

On the performances in simulation of parallel databases: an overview on the most recent techniques for query optimization

A.B.M.Rubaiyat Islam Sadat

Department of Information Engineering and
Computer Science

University of Trento

Trento, Italy

Email: sadat@science.unitn.it

Paola Lecca

Center for Computational and Systems Biology

The Microsoft Research-University Of Trento

Trento, Italy

Email: lecca@cosbi.eu

Outline of the presentation

- Introduction and background
- Parallel database architectures
- Query optimization
- Partitioning techniques
- Skewing effect
- Parallel join techniques
- Comparative studies with experimental results
- Discussion

Introduction

- Parallel machines are becoming quite common and affordable
 - Prices of microprocessors, memory and disks have dropped sharply
 - Recent desktop computers feature multiple processors and this trend is projected to accelerate
- Databases are growing increasingly large
 - large volumes of transaction data are collected and stored for later analysis.
 - multimedia objects like images are increasingly stored in databases
- Large-scale parallel database systems increasingly used for:
 - storing large volumes of data
 - processing time-consuming decision-support queries
 - providing high throughput for transaction processing

Parallelism in Databases

- Data can be partitioned across multiple disks for parallel I/O.
- Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
 - data can be partitioned and each processor can work independently on its own partition.
- Queries are expressed in high level language
 - makes parallelization easier.
- Different queries can be run in parallel with each other.
 - Concurrency control takes care of conflicts.
- Thus, databases naturally lend themselves to parallelism.

Background

- Skew is more important for uneven distribution of data in parallel databases
- Due to several insert and delete operations, the distribution of data is ruffled
- Unbalanced data lead to larger processing time
- Load balancing is important for the effective query processing
- A technique using parallel join with hash partitioning

Parallel database architectures

- Parallel database range between two extremes
 - Shared nothing: each node is independent and self-sufficient, and there is no single point of contention across the system
 - Shared memory: One program will create a memory portion which other processes (if permitted) can access
- A useful intermediate is Shared-disk architecture
- Our experiments were based on shared-nothing architecture

Interquery Parallelism

- Queries/transactions execute in parallel with one another.
- Increases transaction throughput; used primarily to scale up a transaction processing system to support a larger number of transactions per second.
- Easiest form of parallelism to support, particularly in a shared-memory parallel database, because even sequential database systems support concurrent processing.
- More complicated to implement on shared-disk or shared-nothing architectures
 - Locking and logging must be coordinated by passing messages between processors.
 - Data in a local buffer may have been updated at another processor.
 - **Cache-coherency** has to be maintained — reads and writes of data in buffer must find latest version of data.

Intraquery Parallelism

- Execution of a single query in parallel on multiple processors/disks; important for speeding up long-running queries.
- Two complementary forms of intraquery parallelism :
 - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query.
 - **Interoperation Parallelism** – execute the different operations in a query expression in parallel.

the first form scales better with increasing parallelism because the number of tuples processed by each operation is typically more than the number of operations in a query

Query Optimization

- Query optimization in parallel databases is significantly more complex than query optimization in sequential databases.
- Cost models are more complicated, since we must take into account partitioning costs and issues such as skew and resource contention.
- When **scheduling** execution tree in parallel system, must decide:
 - How to parallelize each operation and how many processors to use for it.
 - What operations to pipeline, what operations to execute independently in parallel, and what operations to execute sequentially, one after the other.
- Determining the amount of resources to allocate for each operation is a problem.
 - E.g., allocating more processors than optimal can result in high communication overhead.
- Long pipelines should be avoided as the final operation may wait a lot for inputs, while holding precious resources

Partitioning techniques

- Reduce the time required to retrieve relations from disk by partitioning
- the relations on multiple disks.
- Horizontal partitioning – tuples of a relation are divided among many disks such that each tuple resides on one disk.
- Partitioning techniques (number of disks = n):

Round-robin:

Send the i^{th} tuple inserted in the relation to disk $i \bmod n$.

Hash partitioning:

- Choose one or more attributes as the partitioning attributes.
- Choose hash function h with range $0 \dots n - 1$
- Let i denote result of hash function h applied to the partitioning attribute value of a tuple. Send tuple to disk i .

Partitioning techniques (Cont.)

■ Range partitioning:

- Choose an attribute as the partitioning attribute.
- A partitioning vector $[v_0, v_1, \dots, v_{n-2}]$ is chosen.
- Let v be the partitioning attribute value of a tuple. Tuples such that $v_i \leq v_{i+1}$ go to disk $i + 1$. Tuples with $v < v_0$ go to disk 0 and tuples with $v \geq v_{n-2}$ go to disk $n-1$.

E.g., with a partitioning vector $[5, 11]$, a tuple with partitioning attribute value of 2 will go to disk 0, a tuple with value 8 will go to disk 1, while a tuple with value 20 will go to disk 2.

Comparison of Partitioning Techniques

- Evaluate how well partitioning techniques support the following types of data access:
 1. Scanning the entire relation.
 2. Locating a tuple associatively – **point queries**.
 - E.g., $r.A = 25$.
 3. Locating all tuples such that the value of a given attribute lies within a specified range – **range queries**.
 - E.g., $10 \leq r.A < 25$.

Comparison of Partitioning Techniques (Cont.)

Round robin:

- Advantages

- Best suited for sequential scan of entire relation on each query.
- All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.

- Range queries are difficult to process

- No clustering -- tuples are scattered across all disks

Comparison of Partitioning Techniques(Cont.)

Hash partitioning:

- Good for sequential access
 - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
 - Retrieval work is then well balanced between disks.
- Good for point queries on partitioning attribute
 - Can lookup single disk, leaving others available for answering other queries.
 - Index on partitioning attribute can be local to disk, making lookup and update more efficient
- No clustering, so difficult to answer range queries

Comparison of Partitioning Techniques (Cont.)

- Range partitioning:
 - Provides data clustering by partitioning attribute value.
 - Good for sequential access
 - Good for point queries on partitioning attribute: only one disk needs to be accessed.
 - For range queries on partitioning attribute, one to a few disks may need to be accessed
 - ▶ Remaining disks are available for other queries.
 - ▶ Good if result tuples are from one to a few blocks.
 - ▶ If many blocks are to be fetched, they are still fetched from one to a few disks, and potential parallelism in disk access is wasted

Parallel Processing of Relational Operations

- Our discussion of parallel algorithm assumes:
 - *read-only* queries
 - shared-nothing architecture
 - n processors, P_0, \dots, P_{n-1} , and n disks D_0, \dots, D_{n-1} , where disk D_i is associated with processor P_i .
- If a processor has multiple disks they can simply simulate a single disk D_i .
- Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems.
 - Algorithms for shared-nothing systems can thus be run on shared-memory and shared-disk systems.
 - However, some optimizations may be possible.

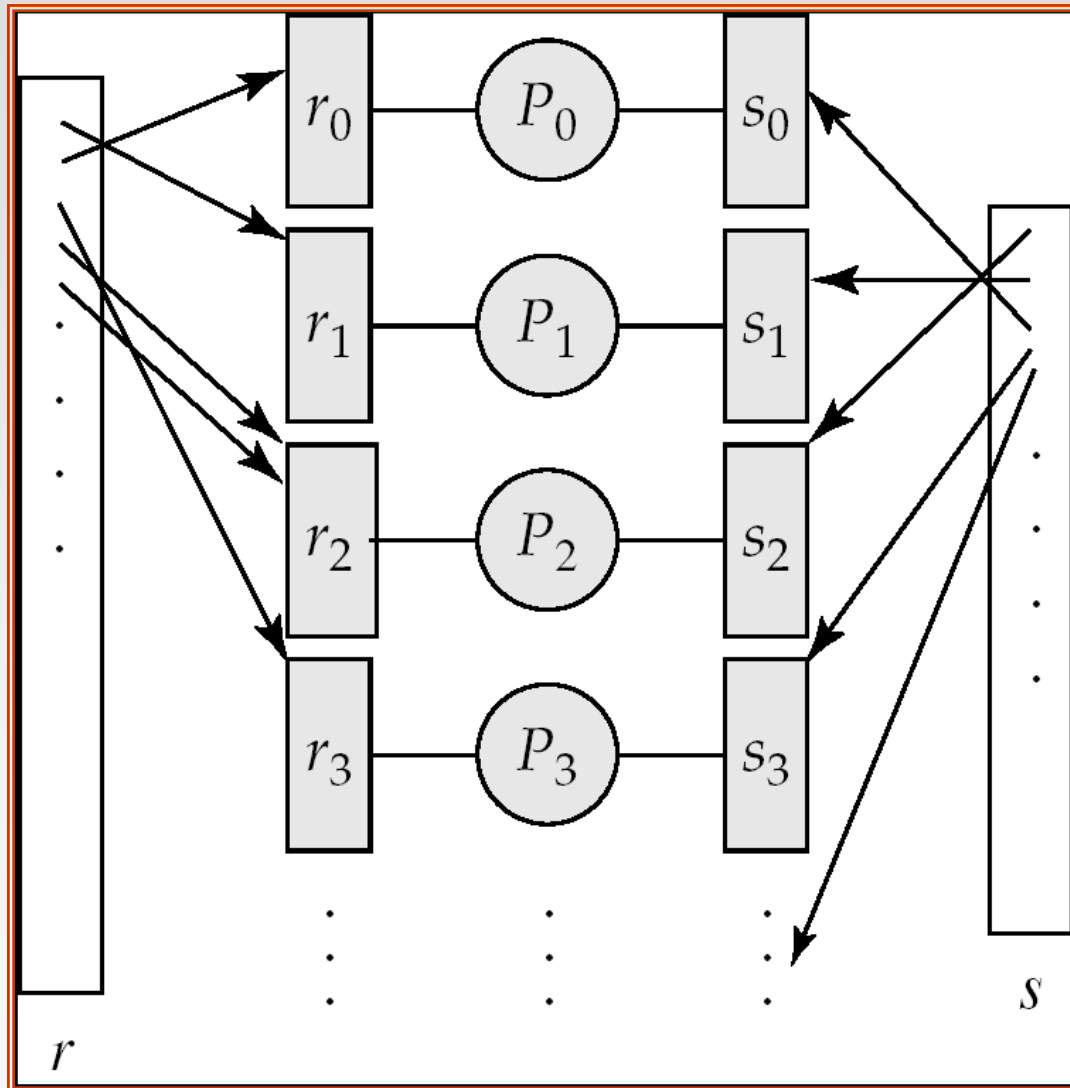
Parallel Join

- The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.
- In a final step, the results from each processor can be collected together to produce the final result.

Partitioned Join

- For equi-joins and natural joins, it is possible to *partition* the two input relations across the processors, and compute the join locally at each processor.
- Let r and s be the input relations, and we want to compute $r \bowtie_{r.A=s.B} s$.
- r and s each are partitioned into n partitions, denoted r_0, r_1, \dots, r_{n-1} and s_0, s_1, \dots, s_{n-1} .
- Can use either *range partitioning* or *hash partitioning*.
- r and s must be partitioned on their join attributes ($r.A$ and $s.B$), using the same range-partitioning vector or hash function.
- Partitions r_i and s_i are sent to processor P_i .
- Each processor P_i locally computes $r_i \bowtie_{r_i.A=s_i.B} s_i$. Any of the standard join methods can be used.

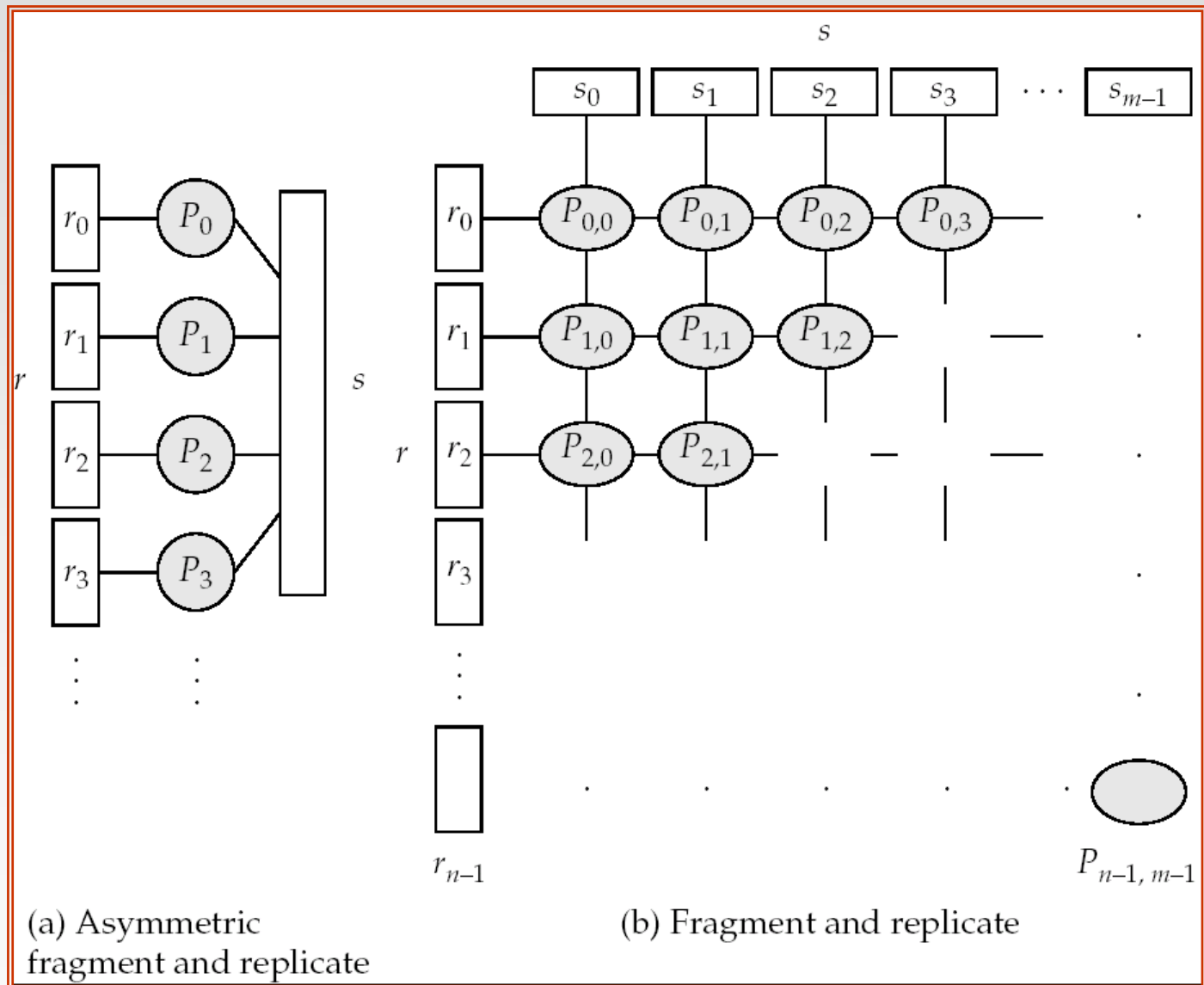
Partitioned Join (Cont.)



Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
 - e.g., non-equijoin conditions, such as $r.A > s.B$.
- For joins where partitioning is not applicable, parallelization can be accomplished by **fragment and replicate** technique
 - Depicted on next slide
- Special case – **asymmetric fragment-and-replicate**:
 - One of the relations, say r , is partitioned; any partitioning technique can be used.
 - The other relation, s , is replicated across all the processors.
 - Processor P_i then locally computes the join of r_i with all of s using any join technique.

Depiction of Fragment-and-Replicate Joins



Fragment-and-Replicate Join (Cont.)

- General case: reduces the sizes of the relations at each processor.
 - r is partitioned into n partitions, r_0, r_1, \dots, r_{n-1} ; s is partitioned into m partitions, s_0, s_1, \dots, s_{m-1} .
 - Any partitioning technique may be used.
 - There must be at least $m * n$ processors.
 - Label the processors as
 - $P_{0,0}, P_{0,1}, \dots, P_{0,m-1}, P_{1,0}, \dots, P_{n-1,m-1}$.
 - $P_{i,j}$ computes the join of r_i with s_j . In order to do so, r_i is replicated to $P_{i,0}, P_{i,1}, \dots, P_{i,m-1}$, while s_j is replicated to $P_{0,j}, P_{1,j}, \dots, P_{n-1,j}$.
 - Any join technique can be used at each processor $P_{i,j}$.

Fragment-and-Replicate Join (Cont.)

- Both versions of fragment-and-replicate work with any join condition, since every tuple in r can be tested with every tuple in s .
- Usually has a higher cost than partitioning, since one of the relations (for asymmetric fragment-and-replicate) or both relations (for general fragment-and-replicate) have to be replicated.
- Sometimes asymmetric fragment-and-replicate is preferable even though partitioning could be used.
 - E.g., say s is small and r is large, and already partitioned. It may be cheaper to replicate s across all processors, rather than repartition r and s on the join attributes.

Partitioned Parallel Hash-Join

Parallelizing partitioned hash join:

- Assume s is smaller than r and therefore s is chosen as the build relation.
- A hash function h_1 takes the join attribute value of each tuple in s and maps this tuple to one of the n processors.
- Each processor P_i reads the tuples of s that are on its disk D_i , and sends each tuple to the appropriate processor based on hash function h_1 . Let s_i denote the tuples of relation s that are sent to processor P_i .
- As tuples of relation s are received at the destination processors, they are partitioned further using another hash function, h_2 , which is used to compute the hash-join locally. (*Cont.*)

Partitioned Parallel Hash-Join (Cont.)

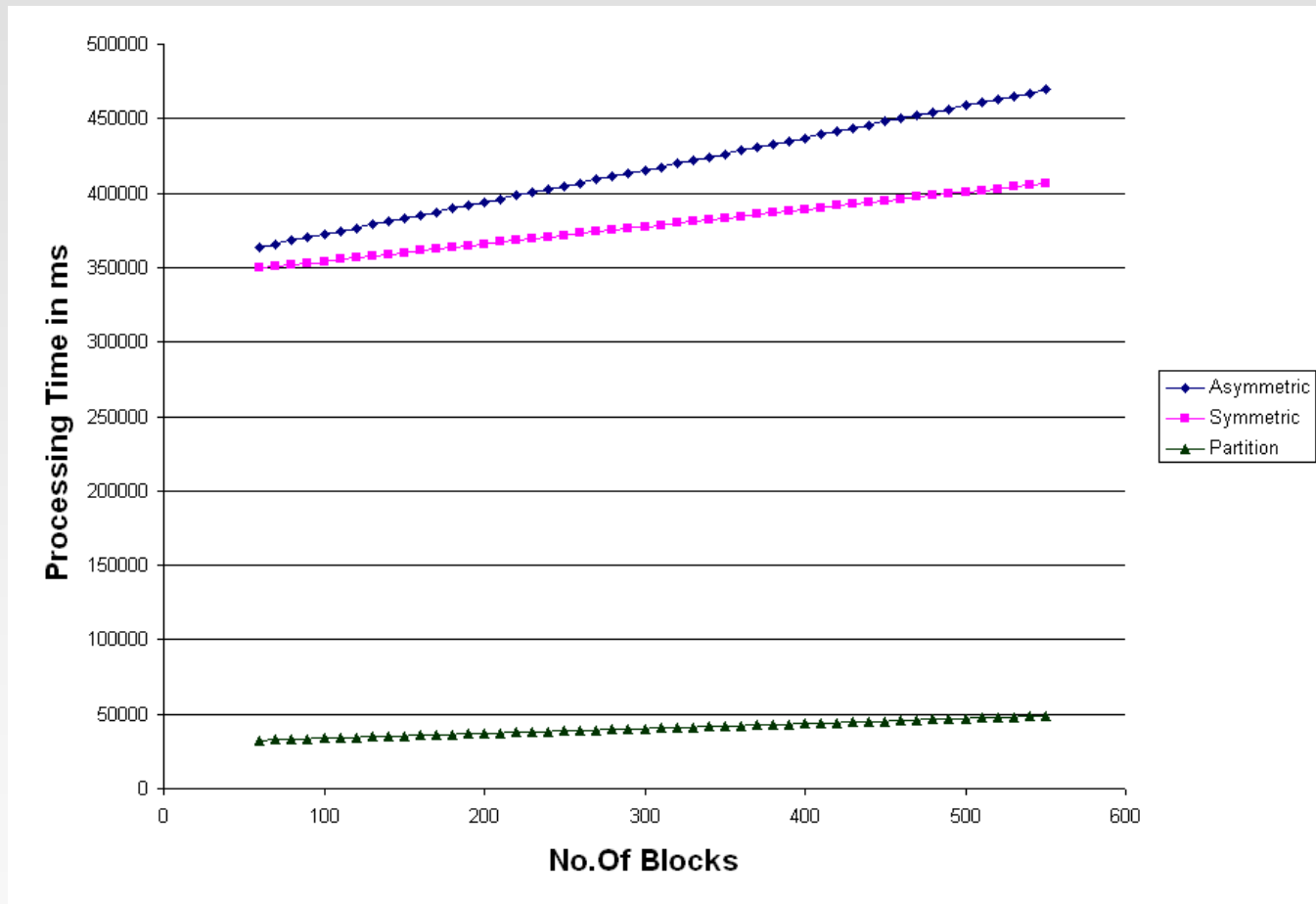
- Once the tuples of s have been distributed, the larger relation r is redistributed across the m processors using the hash function h_1
 - Let r_i denote the tuples of relation r that are sent to processor P_i .
- As the r tuples are received at the destination processors, they are repartitioned using the function h_2
 - (just as the probe relation is partitioned in the sequential hash-join algorithm).
- Each processor P_i executes the build and probe phases of the hash-join algorithm on the local partitions r_i and s of r and s to produce a partition of the final result of the hash-join.
- Note: Hash-join optimizations can be applied to the parallel case
 - e.g., the hybrid hash-join algorithm can be used to cache some of the incoming tuples in memory and avoid the cost of writing them and reading them back in.

Experimental results

- Set up of a parallel environment where we simulated parallel databases in 25 different computers in which data was stored in simple files.
- The texts were arranged in two dimensional matrix with tuples denoted in the rows and attributes denoted in columns.
- The computers were connected through LAN and the configuration of the each of the CPUs was: Pentium- 4 processor with clock speed of 1.86GHz, and as main memory RAM 1GB each.
- We simulated a simple program by which we generated random texts written in text files.
- We computed the time required to execute a join query in one of the nodes.
- By varying different parameters ,we simulated the different join techniques in parallel environment.

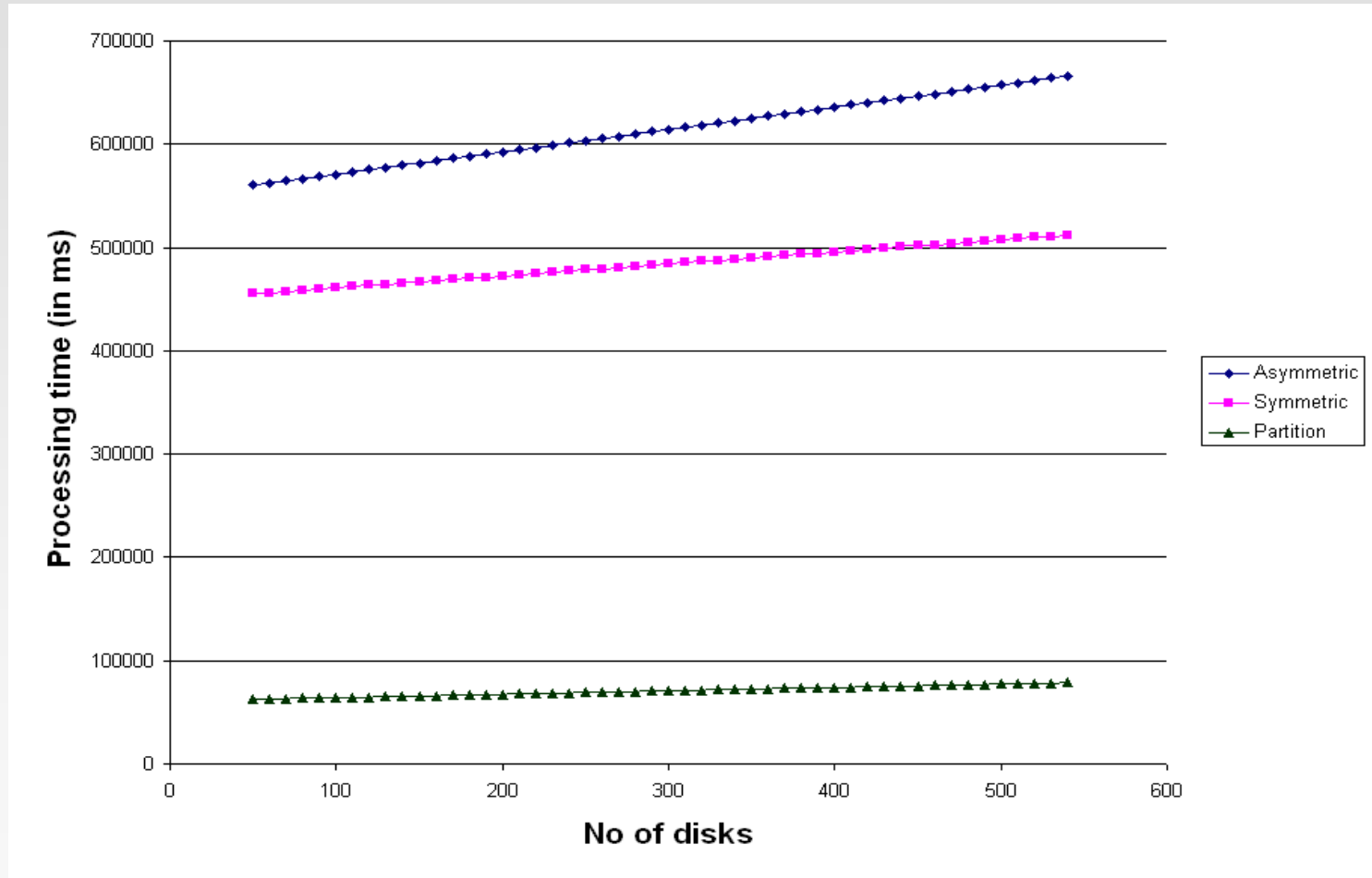
Experimental results—without query optimization

■ Varying block transfer time



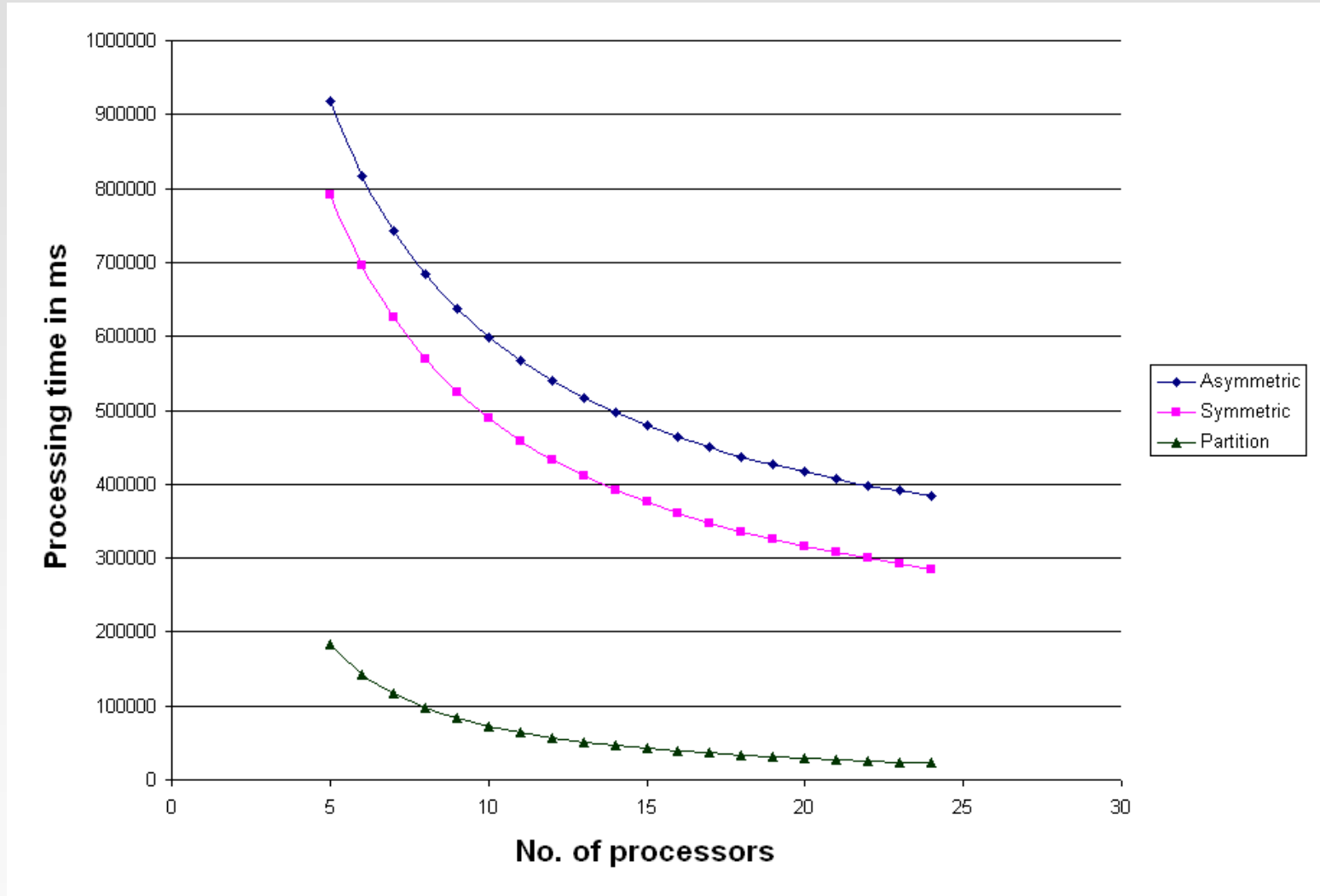
Experimental results—without query optimization (contd.)

- Varying disk access time



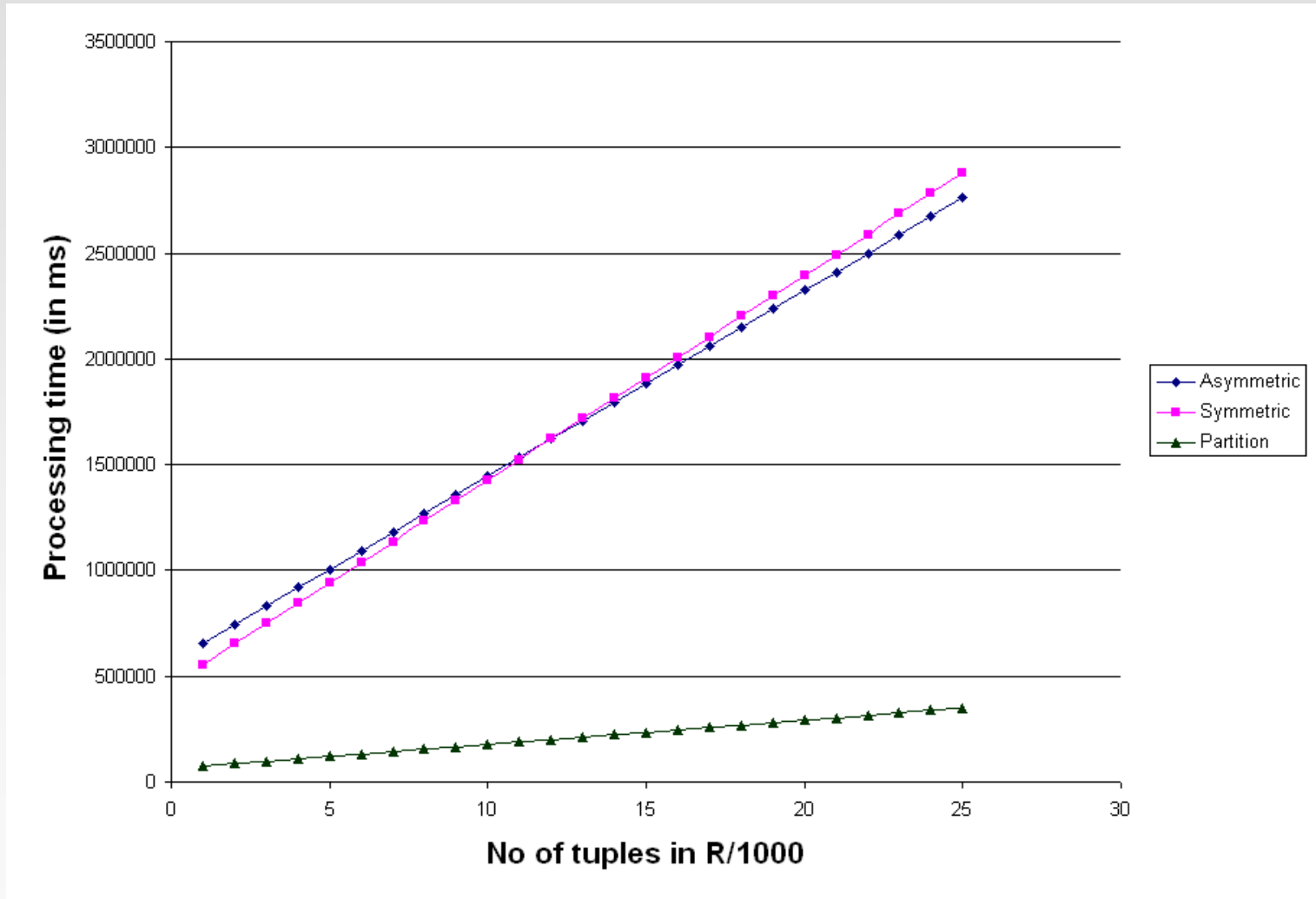
Experimental results—without query optimization (contd.)

■ Varying number of processors



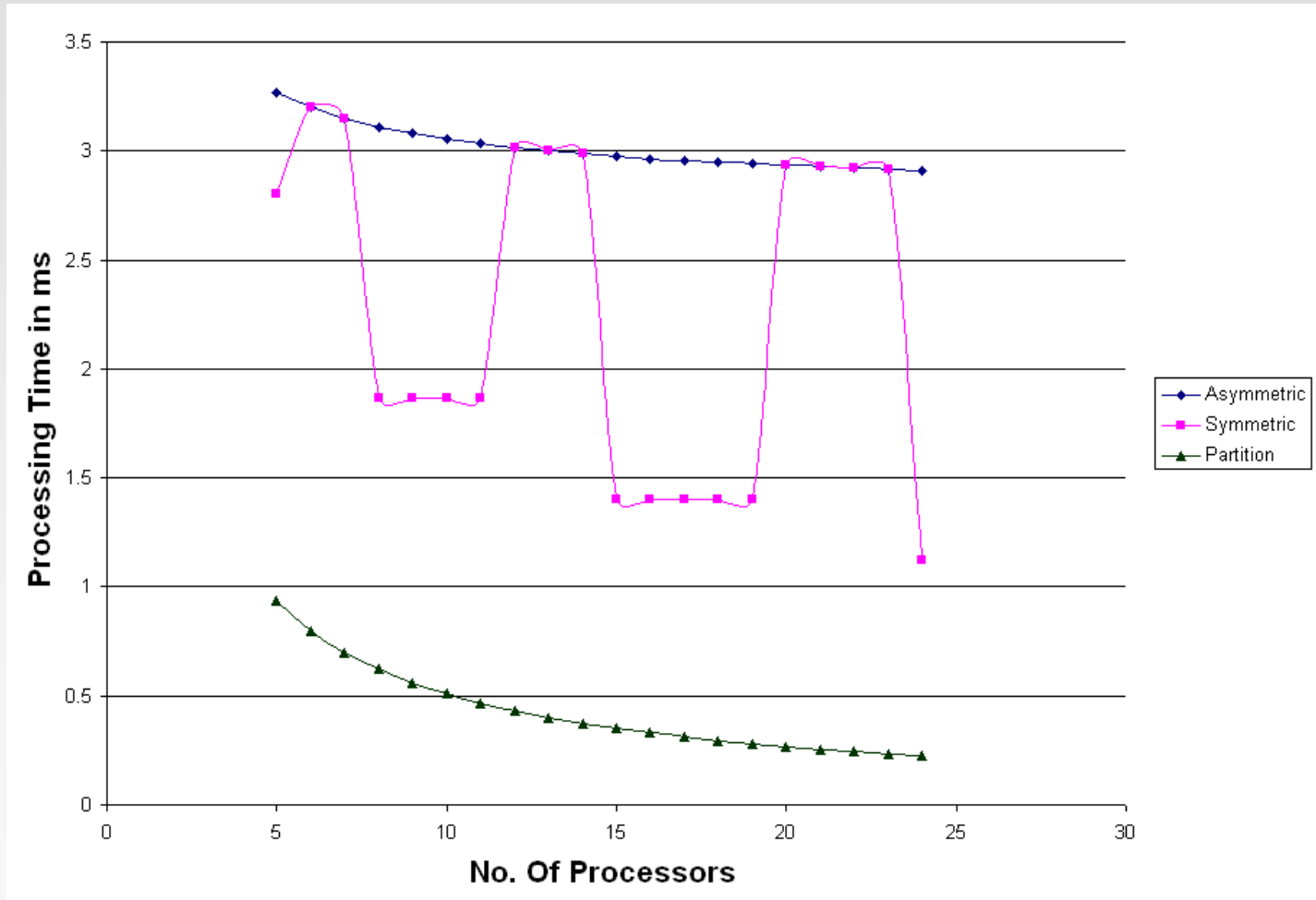
Experimental results—without query optimization (contd.)

- Varying R (the number of tuples that is sent to other nodes)



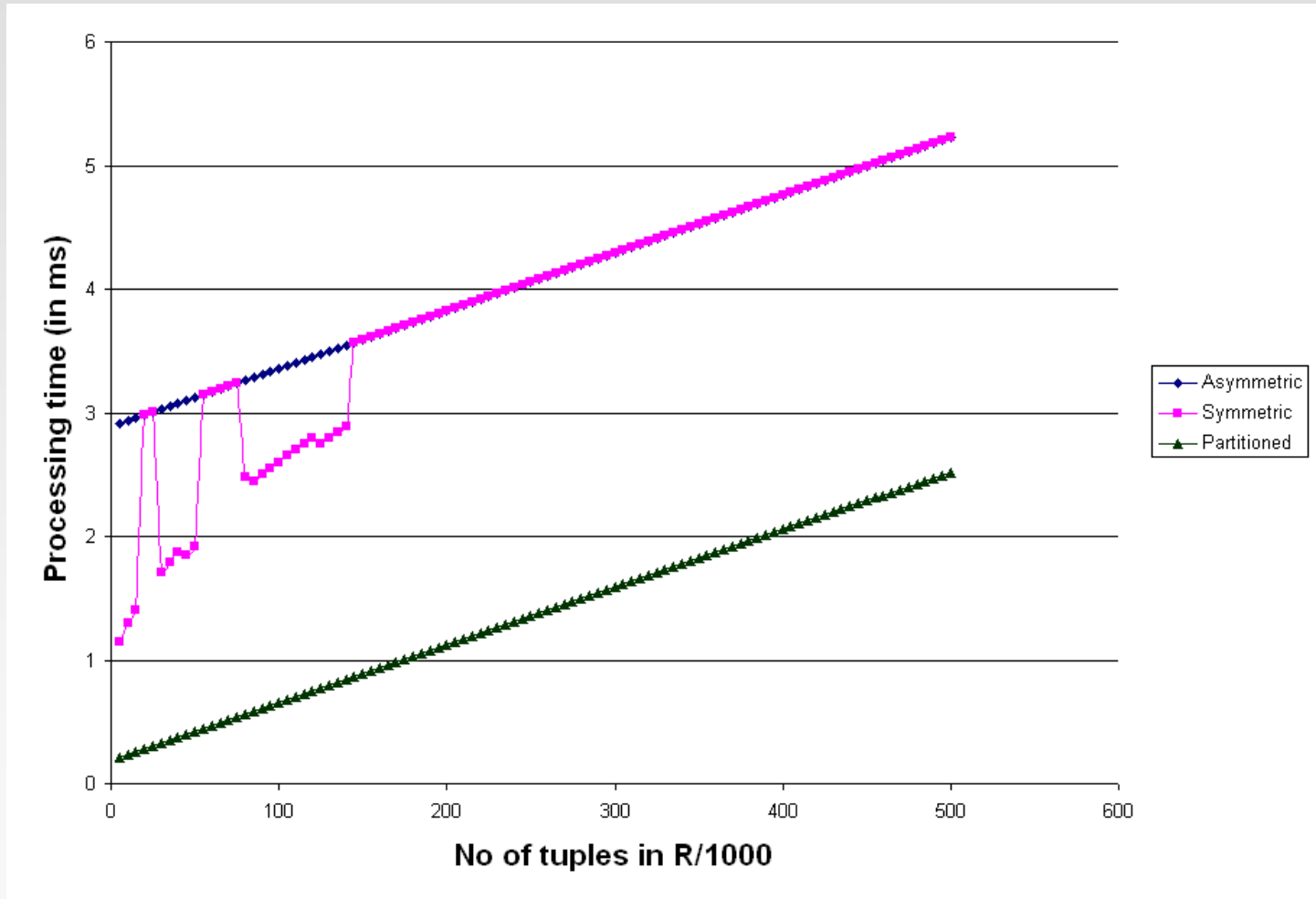
Experimental results—with query optimization

- Varying number of processors



Experimental results—with query optimization (contd.)

- Varying the number of tuples in R



Discussions

- When we considered a non-parallel environment without query optimization, the nested loop join, block nested loop join, indexed loop join and merge join showed almost the same characteristics in case of varying the block transfer time.
- When we varied disk access time and the number of tuples in R (the relation that has to be sent over the network) in single processor, the above join techniques and also the asymmetric and symmetric
- joins showed the same behavior.
- Both in single processor and parallel environment, partitioned join technique showed far better performance in all the variations of the different parameters.
- We find that highly dynamic scheduling methods based on observed execution times are superior in both complexity and attainable load balance.
- We also suggest the tuning of database schemata as a new anti-skew measure.

Discussions (contd.)

- The speedup is less dramatic because
 - Overhead is incurred in partitioning the work among the processors.
 - Overhead is incurred in collecting the results computed by each processor.
 - If the split is not even, the final result cannot be obtained until the last processor has finished.
 - The processors may compete for shared system resources.

Acknowledgement

The authors would like to thank Dr. A.S.M.Latiful Hoque, Associate Professor of Department of Computer Science and Engineering, BUET for his utmost contribution for the arrangement of permission to use the computer laboratory where the simulation was taken place.

Thank you!