Multi-scheduler Concurrency Control for Parallel Database Systems

Sushant Goel[1], Hema Sharda[1], and David Taniar[2]

[1] School of Electrical and Computer systems Engineering,
Royal Melbourne Institute of Technology, Australia
s2013070@student.rmit.edu.au
hema.sharda@rmit.edu.au
[2] School of Business Systems,
Monash University, Australia
David.Taniar@infotech.monash.edu.au

Abstract. Increase in amount of data stored and requirement of fast response time has motivated the research in *Parallel Database Systems* (PDS). Requirement for correctness of data still remains one of the major issues. Concurrency control algorithms used by PDS uses *single scheduler* approach. Single scheduler approach has some inherent weaknesses such as – very big lock tables, overloaded centralized scheduler and more number of messages in the system. In this paper we investigate the possibility of *multiple schedulers* and conclude that single scheduler algorithms cannot be migrated in the present form to multi-scheduler environment. Next, we propose a *Multi-Scheduler Concurrency Control* algorithm for PDS that distributes the scheduling responsibilities to the respective *Processing Elements*. Correctness of the proposed algorithm is then discussed using a different serializability criterion – *Parallel Database Quasi-Serializability*.

1 Introduction

Research and development efforts have made Parallel Database Systems (PDS) a reality during the last decade. PDS are specifically used to meet requirements of Very Large Databases (VLDB) – volume of data spanning to terabyte (10^{12}) size. Any database system must guarantee the consistency of the database.

Researchers in the recent past have concentrated on query optimization [12,13] in parallel database environment. Very little work has been done in the area of transaction processing and concurrency control to meet the specific needs of PDS. Single scheduler approach, and thus serializability [9], has been used as the correctness criterion for concurrency control. High performance and multiprocessor database systems such as *Parallel Database Systems* (PDS) [2,15], Distributed Database Systems (DDS) [15,16], Multidatabase Systems (MDS) [14] have made the notion of *conflict serializability* [9] insufficient as a correctness criterion.

This paper proposes a new serializability criterion, *Parallel Database Quasi-serializability* (*PDQ-serializability*), and then formulates a Multi-scheduler Concurrency Control algorithm to meet the requirements of PDS. We focus on the problem of concurrent transaction execution in parallel database systems. Serializability as a correctness criterion assumes the existence of single scheduler to ensure the database consistency. Single scheduler strategy may not meet the requirements of PDS. Also, Single scheduler strategy may not be directly migrated to multi-scheduler environment, the underlying problem is discussed in detail in the next section. The single scheduler scheme has some inherent weaknesses, such as: 1) Due to centralized scheduling, the lock table grows very big at the scheduler. 2) Abundant computing power of multiple processors is not utilized to the fullest. 3) All sites have to communicate to a central scheduling node thus increasing the number of messages in the system. 4) Sometimes it is difficult to decide the coordinator-node.

The proposed multi-scheduler concurrency control approach minimizes the above-mentioned weaknesses for obvious reasons; we would not discuss them in detail as they are out of the scope of the paper. The purpose of this research is to explore the issues in concurrency control of parallel database systems that uses multiple schedulers. The paper covers two major aspects: 1) We show that the concurrency control algorithms implemented in single scheduler environment may produce incorrect results in multi-scheduler environment. We use multiple schedulers to distribute the load, contrary to single scheduler environment. 2) Next, we propose a Multi-scheduler Concurrency Control algorithm that distributes the scheduling responsibilities to the respective *Processing Elements (PE)* and guarantees a PDQ-serializable schedule. We must emphasize that this concept is similar yet different than multidatabase serializability (MDBS) or heterogeneous distributed database serializability (HDDBS). MDBS and HDDBS consider two levels of transactions – local and global, and treat them differently, but the proposed algorithm treats all the transactions equally, for detailed description and comparison refer [2, 14, 15, 16].

Rest of the paper is organized as follows: Section 2 gives the fundamental definitions of database terms and PDS, Section 3 discusses the motivation of this work, Section 4 presents the formal model of PDS that we consider for the algorithm. Section 5 elaborates the proposed algorithm and proves the correctness of the algorithm. Finally, Section 6 concludes the paper and discusses the future extension of the work.

2 Background

In this section we first give the fundamental definitions of database terminologies that we use through out the paper. Next we briefly discuss most common architecture of the PDS.

2.1 Fundamental Definitions

We would like to briefly define the *transaction* and properties of transactions before we proceed with the proposed algorithm. From the user's viewpoint, a transaction is the execution of operations that accesses shared data in the database, formally [9]:

Definition 1: A transaction T_i is a set of read (r_i), write (w_i), abort (a_i) and commit (c_i). T_i is a partial order with ordering relation $\prec_i$ where:

1) $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item}\} \bigcup \{a_i, c_i\}$
2) $a_i \in T_i$ iff $c_i \notin T_i$
3) If t is a_i or c_i, for any other operation $p \in T_i, p \prec_i t$
4) If $r_i[x], w_i[x] \in T_i$, then either $r_i[x] \prec_i w_i[x]$ or $w_i[x] \prec_i ri[x]$. □

Definition 2: A complete history (or schedule) H over $\boldsymbol{T}$ (let, $\boldsymbol{T}$ be set of transactions, $\boldsymbol{T} = \{T_1, T_2, \ldots T_n\}$) is a partial order with ordering relation $\prec_H$ where [9]:

1) $H = \bigcup_{i=1}^{n} T_i$;
2) $\prec_H \supseteq \bigcup_{i=1}^{n} \prec_i$; and
3) For any two conflicting operations $p, q \in H$, either $p \prec_H q$ or $q \prec_H p$. □

Definition 3: A history H is *Serializable (SR)* if its committed projection, *C(H)*, is equivalent to a serial execution H_s [9]. □

Definition 4: A database history H_s is serial iff [9]
($\exists p \in T_i, \exists q \in T_j$ such that $p \prec_{\text{Hs}} q$) then ($\forall r \in T_i, \forall s \in T_j, r \prec_{\text{Hs}} s$).

□

A history (H) is serializable iff serialization graph is acyclic [3, 9].

2.2 Parallel Database Systems

The enormous amount and complexity of data and knowledge to be processed by the systems imposes the need for increased performance from the database system. The problems associated with the large volumes of data are mainly due to: 1) Sequential data processing and 2) The inevitable input/output bottleneck.

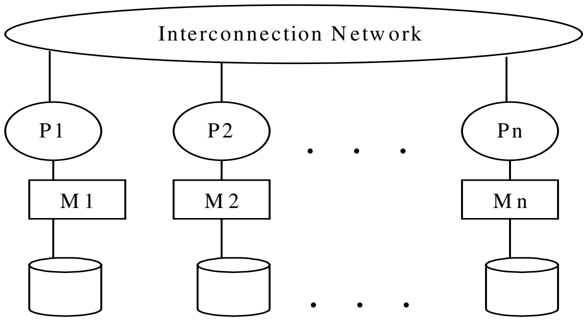

Fig. 1. Shared Nothing PDS

Parallel database systems have emerged in order to avoid these bottlenecks. Different strategies are used for data and memory management in multiprocessor environment to meet specific requirements. In s*hared memory architecture* processors have direct access to the disks and have a global memory. This architecture is good for load balancing. In s*hared disk architecture* processors have direct access to all disks but have private memory. *Shared-nothing architecture* has individual memory and disk for each processor, called *processing element (PE)*. The main advantage of shared-nothing multi-processors is that they can be scaled up to hundreds and probably thousands of processors, for detailed discussion and comparison see [1, 2, 5, 7, 8]. We assume shared nothing architecture for this study.

3 Motivating Example

The following example shows that single scheduler algorithms may not be directly migrated to meet the requirements of multi-scheduler environment. We consider two PE and four data objects (O_1, O_2, O_3, O_4) to demonstrate the motivation of our work. The two PE's are denoted as P_1 and P_2. Say, the data objects are located as follows:

$P_1 = O_1, O_2$

$P_2 = O_3, O_4$

Now, consider two transactions are submitted to the database as shown below.

$T_1 = r_1(O_1)\ r_1(O_2)\ w_1(O_3)\ w_1(O_1)\ C_1$

$T2 = r_2(O_1)\ r_2(O_3)\ w_2(O_4)\ w_2(O_1)\ C_2$

Each transaction is divided into subtransactions according to the location of data objects. From T_1 and T_2 following subtransactions are obtained:

$T_{11} = r_{11}(O_1)\ r_{11}(O_2)\ w_{11}(O_1)\ C_{11}$

$T_{12} = w_{12}(O_3)\ C_{12}$

$T_{21} = r_{21}(O_1)\ w_{21}(O_1)\ C_{21}$

$T_{22} = r_{22}(O_3)\ w_{22}(O_4)\ C_{22}$

T_{11} and T_{12} are read as subtransaction of transaction 1 at PE_1 and subtransaction of transaction 1 at PE_2 respectively. Assume, the following histories are produced by the local scheduler at the processing elements:

$H_1 = r_{11}(O_1)\ r_{11}(O_2)\ w_{11}(O_1)C_{11}\ r_{21}(O_1)\ w_{21}(O_1)\ C_{21}$

$H_2 = r_{22}(O_3)\ w_{22}(O_4)\ C_{22}\ w_{12}(O_3)\ C_{12}$

H_1 and H_2 are history at PE_1 and PE_2 respectively. Both the local scheduler produces serializable history with following serialization order:

Scheduler 1: schedules transaction 1 $\prec$ transaction 2

Scheduler 2: schedules transaction 2 $\prec$ transaction 1

The two serialization orders are contradictory. Although individual schedulers generate serial history the combined effect produces a cycle $T_1 \rightarrow T_2 \rightarrow T_1$. Thus, the responsibility of scheduler at each PE is much more than that of a single scheduler. Hence, we conclude that the concurrency control strategies applicable to single scheduler may not be migrated in its present form to multi-scheduler environment.

In the literature, we could not find any algorithm that considers multiple schedulers for PDS. Weaknesses of single scheduler algorithms motivated us to distribute the scheduling responsibilities to the respective PE and consider multiple schedulers. We propose a new serializability, *Parallel Database Quasi-serializability*, criterion to meet requirements of multi-scheduler concurrency control algorithms in the following sections.

4 Model of Parallel Database System

Shared-nothing parallel database system is considered in this study. For the sake of simplicity we consider only two processing elements to demonstrate our algorithm.

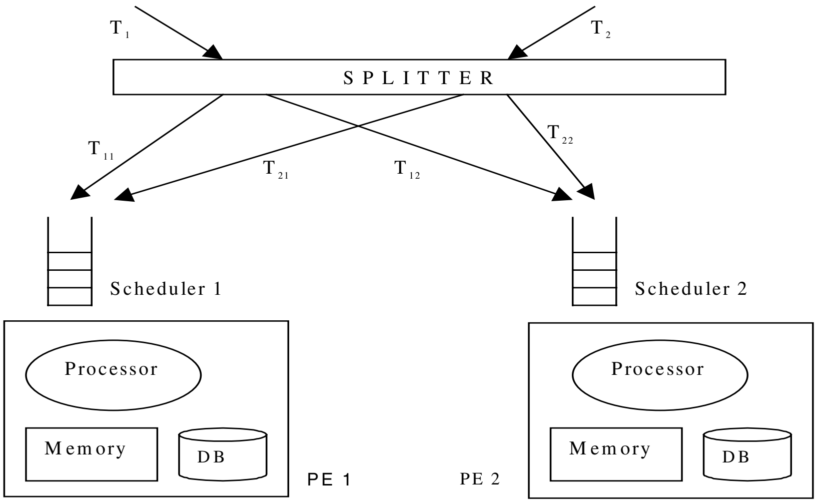

Fig. 2. Conceptual model of multi-scheduler concurrency control

We assume that local schedulers are capable of producing serializable schedule. Different parts of the model is described as follows:

a) *Splitter*: *Splitter* contains the information regarding the data allocated to the processing elements and splits the transaction into multiple subtransactions depending on data location. In the literature *splitter* had the responsibility of central scheduler. We will refer to scheduler as shown in figure-2.
b) *Processing Elements*: PE contains a processor, a memory and a fragment of database as discussed in the shared nothing architecture (section 2).
c) *Schedulers at each PE*: Schedulers are responsible for generating a serializable schedule. The responsibility of each scheduler is much more than in a single scheduler environment as discussed in the previous section.
d) *Transactions*: Transactions are exactly same as that of single processor environment and follow the properties of *Definition 1*.

5 Proposed Serializability Theorem and Algorithm

In this section we first discuss the correctness criterion for the Multi-scheduler Concurrency Control algorithm namely *Parallel Database Quasi-serializability*

(PDQ-serializability) subsection 5.2 proposes a PDQ-serializability theorem, subsection 5.3 explains the proposed *Timestamp based Multi-scheduler Concurrency Control* algorithm and finally subsection 5.4 proves the correctness of the algorithm.

5.1 PDQ-Serializability

Scheduling responsibilities in PDS is distributed to respective PE according to the partitioned data. The motivating example shows that although individual schedulers produce serial histories the database is not in a consistent state. Some additional criterion has to be enforced to ensure multi-scheduler concurrency control in addition to subtransaction level granularity. We propose a new serializability criterion that separates two types of transactions – transactions having only one subtransaction and transactions having more than one subtransaction. The following definition states the correctness criterion for PDS.

Definition 5: A *Multi-Scheduler Serial (MS-Serial)* history is considered correct in parallel database system. A history is *MS-Serial* iff:

1. Every PE produces serializable history and
2. Any transaction having more than one subtransaction i.e. accessing more than one PE, executes the transaction according to *total order*. Total order for two transactions T_i and T_j can be defined as – if T_i precedes T_j at any PE then all of T_i*'s* operations precede T_j*'s* operations in all PE in which they both appear.□

The second condition is similar to *quasi-serializability*. Hence, we name the correctness criterion as *Parallel Database Quasi-Serializability.*

Definition 6: A history in multi-scheduler environment is PDQ-serializable iff it is equivalent to a *MS-Serial* history (definition of equivalence ($\equiv$) from [9], page 30). □

5.2 Proposed PDQ-Serializability Theorem

The PDQ-serializability of the history is determined by analysing *Parallel Database Quasi-Serializability* (PDQ-serializabilty) graphs. Only the committed projection [9] of the history *C(H)* is considered in the definition. The following definition describes the PDQ-serializability graph.

Definition 7: At any given instance, histories of all the schedulers at each PE can be represented using a directed graph defined with the ordered three: ($\mathbf{T}^1$, $\mathbf{T}^n$, $\boldsymbol{A}$). The graph would be referred as *Multi-scheduler Serializability Graph (MSG).*

1. $\mathbf{T}^1$: set of labelled vertices representing transactions with one *subtransaction.*
2. $\mathbf{T}^n$: set of labelled vertices representing transactions with more than one *subtransaction.*
3. $\boldsymbol{A}$ is the set of arcs representing the ordering of transaction in each PE. □

In rest of the paper we would only consider transaction having more than one subtransaction and denote that as *T*. Transactions with single subtransaction are taken care by the individual PE and do not pose any threat to the concerned problem. Based on the definition of MSG we next formalize the following theorem.

Theorem 8: A history in multi-scheduler environment is PDQ-Serializable iff MSG is acyclic.

Proof: (if) Let us assume without loss of generality that the committed history *C(H)* consists of the set $\{T_1, T_2, ..., T_n\}$. $T_1, T_2, ..., T_n$ are transactions with more than one subtransaction. We assume that all processing elements always schedule transactions in serializable order. The *n* vertices of the MSG ($\{T_1, T_2, ..., T_n\}$) are acyclic and thus it may be topologically sorted (definition of topologically sorted from [9]). Let $T_{i1}, T_{i2}, \ldots, T_{in}$ be a topological sort of the multi-scheduler history for permutation of 1, 2, …, n. Let the MS-serial history be $T_{i1}, T_{i2}, \ldots, T_{in}$. We show that: $C(H) \equiv$ *MS-serial*. Let $p \in T_i$ and $q \in T_j$ and *p* and *q* conflict such that $p \prec_H q$. This means an arc exists in the MSG from T_i to T_j. Therefore in any topological sort of multi-scheduler history T_i precedes T_j. Thus, all operations of T_i precede all operations of T_j in any topological sort. Thus $C(H) \equiv$ *MS-serial* and from *Definition-6* the history *H* is PDQ-serializable.

(only if): Suppose a history *H* is PDQ-serializable and let H_s be a MS-serial history equivalent to *H*. Consider an arc exists in the MSG, T_i to T_j. This implies there exists two conflicting operations $p \in T_i$ and $q \in T_j$ such that $p \prec_H q$ at some PE_i. This condition is valid as both operation conflicts at the same PE_i. Since, $H_s \equiv H$, all operations of T_i occur before T_j at that specific PE. Suppose there is a cycle in MSG. This implies at any other PE_j, $T_j \prec T_i$ in MS-serial history. But T_i is known to precede T_j at PE_i, which is contradictory to the earlier assumption. □

5.3 Timestamp Based Multi-scheduler Concurrency Control Algorithm

In this section we propose a *Timestamp based Multi-scheduler Concurrency Control* (TMCC) algorithm that enforce *total order* in the schedule to ensure PDQ-serializability. *Total order* is required only for those conflicting transactions that accesses more than one PE being accessed by other active transactions.

Following functions are used to demonstrate the algorithm:

1. *Split_trans(T_i)*: This function takes the transaction submitted to the *splitter* as parameter and returns a set of subtransactions that access different PEs.
2. *PE_accessed(T_i)*: This function also takes the transaction as parameter and returns the set of PEs where the subtransactions for T_i are executed.
3. *Active_trans(PE)*: This function takes the processing element as parameter and returns the set of transactions that have an active subtransaction executing at that PE.
4. *Cardinality()*: This function can take any set as parameter and returns the number of elements in the set.
5. *Append_TS(Subtransaction):* This function takes a *subtransaction* of a transaction T_i and appends a *timestamp* to the subtransaction. All the subtransaction of a transaction will have the same timestamp value.

Algorithm 1: TMCC Algorithm (Transaction submission)

```
begin
   input $T_i$ : Transaction
   var Active_trans : set of active transactions
   generate timestamp $ts$ : unique timestamp in increasing order is generated

   $Split\_trans(T_i)$
   $PE\_accessed(T_i) \longleftarrow$ set of Processing Elements accessed
(1)  if $Cardinality(PE\_accessed(T_i)) = 1$ then
         begin
(2)        $Active\_Trans(PE_k) \longleftarrow Active\_Trans(PE_k) \bigcup T_i$
(3)        submit subtransaction to PE
         end
   else begin
(4)     Active_trans $\longleftarrow \bigcup Active\_Trans(PE_k)$
(5)     Active_trans $\longleftarrow \{T \mid T \in$ Active_trans $\wedge Cardinality(PE\_accessed(T))>1\}$
(6)     for each subtransaction of $T_i$
(7)        $Append\_TS(Subtransaction)$
        end for
(8)     if $Cardinality(PE\_accessed(T_i) \bigcap (\bigcup_{T \in Active\_trans} PE\_accessed(T_j)) \leq 1)$
          then begin
(9)       for each $PE_k \in PE\_accessed(T_i)$
            begin
(10)          $Active\_Trans(PE_k) \longleftarrow Active\_Trans(PE_k) \bigcup T_i$
(11)          submit subtransaction to $PE_k$   ::Subtransaction executes immediately.
            end
          end for
        end
        else
(12)    for each $PE_k \in PE\_accessed(T_i)$
         begin
(13)       $Active\_Trans(PE_k) \longleftarrow Active\_Trans(PE_k) \bigcup T_i$
(14)    submit subtransaction to PE's Queue   ::Subtransaction submitted to queue
         end
        end for
      end if
    end
  end if
end
```

Working of the algorithm is explained below:

1. As soon as the transaction arrives at the splitter, *split_trans(T_i)* splits the transaction into multiple subtransactions according to the allocation of data.

2. If there is only one subtransaction required by the transaction, the transaction can be submitted to the PE immediately without any delay. (**Line 1**).
3. All the transactions having more than one subtransaction are added to the **Active_Trans** set (**Line 4** and **Line 5**).
4. *Splitter* appends a timestamp with every subtransaction for those transactions that require multiple subtransactions, before submitting to the PE (**Line 6**).
5. If there are active transactions that access one or less than one PEs accessed by the transaction being scheduled then the subtransactions can be scheduled immediately (**Line 8**).
6. If there is active transactions that access more than one of the PEs accessed by the transaction being scheduled then the subtransactions are submitted to the PE's *wait_queue*. The subtransactions from the queue are executed strictly according to the timestamps (**Line 12**).
7. Subtransactions from *step-2* can be assumed to have lowest timestamp value e.g. 0 and can be scheduled immediately.
8. When all subtransactions of any transaction complete the execution at all the sites, the transaction commits and is removed from *Active_trans(PE)* list.

Algorithm 2 explains the steps when any subtransaction terminates at any particular processing element.

Algorithm 2: TMCC Algorithm (Transaction termination)

```
begin
  input subtransaction of $T_i$ completes execution
  $Active\_Trans(PE_k) \longleftarrow Active\_Trans(PE_k) - T_i$ :: removes transaction from the PE
  $PE\_accessed(T_i) \longleftarrow PE\_accessed(T_i) - PE_k$
        :: removes the PE being accessed from the PE_accessed set
end
```

5.4 Correctness of TMCC Algorithm

The model assumes that each PE is capable of producing serializable schedule (Proposition-1). At the same time motivating example demonstrates the necessity of additional strictness criterion to guarantee the serialization of parallel database systems in multi-scheduler environment.

Proposition 9: All processing elements always schedule all transactions in serializable order. □

To prove the correctness of the proposed algorithm we show that the additional criterion enforced by the *splitter* will guarantee serializable schedule in parallel database systems. The *splitter* does not control the execution of schedules but only determines the way subtransactions are submitted to the PE. If any transaction accesses more than one database being accessed by other active transactions, the subtransactions cannot be scheduled immediately and thus have to be submitted to *wait_queue* (Proposition-2). Each processing element's *queue* schedules the subtransactions according to their timestamp.

Proposition 10: *Splitter* submits the subtransactions to the *wait_queue* of PE if active transactions access more than one common database. □

This ensures that the conflicting transactions are executed according to the timestamp value to ensure the PDQ-serializable execution of the multi-scheduler system. The following Lemma proves this.

Lemma 11: For any two transactions T_i, T_j scheduled by TMCC algorithm, either all of T_i*'s* subtransactions are executed before T_j at every PE or vice versa.

Proof: There are three cases to be considered.

Case 1) *A transaction T_i requires only single subtransaction (ST_i):* This situation is shown in **line (1)** of the algorithm. The subtransaction is submitted immediately as shown in the algorithm flow chart. From Poposition-1 it follows that any other subtransaction $ST_j \in T_j$ either precedes or follows ST_i.

Case 2) *A transaction T_i splits into multiple subtransactions but accesses only one PE accessed by other active transaction:* This situation is shown in **line (8)** of the algorithm. **Line (8)** checks for the above mentioned condition and the subtransaction is submitted to the PE immediately with a timestamp. The timestamp will be used for case-3. Consider two active transactions that overlap only at one PE. Since they overlap at only one PE, Proposition-1 ensures that the transactions would be ordered in a serializable way.

Case 3) *A transaction T_i splits into multiple subtransactions and accesses more than one PE accessed by other active transaction:* This situation is shown in **line (12)** of the algorithm. Under this condition schedulers at PEs may schedule the transactions in conflicting mode. To avoid this conflict we submit the transactions in the PEs *wait_queue* instead of scheduling it immediately (Proposition-2). The transactions in the *wait_queue* are executed strictly according to the timestamp order.

Say, transaction T_i has two subtransactions T_{i1} and T_{i2} already executing at PE_1 and PE_2. When T_j arrives and it also has T_{j1} and T_{j2}. Then **if** condition at **line (8)** fails and the subtransactions are submitted to the *wait_queue* at each PE. Assume that the timestamp of T_i, $TS(T_i) \prec TS(T_j)$. Timestamp is appended to the subtransactions of T_i and T_j during the execution of **line (6)**. Then T_i will precede T_j at both the sites because the transactions are scheduled strictly according to the timestamp value, thus avoids execution of incorrect schedules. □

Theorem 12: The Timestamp based Multi-scheduler Concurrency Control algorithm presented produces PDQ-serializable histories.

Proof: All PDQ-serializable histories can be shown by acyclicity of multi-scheduler serializability graph, as demonstrated by Theorem 1. Thus, we will show that the proposed TMCC algorithm avoids cycle in the MSG.

Without loss of generality, consider that there is a sequence of arcs from $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$. This implies that there exists an operation of subtransaction

$T_{1k} \in T_1$ that precedes and conflicts with an operation of $T_{nk} \in T_n$ at any particular, say PE_k. For a cycle, another sequence of arcs must exist from $T_n \rightarrow T_{n-1} \rightarrow \ldots \rightarrow T_1$. Two possibilities can exist due to this sequence. (1) An operation of $T_{nk} \in T_n$ precedes and conflicts with an operation of $T_{1k} \in T_1$ at the same PE_k. This contradicts Proposition 1. (2) An operation of some other subtransaction $T_{nl} \in T_n$ at another PE_l precedes and conflicts with an operation of $T_{1l} \in T_1$. This contradicts Lemma 1. □

6 Conclusion

Due to the inherent weaknesses of single schedulers (discussed in the paper) multi-scheduler approach has to be investigated. Our investigation shows that algorithms developed for single scheduler cannot guarantee correct schedules in multi-scheduler environment. The notion of conflict serializability is insufficient of producing correct schedule and a new serializability criterion is required for multi-scheduler approach. We discuss the PDQ-serializability and propose an algorithm that guarantees PDQ-serializable schedules. The proposed algorithm distributes the scheduling responsibilities to the PEs, and reduces the overheads of single scheduler strategy.

References

[1] A. Bhide, "An Analysis of Three Transaction Processing Architectures", *Proceedings of 14th VLDB Conference*, pp. 339–350, 1988.

[2] D.J. DeWitt, J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Communication of the ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[3] J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[4] L. Sha, R. Rajkumar, J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-time Synchronization", *IEEE Transactions*, vol–39, No.9, pp. 1175– 1185, 1990.

[5] M. Stonebraker, "The Case for Shared-Nothing", *IEEE Data Engineering*, vol. 9, no. 1, pp. 4–9, 1986.

[6] M.-A. Neimat, D.A.Schneider, "Achieving Transactional Scaleup on Unix", *Parallel and Distributed Information Systems, Proceedings of the 3rd Intl. Conf. on*, pp. 249–252, 1994.

[7] P. Valduriez, **"**Parallel Database Systems: The Case For Shared Something", *Proceedings of the International Conference on Data Engineering*, pp. 460–465, 1993.

[8] P. Valduriez, "Parallel Database Systems: Open Problems and New Issues", *Distributed and Parallel Databases*, volume 1, pp. 137–165, 1993

[9] P. A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addision-Wesley, 1987.

[10] S. S. Thakkar, M. Sweiger, "Performance of an OLTP Application on Symmetry Multiprocessor System", *Proceeding, 17th Intl. Conf. on Comp. Arch.*, pp. 228–238, 1990.

[11] T.-W. kuo, J. Wu, H.-C. Hsih, "Real-time concurrency control in multiprocessor environment",*IEEE Transaction, Parallel and Dist. Sys.*, vol.–13, no.–6, pp. 659–671, '02.

[12] S.D.Chen; H. Shen; R. Topor, "Permutation-based range-join algorithms on N-dimensional meshes", *Parallel and Dist. Sys., IEEE Trans,Vol–13*, No–4, p: 413–431, '02.

[13] C.-M. Chen, R. K Sinha, “Analysis and comparison of declustering schemes for interactive navigation queries”, *Knowledge and Data Engineering, IEEE Transactions on, Vol–12 Issue: 5 ,*pp: 763–778, 2000.
[14] K. Barker, “Transaction Management on Multidatabase Systems”, PhD thesis, Department of Computer Science, The university of Alberta, Canada, 1990.
[15] T,Ozsu, P.Valduriez, "*Distributed and Parallel Database Systems*", *ACM Computing Surveys, vol.28, no.1*, pp 125–128, March 1996.
[16] M.T. Ozsu and P. Valduriez, editors. *Principles of Distributed Database Systems* (Second Edition). Prentice-Hall, 1999.