T_{0(i)}, T_0) \in E(\phi_\kappa)$, then from the induction step, we know that $t^c_{0(i)} = \max_{P=\{T_{P(1)},\ldots,T_{0(i)}\}} t_{P(1)} + w(P)$. Hence, by replacing $t^c_{0(i)}$ with $\max_{P=\{T_{P(1)},\ldots,T_{0(i)}\}} t_{P(1)} + w(P)$, the theorem follows.

Theorem 2 illustrates that the commit time of transaction T_0 is determined by one of the *weighted paths* in $G_c(\phi_\kappa)$ which ends at T_0. Specifically, if every node issues its first transaction at the same time, the commit time of T_0 is solely determined by the longest weighted path in $G_c(\phi_\kappa)$ which ends at T_0. However, when transactions are dynamically generated over time, the commit time of a transaction also relies on the starting time of other transactions. To accommodate the dynamic features of transactions, we construct the *dynamic ordering conflict graph* $G^*_c(\phi_\kappa)$ based on $G_c(\phi_\kappa)$.

Definition 2 (Dynamic ordering conflict graph) *Given an ordering conflict graph $G_c(\phi_\kappa)$, the dynamic ordering conflict graph $G^*_c(\phi_\kappa)$ is constructed by making the following modifications on $G_c(\phi_\kappa)$:*

1. *For the sequence of transactions $\{T^i_1, T^i_2, \ldots, T^i_L\}$ issued by each node v_i, an arc (T^i_{j-1}, T^i_j) is added to $G^*_c(\phi_\kappa)$ for $2 \le j \le L$ and $w(T^i_{j-1}, T^i_j) = 0$.*
2. *If transaction T_j which starts at t_j does not have any incoming arcs in $G^*_c(\phi_\kappa)$, then $w(T_j) = t_j + \tau_j$.*

Theorem 3 *The makespan of algorithm* (κ, ϕ_κ) *is the weight of the longest weighted path in* $G_c^*(\phi_\kappa)$*: makespan*$_{(\kappa,\phi_\kappa)} = \max_{P \in G_c^*(\phi_\kappa)} w(P)$

Proof. We start the proof with special cases, and then extend the analysis to the general case. Assume that (i) each node issues only one transaction, and (ii) all transactions start at the same time. Then the makespan of (κ, ϕ_κ) is equivalent to the execution time of the last committed transaction: $\text{makespan}_{(\kappa,\phi_\kappa)} = \max_{T_0 \in \mathcal{T}_c, P \in G_c(\phi_\kappa), P=\{\ldots,T_0\}} w(P) = \max_{P \in G_c(\phi_\kappa)} w(P) = \max_{P \in G_c^*(\phi_\kappa)} w(P)$. Then, we can progressively relax the assumptions and use Theorem 2 to prove this theorem. Due to space constraints, report the detailed proof in Appendix A.1.

Theorem 3 shows that, given an offline algorithm (κ, ϕ_κ), finding its makespan is equivalent to finding the longest weighted path in the dynamic ordering conflict graph $G_c^*(\phi_\kappa)$. Therefore, the optimal schedule OPT is the offline algorithm which minimizes the makespan.

Corollary 4 $\textit{makespan}_{\text{OPT}} = \min_{\kappa,\phi_\kappa} \max_{P \in G_c^*(\phi_\kappa)} w(P)$

It is easy to show that finding the optimal schedule is NP-hard. For the *one-shot scheduling problem*, where each node issues a single transaction, if $\tau_0 = \tau$ for all transactions $T_0 \in \mathcal{T}$ and $D \ll \tau$, the problem becomes the classical node coloring problem. Finding the optimal schedule is equivalent to finding the chromatic number $\chi(G_c)$. As [7] shows, computing an optimal coloring, given complete knowledge of the graph, is NP-hard, and computing an approximation within the factor of $\chi(G_c)^{\frac{\log \chi(G_c)}{25}}$ is also NP-hard.

If $s = 1$, i.e., there is only one object shared by all transactions, finding the optimal schedule is equivalent to finding the traveling salesman problem (TSP) path in G_d, i.e., the shortest hamiltonian path in G_d. When the cost metric d_{ij} satisfies the triangle inequality, the resulting TSP is called the metric TSP, and has been shown to be NP-complete by Karp [6]. If the cost metric is symmetric, Christofides [2] presented an algorithm approximating the metric TSP within approximation ratio 3/2. If the cost metric is asymmetric, the best known algorithm approximates the solution within approximation ratio $O(\log m)$ [5].

When each node generates a sequence of transactions dynamically, it is not always optimal to schedule transactions according to a $\chi(G_c)$-coloring. Since the conflict graph evolves over time, an optimal schedule based on a static conflict graph may lose potential parallelism in the future. In the dynamic ordering conflict graph, a temporarily-optimal scheduling does not imply that the resulting longest weighted path is optimal. An example that illustrates this claim is shown in Appendix A.2.

2.2 Lower bound

Our analysis shows that to compute an optimal schedule, even knowing all information about the transactions in advance, is NP-hard. Thus, we design an online algorithm which guarantees better performance than that can be obtained by simple serialization of all transactions. Before designing the contention manager, we need to know what performance bound an online contention manager could provide in the best case. We first introduce the *work conserving* property [1]:

Definition 3 *A scheduling algorithm is work conserving if it always runs a maximal set of non-conflicting transactions.*

In [1], Attiya *et al.* showed that, for multiprocessor STM, a deterministic work conserving contention manager is $\Omega(s)$-competitive, if the set of objects requested by a transaction changes when the transaction restarts. We prove that for DTM, the performance guarantee is even worse.

Theorem 5 *For DTM, any online, work conserving deterministic contention manager is* $\Omega(\max[s, \frac{s^2}{\overline{D}}])$*-competitive, where* $\overline{D} := \frac{D}{\min_{G_d} d_{ij}}$ *is the* normalized diameter *of the cost graph* G_d.

Proof. The proof uses s^2 transactions with the same local execution duration τ. A transaction is denoted by T_{ij}, where $1 \leq i, j \leq s$. Each transaction T_{ij} contains a sequence of two operations $\{R_i, W_i\}$, which first reads from object o_i and then writes to o_i. Each transaction T_{ij} is issued by node v_{ij} at the same time, and object o_i is held by node v_{i1} when the system starts. For each i, we select a set of nodes $V_i := \{v_{i1}, v_{i2}, \ldots, v_{is}\}$ within the range of the diameter $D_i \leq \frac{D}{s}$.

Consider the optimal schedule Opt. Note that all transactions form an $s \times s$ matrix, and transactions from the same row ($\{T_{i1}, T_{i2}, \ldots, T_{is}\}$ for $1 \leq i \leq s$) have the same operations. Therefore, at the start of the execution, Opt selects one transaction from each row, thus s transactions start to execute. Whenever T_{ij} commits, Opt selects one transaction from the rest of the transactions in row i to execute. Hence, at any time, there are s transactions that run in parallel.

The order that Opt selects transactions from each row is crucial: Opt should select transactions in the order such that the weight of the longest weighted path in $G_c^*(\text{Opt})$ is optimal. Since transactions from different rows run in parallel, we have: $\text{makespan}_{\text{Opt}} = s \cdot \tau + \max_{1 \leq i \leq s} \text{Tsp}(G_d(o_i))$, where $G_d(o_i)$ denotes the subgraph of G_d induced by s transactions requesting o_i, and $\text{Tsp}(G_d(o_i))$ denotes the length of the TSP path of $G_d(o_i)$, i.e., the shortest path that visits each node exactly once in $\text{Tsp}(G_d(o_i))$.

Now consider an online, work conserving deterministic contention manager A. Being work conserving, it must select to execute a maximal independent set of non-conflicting transactions. Since the first access of all transactions is a read, the contention manager starts to execute all s^2 transactions.

After the first read operation, for each row i, all transactions in row i attempt to write o_i, but only one of them can commit and the others will abort. Otherwise, atomicity is violated, since inconsistent states of some transactions may be accessed. When a transaction restarts, the adversary determines that all transactions change to write to the same object, e.g., $\{R_i, W_1\}$. Therefore, the rest $s^2 - s$ transactions can only be executed sequentially after the first s transactions execute in parallel and commit. Then we have: $\text{makespan}_A \geq (s^2 - s + 1) \cdot \tau + \min_{G_d} \text{Tsp}(G_d(s^2 - s + 1))$, where $G_d(s^2 - s + 1)$ denotes the subgraph of G_d induced by a subset of $s^2 - s + 1$ transactions.

Now, we can compute A's competitive ratio. We have: $\frac{\text{makespan}_A}{\text{makespan}_{\text{Opt}}} \geq \max\left[\frac{(s^2-s+1)\cdot\tau}{s\cdot\tau}, \frac{\min_{G_d} \text{Tsp}(G_d(s^2-s+1))}{\max_{1\leq i\leq s} \text{Tsp}(G_d(o_i))}\right] \geq \max[\frac{s^2-s+1}{s}, \frac{(s^2-s+1)\cdot\min_{G_d} d_{ij}}{(s-1)\cdot\frac{D}{s}}] = \Omega(\max[s, \frac{s^2}{\overline{D}}])$. The theorem follows.

Theorem 5 shows that for DTM, an online, work conserving deterministic contention manager cannot provide a similar performance guarantee compared with multiprocessor STM. When the normalized network diameter is bounded (i.e., $\overline{D}$ is a constant, where new nodes join the system without expanding the diameter of the network), it can only provide an $\Omega(s^2)$-competitive ratio. In the next section, we present an online randomized contention manager, which needs partial information of transactions in advance, in order to provide a better performance guarantee.

3 Algorithm: Cutting

3.1 Description

We present the algorithm Cutting (the pseudo-code is shown in Algorithm 1 of Appendix A.3), a randomized scheduling algorithm based on a partitioning constructed on the cost graph G_d. To partition the cost graph, we first construct an approximate TSP path (ATSP path) in G_d $\textsc{Atsp}_A(G_d)$ by selecting an approximate TSP algorithm A. Specifically, A provides the approximation ratio ϕ_A, such that for any graph G, $\frac{\textsc{Atsp}_A(G)}{\textsc{Tsp}(G)} = O(\phi_A)$. Note that if d_{ij} satisfies the triangle inequality, the best known algorithm provides an $O(\log m)$ approximation [5]; if d_{ij} is symmetric as well, a constant ϕ_A is achievable [2]. We assume that a transaction has partial knowledge in advance: a transaction T_i knows its required set of objects $\boldsymbol{o}_i$ after it starts. Therefore, a transaction can send all its object requests immediately after it starts.

Based on the constructed ATSP path $\textsc{Atsp}_A$, we define the (C, A) *partitioning* on G_d, which divides G_d into $O(C)$ partitions. A constructed partition P is a subset of nodes, which satisfies either: 1) $|P| = 1$; or 2) for any pair of nodes $(v_i, v_j) \in P$, $d_{ij} \leq \frac{\textsc{Atsp}_A}{C}$.

Definition 4 ((C, A) partitioning) *In the cost graph G_d, the (C, A) partitioning $\mathcal{P}(C, A, v)$ divides m nodes into $O(C)$ partitions in two phases.*

Phase I. *Randomly select a node v, and let node v^j be the j^{th} node (excluding v) on the ATSP path $\textsc{Atsp}_A(G_d)$ starting from v. Hence, $\textsc{Atsp}_A(G_d)$ can be represented by a sequence of nodes $\{v^0, v^1, \ldots, v^{m-1}\}$.* ***Phase II.*** *Inside each partition $P_t = \{v^k, v^{k+1}, \ldots\}$, each node v^k is assigned a* partition index $\psi(v^j) = (j \mod k)$*, i.e., its index inside the partition.*

1. *Starting from v^0, add v^0 to P_1, and set P_1 as the current partition.*
2. *Check v^1. If $\textsc{Atsp}_A(G_d)[v^0, v^1] \leq \frac{\textsc{Atsp}_A(G_d)}{C}$, where $\textsc{Atsp}_A(G_d)[v^1, v^2]$ is the length of the part of $\textsc{Atsp}_A(G_d)$ from v^0 to v^1, add v^1 to P_1. Else, add v^1 to P_2, and set P_2 as the current partition.*
3. *Repeat Step 2 until all nodes are partitioned. For each node v^k and the current partition P_t, this process checks the length of $\textsc{Atsp}_A(G_d)[v^j, v^k]$, where v^j is the first element added to P_t. If $\textsc{Atsp}_A(G_d)[v^j, v^k] \leq \frac{\textsc{Atsp}_A(G_d)}{C}$, v^k is added to P_t. Else, v^k is added to P_{t+1}, and P_{t+1} is set as the current partition.*

The conflict resolution also has two phases. In the first phase, CUTTING assigns each transaction a partition index. When two transactions T_1 (invoked by node v^{j_1}) and T_2 (invoked by node v^{j_2}) conflict, the algorithm checks: 1) whether they are from the same partition P_t; 2) If so, whether $\exists$ integer $\nu \geq 1$ such that $\lfloor \frac{\max\{\psi(v^{j_1}),\psi(v^{j_2})\}}{2^\nu} \rfloor = \min\{\psi(v^{j_1}), \psi(v^{j_2})\}$. Note that by checking these two conditions, an underlying binary tree $\text{BT}(P_t)$ is constructed in P_t as follows:

1. Set v^{j_0} as the root of $\text{BT}(P_t)$ (level 1), where $\psi(v^{j_0} = 0)$, i.e., the first node added to P_t.
2. Node v^{j_0}'s left pointer points to v^{j_0+1} and right pointer points to v^{j_0+2}. Nodes v^{j_0+1} and v^{j_0+2} belong to level 2.
3. Repeat Step 2 by adding nodes sequentially to each level from left to right. In the end, $O(\log_2 m)$ levels are constructed.

Note that by satisfying these two conditions, the transaction with the smaller partition index must be an *ancestor* of the other transaction in $\text{BT}(P_t)$. Therefore, a transaction may conflict with at most $O(\log_2 m)$ ancestors in this case. CUTTING resolves the conflict greedily so that the transaction with the smaller partition index always aborts the other transaction.

In the second phase, each transaction selects an integer $\pi \in [1, m]$ randomly when it starts or restarts. If one transaction is not an ancestor of another transaction, the transaction with the lower π proceeds and the other transaction aborts. Whenever a transaction is aborted by a remote transaction, the requested object is moved to the remote transaction immediately.

3.2 Analysis

We now study two efficiency measures of CUTTING from the average-case perspective: the average response time (how long it takes for a transaction to commit on average) and the average makespan.

Lemma 6 *A transaction T needs $O\big(C \log^2 m \log n\big)$ trials from the moment it is invoked until it commits, on average.*

Proof. We start from a transaction T invoked by the root node $v^\psi \in \text{BT}(P_t)$. Since v^ψ is the root, T cannot be aborted by another ancestor in $\text{BT}(P_t)$. Hence, T can only be aborted when it chooses a larger π than π', which is the integer chosen by a conflicting transaction T' invoked by node $v^{\psi'} \in P_{t'}$. The probability that for transaction T, no transaction $T' \in N_T$ selects the same random number $\pi' = \pi$ is: $\mathbf{Pr}(\nexists T' \in N_T | \pi' = \pi) = \prod_{T' \in N_T}(1 - \frac{1}{m}) \geq (1 - \frac{1}{m})^{\delta(T)} \geq (1 - \frac{1}{m})^m \geq \frac{1}{e}$. Note that $\delta(T) \leq C \leq m$. On the other hand, the probability that π is at least as small as π' for any conflicting transaction T' is at least $\frac{1}{(C+1)}$. Thus, the probability that π is the smallest among all its neighbors is at least $\frac{1}{e(C+1)}$.

We use the following Chernoff bound:

Lemma 7 *Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \leq i \leq n$, $\boldsymbol{Pr}(X_i = 1) = p_i$, where $0 \leq p_i \leq 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = \boldsymbol{E}[X] = \sum_{i=1}^n p_i$, and any $\delta \in (0, 1]$, $\boldsymbol{Pr}(X < (1 - \delta)\mu) < e^{-\delta^2 \mu/2}$.*

By Lemma 7, if we conduct $16e(C+1)\ln n$ trials, each having a success probability $\frac{1}{e(C+1)}$, then the probability that the number of successes X is less than $8\ln n$ becomes: $\mathbf{Pr}(X < 8\ln n) < e^{-2\ln n} = \frac{1}{n^2}$.

Now we examine the transaction T^l invoked by node $v^{\psi^l} \in P_t$, where v^{ψ^l} is the left child of the root node v^{ψ} in $\textsc{Bt}(P_t)$. When T^l conflicts with T, it aborts and holds off until T commits or aborts. Hence, T^l can be aborted by T at most $16e(C+1)\ln n$ times with probability $1-\frac{1}{n^2}$. On the other hand, T^l needs at most $16e(C+1)\ln n$ to choose the smallest integer among all conflicting transactions with probability $1-\frac{1}{n^2}$. Hence, in total, T^l needs at most $32e(C+1)\ln n$ trials with probability $(1-\frac{1}{n^2})^2 > (1-\frac{2}{n^2})$.

Therefore, by induction, the transaction T^L invoked by a level-L node v^{ψ^L} of $\textsc{Bt}(P_t)$ needs at most $(1+\log_2 L)\log_2 L \cdot 8e(C+1)\ln n$ trials with probability at least $1-\frac{(1+\log_2 L)\log_2 L}{2n^2}$. Now, we can calculate the average number of trials: $\mathbf{E}[\#\ \textit{of trials a transaction needs to commit}] = O\big(C\log^2 m \log n\big)$.

Since when the starting point of the ATSP path is randomly selected, the probability that a transaction is located at level L is $1/2^{L_{max}-L+1}$. The lemma follows.

Lemma 8 *The average response time of a transaction is* $O\big(C\log^2 m\log n\cdot(\tau+\frac{\textsc{Atsp}_A}{C})\big)$.

Proof. From Lemma 6, each transaction needs $O\big(C\log^2 m\log n\big)$ trials, on average. We now study the duration of a trial, i.e., the time until a transaction can select a new random number. Note that a transaction can only select a new random number after it is aborted (locally or remotely). Hence, if a transaction conflicts with a transaction in the same partition, the duration is at most $\tau+\frac{\textsc{Atsp}_A}{C}$; if it conflicts with a transaction in another partition, the duration is at most $\tau+D$. Note that a transaction sends its requests of objects simultaneously once after it (re)starts. If a transaction conflicts with multiple transactions, the first conflicting transaction it knows is the transaction closest to it. From Lemma 6, a transaction can be aborted by transactions from other partitions by at most $16e(C+1)\ln n$ times. Hence, the expected commit time of a transaction is $O\big(C\log^2 m\log n\cdot(\tau+\frac{\textsc{Atsp}_A}{C})\big)$. The lemma follows.

Theorem 9 *The average-case competitive ratio of* Cutting *is* $O\big(s\cdot\phi_A\cdot\log^2 m\log^2 n\big)$.

Proof. By following the Chernoff bound provided by Lemma 7 and Lemma 8, we can prove that Cutting produces a schedule with average-case makespan $O\big(C\log^2 m\log n\cdot(\tau+\frac{\textsc{Atsp}_A}{C})+(N\cdot\log^2 m\log^2 n\cdot\tau+\textsc{Atsp}_A)\big)$, where N is the maximum number of transactions issued by the same node. We then find that $\text{makespan}_{\textsc{Opt}} \geq \max_{1\leq i\leq s}\big(\tau\cdot\max[\gamma_i,N]+\textsc{Tsp}(G_d(o_i))\big)$, since γ_i transactions concurrently conflict on object o_i. Hence, at any given time, only one of them can commit, and the object moves along a certain path to visit γ_i transactions one after another. Then we have: $\text{makespan}_{\textsc{Opt}} \geq \max_{1\leq i\leq s}\big(\tau\cdot\max[\gamma_i,N]+\textsc{Tsp}(G_d(o_i))\big) \geq \tau\cdot\max[\frac{\sum_{1\leq i\leq s}\gamma_i}{s},N]+\frac{\sum_{1\leq i\leq s}\textsc{Tsp}(G_d(o_i))}{s}$. Therefore, the competitive ratio of Cutting is: $\frac{\text{makespan}_{\textsc{Cutting}}}{\text{makespan}_{\textsc{Opt}}} = s\cdot\log^2 m\log^2 n\cdot$

$\frac{\tau \cdot C + \text{ATSP}_A}{\tau \cdot \sum_{1 \le i \le s} \gamma_i + \sum_{1 \le i \le s} \text{TSP}(G_d(o_i))}$. Note that $C \le \sum_{1 \le i \le s} \gamma_i$ and $\sum_{1 \le i \le s} \text{TSP}(G_d(o_i)) \ge \text{TSP}(G_d)$. The theorem follows.

4 Conclusions

CUTTING provides an efficient average-case competitive ratio. This is the first such result for the design of contention management algorithms for DTM. The algorithm requires that each transaction be aware of its requested set of objects when it starts. This is essential in our algorithms, since each transaction can send requests to objects simultaneously after it starts. If we remove this restriction, the original results do not hold, since a transaction can only send the request of an object once after the previous operation is done. This increases the resulting makespan by a factor of $\Omega(s)$.

References

1. H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC*, 2006.
2. N. Christofides. Worst case analysis of a new heuristic for the traveling salesman problem. Technical Report CS-93-13, G.S.I.A., Carnegie Mellon University, Pittsburgh, USA, 1976.
3. R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC*, 2005.
4. M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.
5. H. Kaplan, M. Lewenstein, N. Shafrir, and M. Sviridenko. Approximation algorithms for asymmetric tsp by decomposing directed regular multigraphs. *J. ACM*, 52:602–626, 2005.
6. R. M. Karp. Reducibility among combinatorial problems. In *R. E. Miller and J. W. Thatcher (editors). Complexity of Computer Computations*, pages 85–103, 1972.
7. S. Khot. Improved inapproximability results for maxclique, chromatic number and approximate graph coloring. In *Proc. 42nd IEEE Symp. on Foundations of Computer Science*, pages 600–609, 2001.
8. M. M. Saad and B. Ravindran. Distributed hybrid-flow STM. In *HPDC*, 2011.
9. J. Schneider and R. Wattenhofer. Bounds On Contention Management Algorithms. In *ISAAC*, 2009.
10. G. Sharma, B. Estrade, and C. Busch. Window-based greedy contention management for transactional memory. In *DISC*, 2010.
11. B. Zhang and B. Ravindran. Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In *OPODIS*, 2009.

A Apendix

A.1 Detailed Proof of Theorem 3

Theorem 3. *The makespan of algorithm (κ, ϕ_κ) is the weight of the longest weighted path in $G_c^*(\phi_\kappa)$: makespan$_{(\kappa,\phi_\kappa)} = \max_{P \in G_c^*(\phi_\kappa)} w(P)$*

Proof. We start the proof with special cases, and then extend the analysis to the general case. Assume that (i) each node issues only one transaction, and (ii) all transactions start at the same time. Then the makespan of (κ, ϕ_κ) is equivalent to the execution time of the last committed transaction: makespan$_{(\kappa,\phi_\kappa)} = \max_{T_0 \in \mathcal{T}_c, P \in G_c(\phi_\kappa), P=\{\ldots,T_0\}} w(P) = \max_{P \in G_c(\phi_\kappa)} w(P) = \max_{P \in G_c^*(\phi_\kappa)} w(P)$. Now, we relax the second assumption: each node issues a single transaction at arbitrary time points. Let P be the path which maximizes makespan$_{(\kappa,\phi_\kappa)}$. Therefore, T_{P_1} (the head of P) has no incoming arcs in $G_c^*(\phi_\kappa)$ (since each node only issues a single transaction). From the construction of $G_c^*(\phi_\kappa)$, $w(T_{P_1}) = t(P_1) + \tau_{P_1}$. We can find a path P^* in $G_c^*(\phi_\kappa)$ which contains the same elements as P with weight $w(P^*) = t(P_1) + w(P)$, which is the longest path in $G_c^*(\phi_\kappa)$.

Now, we relax the first assumption: each node issues a sequence of transactions, and all nodes start their first transactions at the same time. Similar to the first case, we have: makespan$_{(\kappa,\phi_\kappa)} =$
$\max_{P \in G_c(\phi_\kappa), P=\{T_{P_1},\ldots,T_0\}} t(P_1) + w(P)$.

Let P be the path which maximizes makespan$_{(\kappa,\phi_\kappa)}$. If T_{P_1} (the head of P) is the first transaction issued by a node, the theorem follows. Otherwise, $\forall o_i \in \boldsymbol{o}(T_{P_1})$, T_{P_1} is the first transaction scheduled to access o_i by (κ, ϕ_κ), because there is no incoming arc to T_{P_1} in $G_c(\phi_\kappa)$. If T_{P_1} is the l^{th} transaction issued by node v_j, when we convert from $G_c(\phi_\kappa)$ to $G_c^*(\phi_\kappa)$, the longest path P^* that ends at T_0 is a path starting from T_1^j to T_{l-1}^j, followed by an arc (T_{l-1}^j, T_{P_1}), and then followed by P. Note that T_{l-1}^j commits at t_{P_1} (the starting time of T_{P_1}). Hence, we have $w(P^*) = t(P_1) + w(T_{l-1}^j, T_{P_1}) + w(P)$. Since $w(T_{l-1}^j, T_{P_1}) = 0$ (from the construction of $G_c^*(\phi_\kappa)$), we have $t(P_1) + w(P) = w(P^*)$. We conclude that the path in $G_c(\phi_\kappa)$ corresponding to the commit time of transaction T_0 is equivalent to the longest path which ends at T_0 in $G_c^*(\phi_\kappa)$. The theorem follows.

A.2 Illustrative Example Conflict Graph

Consider an 8-node network, where each node issues two transactions sequentially. Figure 2 shows the conflict graph of the transactions. Each transaction in the set $\{T_1^1, T_1^2, \ldots, T_1^8\}$ (the first transaction issued by each node) requests accesses to three objects, and each object is requested by two transactions. In total, twelve objects (o_1 – o_{12}) are shared among eight transactions. Object o_{13} is requested by all four transactions of the set $\{T_2^1, T_2^3, T_2^6, T_2^8\}$, and object o_{14} is requested by all four transactions of the set $\{T_2^2, T_2^4, T_2^5, T_2^7\}$. For each node v_i, transaction T_2^i can only start after T_1^i commits.

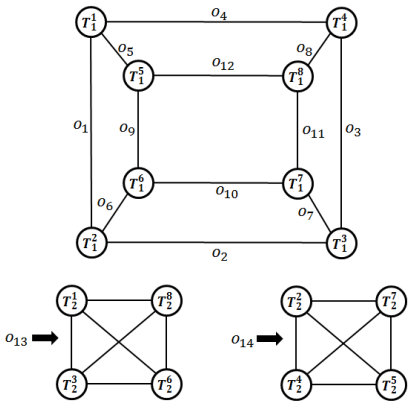


Fig. 2. Example 1: conflict graph G_c.

The dynamic ordering conflict graphs of two different offline algorithms (ϕ_{κ_1} and ϕ_{κ_2}) are depicted in Figures 3(a) and (b), respectively. Note that, initially, G_c only contains $\{T_1^1, T_1^2, \ldots, T_1^8\}$. Algorithm ϕ_{κ_1} selects a 2-coloring of G_c, which is temporarily optimal: $\{T_1^1, T_1^3, T_1^6, T_1^8\}$ and $\{T_1^2, T_1^4, T_1^5, T_1^7\}$ are selected as the two 2-coloring sets, and they execute one after another. After $\{T_1^1, T_1^3, T_1^6, T_1^8\}$ commits, $\{T_2^1, T_2^3, T_2^6, T_2^8\}$ starts to execute. However, since transactions in $\{T_2^1, T_2^3, T_2^6, T_2^8\}$ mutually conflict, they can only be scheduled to execute sequentially. A similar schedule applies for $\{T_2^2, T_2^4, T_2^5, T_2^7\}$. The resulting $G_c^*(\phi_{\kappa_1})$ contains 6 independent sets. The dashed arrows represent zero-weighted arcs (i.e., (T_1^i, T_2^i) for each node v_i).

Given the initial input of G_c, algorithm ϕ_{κ_2} selects a 4-coloring of G_c, which is not temporarily optimal: $\{T_1^1, T_1^7\}$, $\{T_1^2, T_1^8\}$, $\{T_1^3, T_1^5\}$ and $\{T_1^4, T_1^6\}$ are selected to execute sequentially. When $\{T_1^1, T_1^7\}$ commits, the second transactions issued by v_1 and v_7 can execute immediately in parallel with all other transactions. A similar case applies for $\{T_1^2, T_1^8\}$, $\{T_1^3, T_1^5\}$ and $\{T_1^4, T_1^6\}$. Therefore, $G_c^*(\phi_{\kappa_2})$ only contains 5 independent sets.

To compute the makespan, we need to find the longest weighted paths in $G_c^*(\phi_{\kappa_1})$ and $G_c^*(\phi_{\kappa_2})$. Since the longest path in $G_c^*(\phi_{\kappa_1})$ may contain at most six transactions, and the longest path in $G_c^*(\phi_{\kappa_2})$ may contain at most five transactions, it is likely that $\text{makespan}_{(\kappa_1, \phi_{\kappa_1})} > \text{makespan}_{(\kappa_2, \phi_{\kappa_2})}$, despite the fact that ϕ_{κ_1} always selects a $\chi(G_c)$-coloring of G_c.

The inherent reason that the $\chi(G_c)$-coloring of G_c does not always lead to an optimal schedule is due to the dynamic nature of G_c. If an algorithm selects a set of transactions to execute based on the $\chi(G_c)$-coloring of G_c at time t, the chromatic number of the updated G_c after t (when committed transactions leave G_c and new transactions join G_c) may not always be optimal. For Example 1, the initial G_c is 2-colorable. However, after the execution of $\{T_2^1, T_2^3, T_2^6, T_2^8\}$ by ϕ_{κ_1}, the chromatic number of the updated G_c is 5. On the other hand, ϕ_{κ_2} always keeps the chromatic number of the updated G_c to be lower than 3. Therefore, for

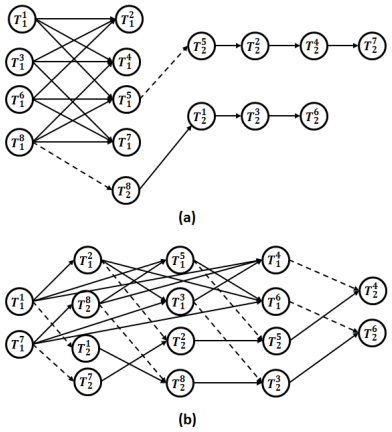


Fig. 3. Example 2: dynamic ordering conflict graphs $G_c^*(\phi_{\kappa_1})$ and $G_c^*(\phi_{\kappa_2})$.

the dynamic conflict graph, local optimal selection does not imply the globally optimality.

A.3 Pseudo-code Algorithm Cutting

Algorithm 1: Algorithm CUTTING

Input: A set of transactions $\mathcal{T}$; the conflict graph G_c; the cost graph G_d with diameter D; each transaction has the same local execution time duration τ; the approximate TSP algorithm A; a (C, A) partitioning $\mathcal{P}(C, A, v)$ on G_d; each transaction T_i knows $\boldsymbol{o}_i$ after it starts;
Output: An execution schedule of $\mathcal{T}$.

procedure Abort (T_1, T_2)
Abort transaction T_2;
T_2 waits until T_1 commits or aborts;
end procedure

On (re)start of transaction T_1;
$\pi_1 \leftarrow$ random integer in $[1, m]$;
On conflict of transaction T_1 invoked at node $v^{j_1} \in P_{t_1}$ with transaction T_2 invoked at node $v^{j_2} \in P_{t_2}$;

if $t_1 = t_2$ *&* $\exists$ *integer* $\nu \geq 1$ *s.t.* $\lfloor \frac{\max\{\psi(v^{j_1}), \psi(v^{j_2})\}}{2^\nu} \rfloor = \min\{\psi(v^{j_1}), \psi(v^{j_2})\}$ **then**
    `// one transaction is an ancestor of the other in` BT(P_t)
    if $j_1 < j_2$ **then Abort** (T_1, T_2); **else Abort** (T_2, T_1);
else
    if $\pi_1 < \pi_2$ **then Abort** (T_1, T_2); **else Abort** (T_2, T_1);