Distrib Parallel Databases (2012) 30:145–176
DOI 10.1007/s10619-012-7090-6

Dynamic routing of data stream tuples among parallel query plan running on multi-core processors

Ali A. Safaei · Ali Sharifrazavian · Mohsen Sharifi · Mostafa S. Haghjoo

Published online: 8 March 2012
© Springer Science+Business Media, LLC 2012

Abstract In this paper, a method for fast processing of data stream tuples in parallel execution of continuous queries over a multiprocessing environment is proposed. A copy of the query plan is assigned to each of processing units in the multiprocessing environment. Dynamic and continuous routing of input data stream tuples among the graph constructed by these copies (called the *Query Mega Graph*) for each input tuple determines that, after getting processed by each processing unit (e.g., processor), to which next processor it should be forwarded. Selection of the proper next processor is performed such that the destination processor imposes the minimum *tuple latency* to the corresponding tuple, among all of the alternative processors. The tuple latency is derived from processing, buffering and communication time delay which varies in different practical parallel systems.

Parallel system architectures that would be suitable as the desired multiprocessing environment for employing the proposed *Dynamic Tuple Routing* (*DTR*) method are considered and analyzed. Also, practical challenges and issues for the proper parallel underlying system are discussed. Implementation of the desired parallel system on multi-core systems is provided and used for evaluating the proposed *DTR* method. Evaluation results show that the proposed *DTR* method outperforms similar method such as the Eddies in terms of tuple latency, throughput and tuple loss.

Communicated by: Mohamed F. Mokbel.

A.A. Safaei (✉) · M. Sharifi · M.S. Haghjoo
School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
e-mail: safaeei@iust.ac.ir

M. Sharifi
e-mail: msharifi@iust.ac.ir

M.S. Haghjoo
e-mail: haghjoom@iust.ac.ir

A. Sharifrazavian
School of Computer Engineering and IT, Sharif University of Technology, Tehran, Iran
e-mail: sharifrazavian@ce.sharif.edu

Keywords Data stream · Parallel processing · Query plan · Multi-core systems · Multistage Interconnection network

1 Introduction

Query processing in data stream systems is involved with new challenges regards to properties of data stream (i.e., infinite, continuous, rapid and unpredictable sequence of data elements). One of the most critical requirements of query processing in data stream systems is fast processing. Single processor data stream management systems are not capable of processing continuous queries over continuous streams with a satisfactory speed. So, parallel processing of queries using multiple processing units would be a good solution.

In [1], we proposed parallel processing of continuous queries in a multiprocessing environment. Assuming to have $\boldsymbol{K}$ identical logical machines (e.g., whether physical such as CPUs or GPUs or logical such as threads running on a multi-core CPU), a copy of the main query plan is assigned to each of logical machines [1]. Machines would collaborate to execute operators of the query plan in parallel. In order to do this, a graph consisting of all k copies of the main query plan is generated. This graph which is called *QMG* (*Query Mega Graph*) represents only a logical view of the machines. After that, operator scheduling process determines which machine should execute which operator of the query plan assigned to it. In other words, scheduling (on the *QMG*) determines that each input data stream tuple must visit which operator in which machine, to be processed [1]. This process (scheduling) is performed in an event-driven manner in [1] (i.e., whenever an input queue becomes full).

In order to be more compatible with the continuous nature of data streams as well as continuous queries, we proposed in [2], short-term continuous scheduling (called *Dispatching*) instead of the event-driven one, which provides a significant improvement of system performance [2].

In this paper, dynamic routing of input data stream tuples among operators of the *QMG* is proposed; theoretical concepts are based on the continuous scheduling method (*Dispatching*) and practical challenges and issues for designing and implementing proper parallel underlying system for employing the *Dynamic Tuple Routing* (*DTR*) method are proposed, too.

A background on the parallel query processing method we have proposed in [1] is briefly described in Sect. 2. Basic concepts and fundamentals of the proposed *Dynamic Tuple Routing* (*DTR*) are presented in Sect. 3 shows how employing *DTR* causes to minimize tuple latency (fast query processing). Employing this method in real-world applications needs operational parallel underlying system. Practical challenges and issues which we will deal with, are discussed and analyzed in Sect. 4. Implementation of the *DTR* over a multi-core system as the proper parallel system is presented in Sect. 5. Performance of the proposed *DTR* method over the implemented parallel system is evaluated in Sect. 6. Finally we review related work in Sect. 7 and conclude in Sect. 8.

2 Background

We presented a parallel DSMS in [1] in which there exist k identical machines (*logical machines*) which collaborate to process queries over input streams in parallel. For each registered query, k identical copies of query plan are generated and each copy is assigned to a machine. If an operator A sends its output tuples to operator B in a query plan, then each particular machine P_i is capable of sending output tuples of operator A to operator B in the next *level* of all machines (Fig. 1).

The k identical copies of query plan make a directed graph together. Two special nodes are added to this graph, *source* and *sink*. An edge connects the *source* node to each node in the first level. Also, an edge connects each node in the last level to the *sink* node. An edge connecting two nodes corresponds to a queue of operators. Weight of an edge indicates number of tuples waiting in the queue. The result is a weighted DAG named *Query Mega Graph* (*QMG*). In this way, the query plan is recognized by all machines and they are capable of collaborating for parallel execution of operators (Fig. 2).

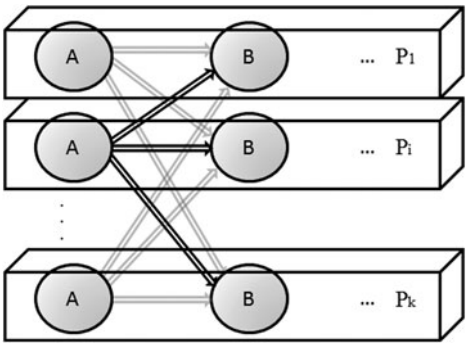


Fig. 1 Collaboration of logical machines for processing operators of a query plan

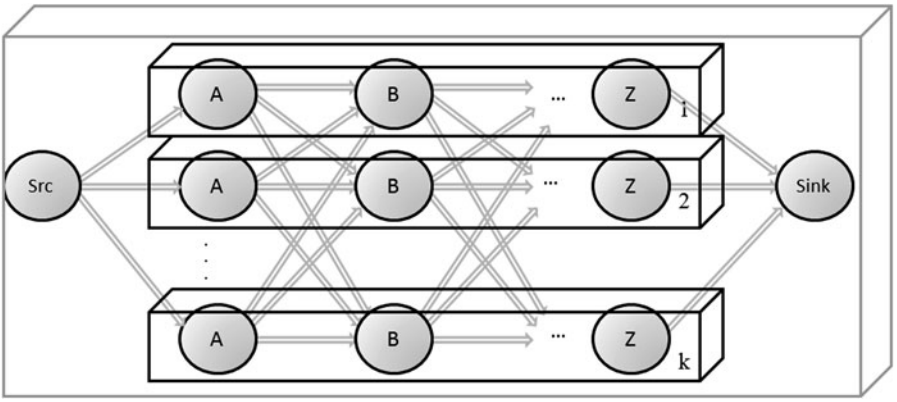


Fig. 2 Query Mega Graph

Notations[1]:

A query plan is a DAG $G_{QP} = \langle V, E \rangle$ in which:

$$V = \{o \mid o \in \text{stream query operators}\}$$
$$E = \left\{\langle A, B\rangle \mid A, B \in V\right\}$$

$\boldsymbol{O}^{\boldsymbol{i}}_{\boldsymbol{j}}$ denotes operator $\boldsymbol{O}$ which is the $\boldsymbol{i}$th operator of main query plan in machine $\boldsymbol{j}$.

$\boldsymbol{O}^{\boldsymbol{i}}_{-}$ denotes the $\boldsymbol{i}$th operator of main query plan, $\boldsymbol{O}^{-}_{\boldsymbol{j}}$ denotes operator $\boldsymbol{O}$ in machine $\boldsymbol{j}$ and $\boldsymbol{O}^{|\mathbf{V}|}_{-}$ denotes the last operator of main query plan.

Query Mega Graph created from $G_{QP} = \langle V, E \rangle$ is a triple $QMG = \langle V', E', W' \rangle$ such that:

$$V' = \{\text{src}, \text{sink}\} U \bigcup_{\substack{1 \le i \le |V| \\ 1 \le j \le K}} O^i_j$$

$$E' = E \cup \Big\{\langle x, y\rangle | \forall A, B \in V\big(\langle A, B\rangle \in E\big) \Rightarrow \big(\forall i = 1, 2, \ldots, k,\ \forall j = 1, 2, \ldots, k,\ i \neq j\big(\big\langle A^-_i, B^-_j\big\rangle \in E' \wedge \big(x = A^-_i \wedge y = B^-_j\big)\big)\big) \vee \big(\forall i = 1, 2, \ldots, k\big(\big(x = \text{src} \wedge y = O^1_i\big) \vee \big(x = O^{|V|}_i \wedge y = \text{sink}\big)\big)\big)\Big\}$$

$$W' : E' \to \mathbb{Z}^+ \quad \text{such that} \quad \forall a, b \in V'\big(W'(a, b) = q\_count(a, b)\big)$$

$q\_count(a, b)$ returns number of tuples waiting in queue of edge (a, b).

Note that number of nodes and edges in *QMG* can be computed as:

$$\big|V'\big| = \big(|V| \times K\big) + 2$$
$$\big|E'\big| = \big(|E| \times K\big) + (K - 1) \sum_{i=1}^{|V|} fan\_out_i + 2K$$

fan_out_i is the number of edges outgoing from node i.

The operator scheduling process finds the shortest path in *QMG* to minimize tuple latency. This path is a sequence of query plan operators distributed over $\boldsymbol{k}$ logical machines (Fig. 3).

After finding the shortest path, a triple $\langle p, o, s\rangle$ (*predecessor* machine, *operator_Id*, *successor* machine) is sent to each machine. In this way, operator path could be executed in parallel over $\boldsymbol{k}$ logical machines instead of serially in one. In [1], scheduling (finding the shortest path in *QMG*) is performed using the *Dijkstra's* shortest path algorithm when a queue becomes full (considered as an *event*). In contrast to *event-driven scheduling*, operator scheduling presented in this paper is performed continuously over query processing period.

In [1], query plan operators are first assigned to the machines according to (1):

$$j = i \,\boldsymbol{mod}\, k \tag{1}$$

[1] Notations are based on predicate logic in the *Z* notation [40].

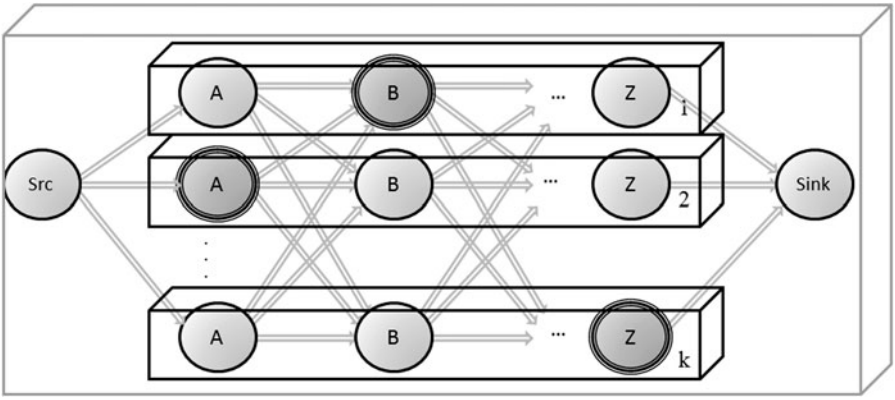


Fig. 3 A sample outcome of scheduling algorithm (routing in *QMG*)

(j: *machine_ Id*, i: *operator_Id* and k: *number of machines*).

By lapse of time, since operators have different processing speed (some are faster such as *projection* while others like *join* are much slower), number of tuples waiting in input queue of different operators may vary too much. When a queue becomes full, re-scheduling process should be performed, in which weight of edges are updated, then the shortest path is found via the *Dijkstra's* shortest path algorithm and triple $\langle p, o, s \rangle$ (*predecessor* machine, *operator_Id* and *successor* machine) is sent to each machine to set the operator path. The new path and the old one are used concurrently until all tuples waiting in queues of the old path are processed. The input ordering of tuples may change in parallel query processing engine. In order to preserve input ordering of data stream tuples, they are buffered and sorted in the *sink* node, before delivery.

Moreover, short-term (high frequency) scheduling of data stream query operators is proposed in [2] in which system's performance is improved as well as its adaptivity.

In this paper, a dynamic tuple routing method is proposed which can be used for processing data stream tuples by parallel processing of desired continuous queries in such a multiprocessing environment.

3 The proposed *Dynamic Tuple Routing* (*DTR*) method

In [1], tuple processing schema (traversing the query mega graph operators distributed over $\boldsymbol{K}$ logical machines) is determined by the scheduling process. This is performed via finding the shortest path in *QMG* that is done in an event-driven manner [1]. This causes fluctuations in system's performance level; when scheduling is done, the path that would be used is the best one (the shortest path) in which tuple suffers minimum latency. As the time proceeds, while queues are becoming full, tuple latency increases. So, that path would not be the best one and tuple latency is not the minimum anymore. Such situations would be continued up to the state in which one of the queues becomes completely full and excess tuples should be discarded. In this case, the reduced performance reaches to its minimum.

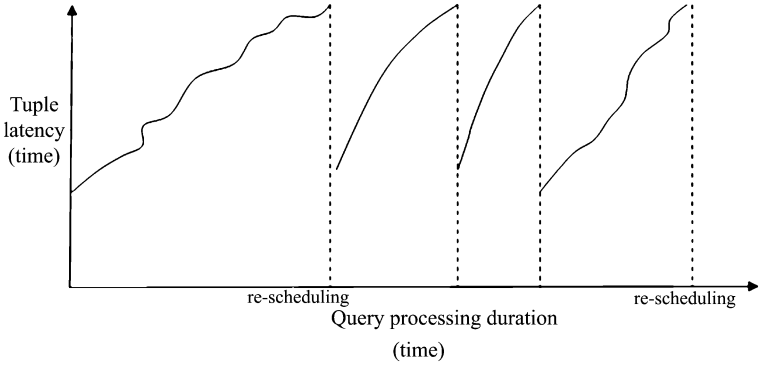


Fig. 4 Fluctuations of performance level caused by the event-driven scheduling [2]

By doing scheduling (re-scheduling) at this time (finding the new shortest path w.r.t. the updated weights of edges in *QMG*), tuple latency would be minimal again. An example of such scenario is shown in Fig. 4.

As a solution for this problem and reducing system performance fluctuations, operator scheduling (routing in *QMG*) can be performed with higher frequency (e.g., for each incoming data stream tuple).

The golden rule and strength for employing dynamic tuple routing method is the property that the *QMG* is of a specific and restricted type of graphs. In fact, *QMG* is a *multistage* graph (Theorem 1). A multistage graph is defined as a graph in which nodes are partitioned into sets (each set is called a *stage*); and each edge in this graph connects a node in stage i ONLY to one of the nodes in stage $i+1$. We call such transition (from a node in stage i to a node in stage $i+1$) a *step*.

Theorem 1 *The QMG is a multi-stage graph.*

Proof it is proved in [1] that query plan of a continuous query is a multi-stage graph. According to the definition, *QMG* is composed of k identical copies of query plan, plus nodes and edges which are added while preserving semantic of stages. It means that there is an edge from operator a in machine l to operator b in machine t in *QMG* if an edge from a to b exists in the main query plan:

$$\begin{aligned}&\left(\forall a, b \in V \wedge i, j = 1, 2, \ldots, |V| \wedge l, t = 1, 2, \ldots, k, \left(a_l^i, b_t^j\right) \in E'\right)\\ &\quad \Rightarrow \quad \left(a_-^i, b_-^j\right) \in E\end{aligned}$$

therefore, a is in the stage immediately before b.

Also, edges from the source node are connected only to nodes in first level of query plan and edges to the sink node are connected only from nodes in the last level.

Hence, all of the edges in *QMG* connect a node in stage i only to a node in stage $i+1$. □

In *Query Mega Graph* $QMG = \langle V', E', W' \rangle$ the notation $OS_i \subseteq V'$ shows the ith stage in which $V' = \bigcup_{i=1}^{L} OS_i$ such that:

$$\big(\forall i, j \therefore i \neq j.(OS_i \cap OS_j = \emptyset)\big) \\ \wedge \big(\forall e \in E', e = (a, b)\big(\nexists c, d \in V'\big((a, c) \in E' \wedge (d, b) \in E'\big)\big)\big)$$

In data stream query processing context, the path in query plan that each input tuple must traverse is known as *operator path* [3].

According to the definition of QMG and operator path, an operator path in *QMG* is a sequence of nodes located chronologically in stages 1 to L (L is number of stages in *QMG*):

$$Operator\_paths(QMG) \\ = \big\{\langle x_1, x_2, \ldots, x_L\rangle | (x_i \in OS_i \wedge x_{i+1} \in OS_j) \quad \Rightarrow \quad (j = i + 1)\big\}$$

Cost of each operator path in *QMG* is defined as the time delay each input tuple will suffer while traversing corresponding path [1]. It is shown in [1] that in order to have minimum tuple latency, the operator path that is selected for tuples to traverse, must be the shortest path. Finding the shortest path (routing in *QMG*) in [1] is performed using the Dijkstra's shortest path algorithm and repeated in an event-driven manner [1]. Regarding to the mentioned property of *QMG* (i.e., is a multistage graph), routing in *QMG* could be performed dynamically and continuously instead of event-driven using the Dijkstra algorithm.

Theorem 2 *The shortest path in QMG is a sequence of steps from stage* 1 *to stage L so that for each step, the edge with minimum weight* (*cost*) *among all corresponding edges is selected.*

Proof Sum of positive variables is minimum *iff* each variable has its minimal value. According to the definition of OMG and operator path [2, 3], an operator path in *QMG* (as a multi-stage graph) is a sequence of edges corresponding to steps 1 to L. Since weight of an edge (number of tuples waiting in corresponding queue) is always a positive value, sum of these weights is minimum *iff* each edge in this path has its minimal value.

In other words, let's assume that $D_a, D_b, \ldots, D_z$ are domain of values for $a, b, \ldots, z$ variables, respectively: $D_a, D_b, \ldots, D_z \subseteq \mathbb{N}$

$$a + b + \cdots + z = MIN\_VALUE \iff \Big(a = \min_{a_i \in D_a}\{a_i\}\Big) \wedge \Big(b = \min_{b_i \in D_b}\{b_i\}\Big) \wedge \cdots \\ \wedge \Big(z = \min_{z_i \in D_z}\{z_i\}\Big)$$

□

According to Theorem 2, in order to find the shortest path in *QMG*, for each tuple, in each step, we must determine and select the edge with the minimum cost value among all corresponding edges in that step (Fig. 5).

The reason is that, cost (i.e., latency) of an operator path is equal to summation of costs that traversing tuple suffers at each of steps (Lemma 2 in [1]).

As an axiom, summation of some integer numbers (e.g., cost of steps of an operator path) will have its minimum value when each of these integer numbers has its minimum value. So, in order to determine the shortest path in *QMG* to perform *Dynamic Tuple Routing* (*DTR*), for each input data stream tuple, in each step, we dynamically select the edge with the minimum weight (Fig. 5).

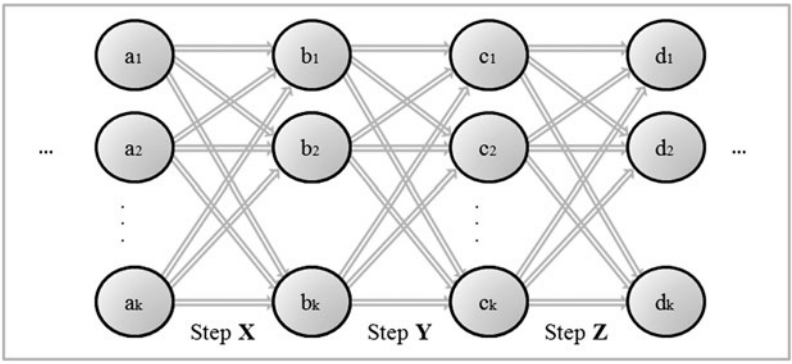


Fig. 5 Selecting minimum weight edge in each step in the *DTR* method

$$\begin{aligned} &\textit{If } \langle X, Y, Z\rangle \in \textit{Operator\_paths}(\textit{QMG}),\ \textit{then}:\\ &\textit{Cost}\big(\langle X, Y, Z\rangle\big) = \textit{MIN\_VALUE}\\ &\iff \quad \left\{\left(X = \min_{\substack{1\le i\le k,\\ 1\le j\le k}}^{\{q\_count(a_i,b_j)\}}\right) \wedge \left(Y = \min_{\substack{1\le i\le k,\\ 1\le j\le k}}^{\{q\_count(b_i,c_j)\}}\right) \wedge \left(Z = \min_{\substack{1\le i\le k,\\ 1\le j\le k}}^{\{q\_count(c_i,d_j)\}}\right)\right\} \end{aligned}$$

Such a determined operator path is the shortest path and has minimum tuple latency. We can conclude that routing algorithm used in [1] (i.e., Dijkstra's shortest path algorithm) employs the greedy approach whilst the *Dynamic Tuple Routing* (*DTR*) proposed in this paper employs dynamic programming approach; shortest path from the *src* node to the *sink* node of QMG consists of the edge with minimum cost from *src* to one of the nodes in next stage, named q (i.e., the min cost edge in the first step), concatenated with the shortest path from q to the *sink* node which is determined with the same manner, recursively.

Shortest path that is found by this method is the shortest one in that time instance (system snapshot). This means that, in each step, the best choice in that time instance is employed. So, although it may be possible that there exist a better path at some future time, but we do not have the chance of analyzing and using it at this time instance.

Therefore, the shortest path found by *DTR* is the best one w.r.t our current approach (selecting the best possible choice at each time instance).

In the proposed *DTR* method, at the beginning, the first operator of the main query plan is assigned to the first logical machine. Starting from first node of *QMG*, for each tuple, each node performs:

1. Executing its assigned operator on the tuple, and
2. Forwarding result tuple to the next proper node (i.e., the node in the next stage with the minimum cost).

By "cost of a step" we mean: the time a tuple suffer to be processed by destination node of the corresponding step. Generally, this cost consists of processing cost if the node (assigned operator's execution time) and buffering time (waiting in operator's input queue) [1]. Of course, in an operational and practical parallel underlying

system, as the desired multiprocessing environment, some other costs (e.g., communication costs) would be considerable. The cost model which is used for selecting min-cost edge at each step of DTR is presented in Sect. 4.6.

Since, by assigning a copy of main query plan to each of logical machines, they are aware of the query plan (operators and their arrangement), each node sends its result tuple beside the operator Id to the selected (min-cost) next node.

Pseudo-code of the proposed *DTR* and its time complexity analysis is presented in Sect. 4.6.

Employing this *DTR* method in real-world applications requires a proper parallel system as the desired multiprocessing environment. In next section, practical challenges and issues for such parallel underlying system are discussed and analyzed.

4 Practical challenges and issues

In order to implement multiprocessing environment of the proposed dynamic tuple routing method, many practical challenges and issues would be considered. Some of the most important ones are discussed below.

4.1 Context-switching

As selected edge with minimum weight changes dynamically, destination node (operator) migrates among different machines. From a machine point of view, it may execute different operators in each edge selection phase. Obviously, high frequency of changing operators for a machine to execute imposes considerable context-switching overhead for that machine. Some solutions for reducing this context-switching overhead are itemized below:

(a) Restricting domain of operators for each machine to execute
Domain of operators for each machine to execute can be restricted in two ways:

- Partition set of logical machines as well as set of main query plan operators into subsets, and assign operator subsets to logical machine subsets.
As a case, we may NOT copy all operators of query plan to all of the machines. In fact, lightweight operators such as projection which does not introduce long latency may not be replicated to all machines. Each such operator may be bound to a specific machine. Corresponding issue, called asymmetry of the machines, is discussed in Sect. 4.2. Although such approaches introduce some more complexities to the system's design and management, but would reduce context-switching overheads and communication costs (Sect. 4.6). The case that is mentioned above may lead to an extension of the proposed DTR and is leftover for future work.
- Assign copies of query plan to all machines, but set weight of *QMG* edges to ∞ (*infinite* value) if the edge's destination node is located in machine assigned to an undesired operator.

(b) Per-ω-tuples instead of per-tuple scheduling

The more the machine change frequency, the more the context-switching overhead. Since scheduling frequency of each operator is equal to the tuple arrival rate, the context-switching overhead can be reduced by modifying *routing* frequency from per-tuple to per-ω-tuples ($\omega > 1$). Finding an optimal value for ω is important because: (1) on one hand, *routing* is better to work per-tuple to be compatible with the continuous nature of data streams and continuous queries (2) on the other hand, the more the scheduling frequency, the more the context-switching overhead.

4.2 Symmetry of the machines

In a multiprocessing system, all of the machines (processing units) may have identical assigned functionality (i.e., SMP[2]) or some may have some special functionalities (i.e., ASMP[3] for example executing kernel-mode code of an OS on a processor while executing user-mode code on the other processors).

Although making these limitations simplifies design of multiprocessor systems but reduces system efficiency compared to the case in which processors are fully utilized.

In [1], since operator scheduling process is centralized, one of the processors is assigned scheduling process (i.e., scheduler machine) whilst the others are executing query operators. In contrast, the proposed *DTR* method, scheduling process (determining which operator must be executed on which processors) is performed dynamically, continuously and in a distributed manner by all of the processors (each machine itself determines the machine for executing next operator on its processed tuple). So, in *DTR*, parallel multiprocessing environment is SMP (compared to the PQP [1]) and employs processors utilizations more efficiently. Processor utilization comparison between PQP and *DTR* is analyzed in Sect. 6. Albeit, sorting the output result tuples is still performed in a specific processor in *DTR*, as well as in PQP.

4.3 Heterogeneity of the machines

Designing a practical system for DTR to be implemented and applied, requires to determine a proper heterogeneity level of machines (processing units e.g., GPP, GPU, DSP, FPGA, etc.).

In modern computing systems, level of heterogeneity between processing units is increased, as well as heterogeneity supports is provided in lower physical layers (i.e., in chip area)

In general, processors in heterogeneous computing may have different ISA.[4] More heterogeneity would be beneficial for computing systems that require better performance, reliability, reaction and correlation to the environment (e.g., network or control systems).

Heterogeneity of the processing units could be derived in subjects such as API,[5] low-level language capabilities, memory hierarchy, interface and interconnection.

[2] Symmetric Multi-Processors.

[3] ASymmetric Multi-Processors.

[4] Instruction Set Architecture.

[5] Application Programming Interface.

One of the most important heterogeneity subjects for the *DTR* is interconnection and communication model which processors in the multiprocessing environment must support. These will be determined when proper system architecture is analyzed and determined (Sect. 4.4).

Moreover, heterogeneity in processors' memory interface can be in a level in which length and structure of data (for data items that processors read, write and transfer) should be homogeneous. This is because they must support sliding window mechanisms. Also, supporting cache coherency protocols and NUMA access would be beneficial.

4.4 System architecture analysis

In this paper, dynamic routing of data stream tuples in parallel execution of continuous queries, over a multiprocessing environment is proposed. Multiprocessing means employment of more than one processing unit at a time in order to improve system throughput and performance. Based on the type of processing units employed, multiprocessing parallel systems are either *multicomputer* or *multiprocessor*.

Multicomputer systems in which some computers are connected using a communication network are suitable for applications that do not need high volume of communication between processing units. Since in DTR's multiprocessing environment there will be a high volume of communications (multiple communications for each data stream tuple), so multicomputer would not be a good choice.

Multiprocessor systems, in which some processors and some memory modules in a computer system are connected, are classified as follows [5]:

- *Tightly coupled:* primary memory is shared between processors such that each communication is performed via this *shared memory* (Fig. 6(a)).
- *Loosely coupled:* processors do not have shared memory and communication is performed via passing messages through communication network (*message passing* (Fig. 6(b)).

Shared memory as communication model cause to have high performance as well as low overhead via providing a global address space to programmers and sharing data between tasks; but in addition to problems that the programmer must handle (e.g., processors synchronization), weak scalability is the most critical shortcoming of shared memory communication model.

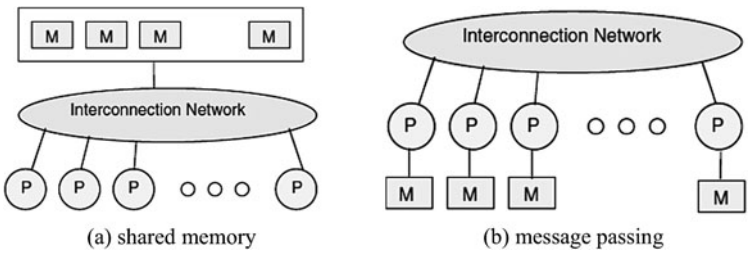


Fig. 6 Communication models

On the contrary, message passing provides high scalability as well as flexibility of employing off-the-shelf processors, but has high communication overhead in addition to complexity in communication management and synchronization.

Although due to the following reasons message passing seems to be more proper for multiprocessing environment of the proposed *DTR* method, but the two models are compared and analyzed in more details in Sect. 6:

(a) Message passing is known as suitable model for streaming applications [4].
(b) Communication between logical machines' operators (i.e., sending processed tuple beside operator $Id + 1$ to the next logical machine) essentially uses message passing.
(c) In data stream query processing context, operators are such that they modify structure of their input data instead of overwriting.

    For example, a *projection* operator selects only some of the all attributes and passes it to the next operator. A *selection* operator may discard (not to select) the considered tuple entirely; and a *join* operator may create a bigger data item for some of the tuples.

    Accordingly, shared memory would not be suitable due to its high overhead and complexity of memory management which is very frequent while message passing communication model seems to be more suitable. Note that, if the operators modify content of data items instead of its structure (i.e., overwriting data items such as in transaction management), then the shared memory model would be preferred; in shared memory, this can be handled simply by assigning the pointer to the modified data items in shared memory to the next processors.

Interconnection networks that are used to connect processors to each other (and/or to memory modules) are broadly classified in two classes:

(a) *Static networks* in which links are passive and like a dedicated bus between source and destination elements. Some well-known static network topologies are: completely connected, star, ring, star, linear array, mesh and hypercube [5, 6] (Fig. 7(a)).
(b) *Dynamic networks* in which links can be re-configured by switches. Some of the most well-known categories for dynamic networks are bus-based, crossbar and MIN (multistage Interconnection Network) [5, 6] (Fig. 7(b)).

Although it may sound to someone that the proposed architecture is somewhat similar to the completely connected or mesh networks, but in fact, it is a dynamic interconnection network not static. The reason is links between nodes in architecture of the proposed *DTR* multiprocessing environment are not passive and dedicated and not all of them are used every time (among all of the links between each a and b, only those with minimum cost would be used i.e., re-configurability). So, the interconnection network of the proposed *DTR* multiprocessing environment is a dynamic network that can use message passing communication model to send the messages (processed tuple beside its *operator* $Id + 1$) to each selected next processor. Table 1 illustrates a brief comparison between dynamic interconnection networks topologies.

Among different dynamic interconnection network categories, according to Theorem 1 (i.e., *QMG* is a multistage graph), *MIN* (Multistage Interconnection Network) is the most adequate one for our proposed DTR's system architecture.

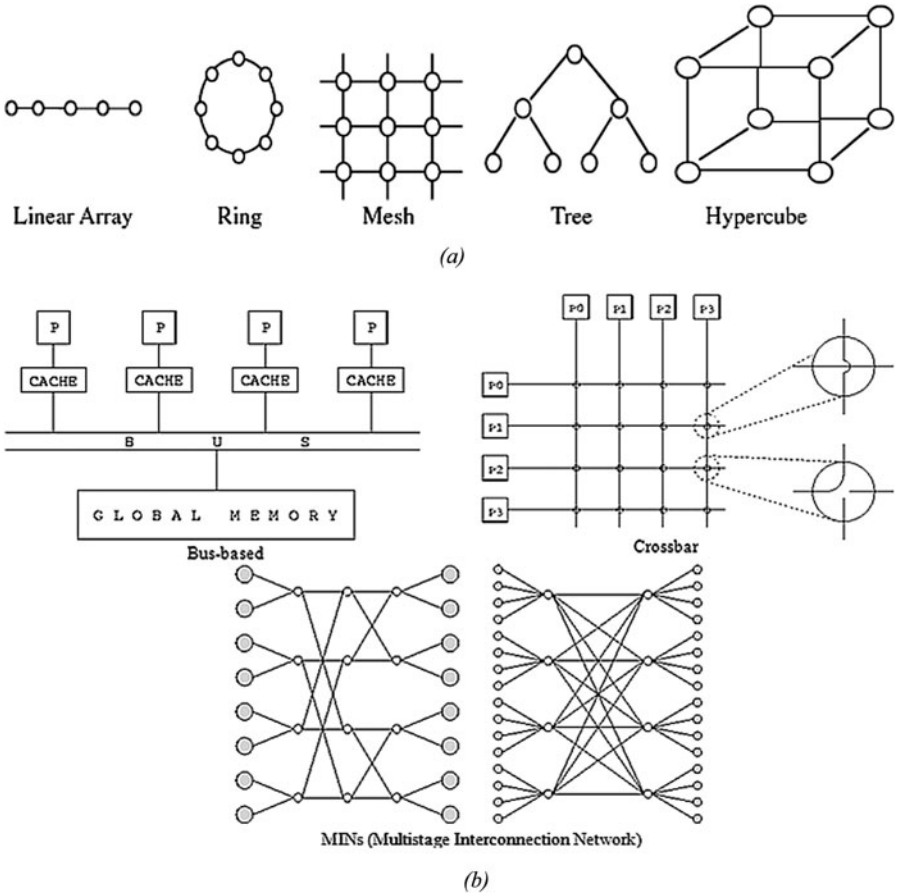


Fig. 7 Examples of well-known (**a**) static and (**b**) dynamic interconnection networks topologies [5]

Table 1 Comparison between dynamic interconnection networks topologies [7]

Property	Bus-based	Crossbar	Multistage
Speed	Low	High	High
Cost	Low	High	Moderate
Reliability	Low	High	High
Configurability	High	Low	Moderate
Complexity	Low	High	Moderate

In a MIN, processors are connected to the others via a set of *Switching Elements* (*SE*) such as in Fig. 8.

Each SE may have m inputs and n outputs (Fig. 9(a)) while switching policy determines how inputs are mapped to the outputs. For example, for a 2×2 SE, four switching policies can be defined as in Fig. 9(b).

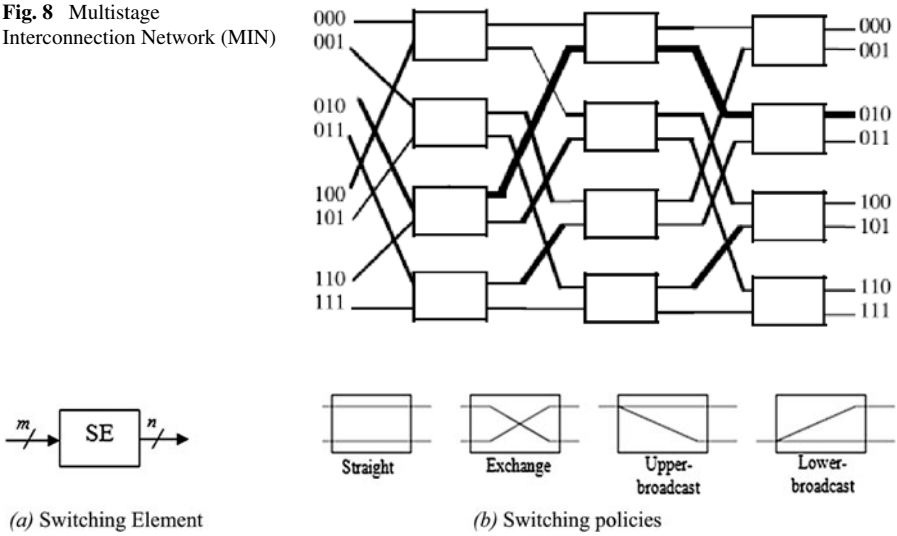


Fig. 8 Multistage Interconnection Network (MIN)

Fig. 9 Switching element and switching policies for a 2×2 SE

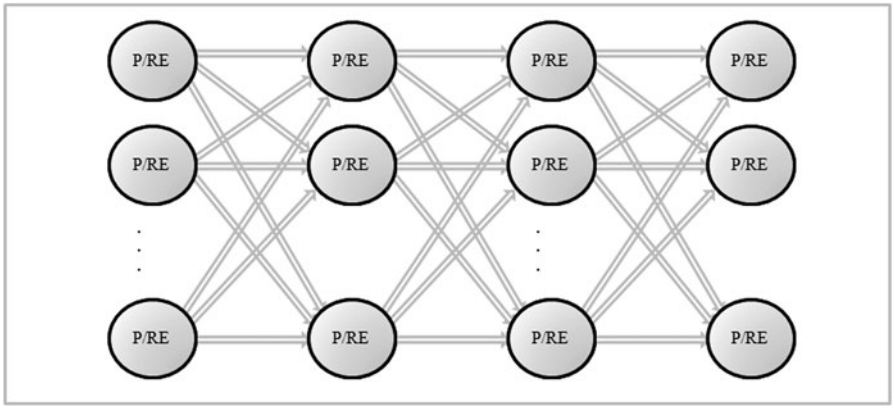


Fig. 10 The proposed *UACP–MIN*

According to the connection type (one-to-one, multipath or permutation, and also the Inter-Stage Connection (ISC) pattern [5], different types of MIN exist. Some examples are Delta [8], Omega [9], Butterfly [10] and certain permutation [11] MINs.

Routing in MINs is commonly static self-routing performed by SEs based on destination processor address of each message.

Interconnection network of the proposed *DTR* multiprocessing environment is a MIN partially different from the other ones. One of its distinguishing properties is this that the processors themselves act the role of switches (Fig. 10). It means that each of the processors, in addition to performing its assigned task (i.e., executing the operator on the received tuple), performs switching process to communicate with the

next processor that is in fact, a *per-hop min-cost routing* instead of switching. So, hereafter, we use "*Routing Element*" (*RE*) instead of the "*Switching Element*" (*SE*).

Number of inputs and outputs for each RE is f_i and f_o, respectively that are equal to $\boldsymbol{K}$ in general:

$$f_i = f_o = \boldsymbol{K}$$

Also, inter-stage connection pattern in the MIN of the proposed system architecture has two important properties as follows:

1. *Unidirectional*: unlike other generic MINs in which each processor is able to communicate with each other processors, communication in the proposed MIN is allowed only in one direction (from the processor m that executed operator O_i^m only to the processors that execute O_{i+1}^-, but not to the other ones). So, we name the proposed MIN a "*Unidirectional*" MIN due to this property.
2. *Adjutants-only*: each RE can communicate ONLY with its next stage directly-connected adjutants.
3. *Complete-Permutation*: each RE in each stage is connected to ALL of the REs in the next stage (Fig. 10). MINs with such inter-stage connection pattern are also called *baseline* [11, 12].

Accordingly, the interconnection network for connecting processing units in multiprocessing environment of the proposed *DTR* method is a *Unidirectional, Adjutants-only, Complete-Permutation, Multistage Interconnection Network* (*UACP–MIN*) as shown in Fig. 10.

Routing policy for this interconnection network is a *hop-by-hop*, min-cost routing in which each RE selects its next connected RE with the minimum cost. Cost model in which practical cost metrics of the selected system architecture are considered is proposed in Sect. 4.6.

The most important properties of a dynamic interconnection network for parallel processing are as follow [5]:

- low latency (delay)
- low cost (e.g., number of REs)
- high reliability (degree of fault tolerance)
- full access compatibility
- simple routing control

Comparison of the proposed unidirectional, adjutants-only, complete-permutation MIN with the other dynamic interconnection network categories is shown in Table 2:

Table 2 The proposed *UACP–MIN* vs the other dynamic interconnection network categories

	Delay	Cost	Blocking	Degree of fault tolerance
Bus-based	$O(N)$	$O(1)$	Yes	0
Crossbar	$O(1)$	$O(N^2)$	No	0
MIN	$O(\log N)$	$O(\log N)$	Yes	0
UACP–MIN	$O(\lvert V\rvert)$ and $\Omega_{(1)}$	$O(N)$	Yes	$(k-1).\lvert V\rvert$

Cost of an interconnection network which is defined as number of SEs [5] in the UAPC–MIN (in which no SE exists) is equal to Zero. But, since the processors themselves perform switching (routing) processes (Res instead of SEs) number of Res can be considered as network cost. In the proposed multiprocessing environment, if a processor is assigned to each of the nodes in QMG, number of processors (i.e., REs) is:

$$N = \big(|V| \times K\big) + 2$$

Albeit, the sink node of the *QMG* does not perform routing and hence number of Res will be $(N-1)$. Anyway, cost of the proposed *UACP–MIN* is of $O(N)$ as stated in Table 2.

In addition to better order of the *UACP–MIN* (in term of cost) rather than generic MINs, another advantage is that no special hardware element (as *SE*) is required, since processors themselves perform desired routing task.

Also, delay in an interconnection network is defined as delay of transferring an (empty) message through the network (between two processors, or a processor and a memory module), in the worst-case.

Theorem 3 *Delay of the UAPC–MIN is of $O(|V|)$ in which $|V|$ is the number of query's operators.*

Proof Generally, delay of a MIN is equal to number of its stages (i.e., number of hops the message should traverse) [5].

In the *UAPC–MIN* (which is based on the *QMG*, a multistage graph), number of stages is equal to number of stages in the *QMG*. QMG is semantically equivalent to the main query graph [1] and it preserves the set and arrangement of operators (only existent edges are replicated). So, number of stages in QMG is equal to number of stages of the main query graph. Number of stages in the main query's execution-plan graph is equal to number of operators in query-plan that are executed one after the other (i.e., equal to $|V|$). □

So, in the worst-case, delay of transferring a message (tuple) in the UAPC–MIN is of $O(|V|)$ and in practice, number of operators of a query is relatively small. Moreover, in the best-case, when the two source and destination processors are located in stages i and $i+1$ respectively, transferring a message is of $O(1)$. Therefore, delay of the *UAPC–MIN* will be of $O(N)$ and $\Omega(1)$.

Probability of transferring a message with delay of $O(1)$ is:

$$\begin{aligned}
&\mathrm{Prob}\big(\Omega(1)\big) \\
&= \frac{\sum_{i=1}^{N} \textit{number of states processor } P_i \textit{ send a message to its next stage adjustent processor}}{\sum_{i=1}^{N} \textit{number of states processor } P_i \textit{ wants to send a message to a processor}} \\
&= \frac{\textit{number of edges existent in the QMG (i.e., } |E'|)}{\textit{number of all possibleedges in the QMG (i.e., } N \times (N-1) \textit{ as in a completly connected graph)}} \\
&= \frac{(|E| \times K) + (K-1)\sum_{i=1}^{|V|} fan_out_i + 2K}{((|V| \times K) + 2) \times ((|V| \times K + 2) - 1)}
\end{aligned}$$

In other words, since each processor can send its tuple (message) to one of its $\boldsymbol{K}$ adjustment next processors and this will be replicated for each of N processors, the numerator is $(K \times N)$ whilst its denominator is $(N \times (N-1))$. Hence:

$$\begin{aligned} \text{Prob}\big(\Omega(1)\big) &= \frac{K \times N}{N \times (N-1)} = \frac{K}{N-1} = \frac{K}{((|V| \times K)+2)-1} \\ &\cong \frac{K}{|V| \times K} = \frac{1}{|V|} \end{aligned} \tag{2}$$

Practical analyzing of tuple latency (i.e., message delay) for the proposed DTR method over the underlying implemented system is provided in Sect. 6.

A network is blocking if some permutations of the network cannot be realized [12]. Since in the *UACP–MIN* some of the processors cannot be accessible through some others, there is some impossible permutation in the network and it is blocking, such as the generic MINs.

Finally, fault tolerance in the *UACP–MIN* is discussed in the next section.

4.5 Machine-failure handling

Generally, fault tolerance in parallel database systems is performed by re-executing the faulty query entirely. Since this approach is very costly and not efficient especially for large queries, some extended approaches (e.g., MapReduce [13, 14]) intends to restart only the faulty operator (instead of the whole query). These approaches impose high overhead to the system among its normal operation [14].

None of these are applicable for the proposed dynamic tuple routing method. Some reasons are as follow:

- The *MapReduce* operator is blocking [14] which is not proper to be used for data stream systems.
- Since in data stream query processing context, tuples that are processed and delivered as output results do not need (and also do not exist) to be processed again, there is no need to restart the faulty query or even one of its operators.

In general, fault tolerance is realized with the redundancy (e.g., code, data, etc.). Employing multiple processing units in the proposed *DTR*'s multiprocessing environment and assigning copies of the query plan to all of them, improves potential of fault tolerance via redundancy.

Also, since the *UACP–MIN* employed in *DTR* system architecture is a *multipath* MIN [7], so there are multiple path to be substituted with the faulty path. To re-direct tuples to a non-faulty path, cost of stages and steps are set to *8* for the faulty path (so, the cost model should consider the failure (Sect. 4.6)).

Machine (processing unit) failure is the most important type of failure that should be considered in such multiprocessing environment. Machine failure mechanism in the proposed *DTR* multiprocessing environment is as follow:

- *When failure happens*:
  (a) *faulty processors*: to inform (notify) the upstream processors about failure happening in order to prevent sending their next tuples.

Notification about failure happening can be performed via different mechanisms such as messaging, shared status flag and timeout. Anyway, to select and implement the proper mechanism, it must be considered that if the faulty processor is able to do any operation after its crash?

(b) *upstream processors*: setting cost of any outgoing edges to the faulty processor to 8, causing bypassing the faulty processor.

(c) *handling trapped tuples*: in order to handle the tuples *trapped* in the faulty processor's input queue, the tow basic strategies are:
  (i) *non-strategy*: trapped tuples are simply discarded.
  (ii) *re-directing-strategy*: trapped tuples are re-directed to one or some non-faulty, low-loaded processors.

An important challenge in the later strategy is preserving order of tuples (w.r.t. the trapped ones).

If shared memory is used as communication model, re-directing strategy can be implemented simply via assigning pointer of the trapped data tuples to the non-faulty processors.

- *After failure recovery*:

(d) *Faulty processors*: returning to the set of processing units via notifying its upstream processors.

(e) *Upstream processors*: *resetting* the outgoing edges' cost.

4.6 Cost model

Routing in the proposed *DTR* method is performed as a hop-by-hop, min-cost routing at each RE. So, each RE must compute the cost of forwarding its processed tuple to the next connected REs. Note that, the main goal of the *DTR* is to minimize tuple latency. Regarding practical challenges and issues (especially the selected system architecture), proper practical cost model which is used by each RE in the routing process is determined.

To achieve the proper cost model, we should walk through the steps in the proposed Dynamic Tuple Routing (*DTR*) method (stated in pseudo-code in Fig. 11).

According to the steps of the proposed *DTR* method (pseudo-code in Fig. 11), costs (in term of time) of the *DTR* steps can be detailed as follow. These costs are used by each processing units (per-processed tuple) to select the min-cost next processing unit, to pass the processed tuple to.

- C_{O_Sch}: Cost of scheduling of operators the processor should execute.
  (not taken into account in case that each processing unit (node of the QMG) is assigned to a dedicated processor, but should be considered if not, as in our implemented system (Sect. 5)).
- C_{T_Sel}: for each selected operator, cost of selecting the respective tuple among tuples in operator's input queue (i.e., the Round Robin scheduling algorithm).
- C_{T_Fet}: Cost of fetching the selected tuple (reading tuple from memory module and transferring to the processor via the cache memories, if used).
- C_{O_Exec}: Operator's execution time.
- C_{T_Wrt}: Cost of writing the result tuple.

```
1. DO IN PARALLEL, at each RE {
2.       Select a message (tuple,Op_Id) from RE's input queue, according to its queuing policy
   (i.e., Round Robin)
3. processed_tuple ← CALL Op_Exec (Op_Id + 1, tuple)
4. m ← Make_Message(processed_tuple, Op_Id + 1)
5.    min_cost = 0; min_cost_processor_Id = 1;
6.    { for(1 ≤ i ≤ number of outgoing edges from this operator to each of the other proces-
   sors)//(k − 1) times
7.       {//finding each destination processor's cost
8.       for (1 ≤ j ≤ number of processors that can send tuples to this processor)//i.e., (k − 1)
   times
9.             Compute_Costs ( P_j )// according to (3)
10.            if ( P_j costs < min_cost)
                        min − cost_processor_Id = j;
       }
11. Send_Message (m, min_cost_processor_Id);
12. }
```

Fig. 11 *DTR* in pseudo-code

  - *message startup time*: if message passing communication model is used.
  - *writing result tuple in processor's local memory*: if shared memory communication model is used
- C_{T_Rout}: time of routing (computing weights (costs) for each of its outgoing edged and selecting the min-cost edge for forwarding the result tuple).
- C_{T_Trans}: cost of transferring the result tuple to the next processor
  - *sending the message via links of the UACP-MIN*: if message passing communication model is used
  - *notification and synchronization of the next processor*: if shared memory communication model is used.

In general, the cost model is a function of processing costs, memory delay and communication costs (which are considered above). Physical characteristics of hardware elements in the operational underlying system (e.g., CPU speed, delay and bandwidth of memory module and Bus, and levels of cache memories, if used) are important factors that should be considered for more accurate computation of the costs. Roughly speaking, cost model can be simplified and stated as summation of the factors mentioned above:

$$Cost\left(O_m^i, O_n^{i+1}\right) = \left(m.C_{T\_Rout}^-\right) + \left(m.C_{T\_Trans}^-\right) + \left(n.C_{O\_Sch}^{i+1}\right) + \left(n.C_{T\_Sel}^{i+1}\right) + \left(n.C_{T\_Fet}^{i+1}\right) + \left(n.C_{O\_Exec}^{i+1}\right) + \left(n.C_{T\_Wrt}^{i+1}\right) \quad (3)$$

On the other hand, total time to transfer a message over a network comprises of the following [15]:

- *Startup time* (t_s): Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).

- *Per-hop time* (t_h): This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.
- *Per-word transfer time* (t_w): This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.

The total communication cost for a message of size m words to traverse l communication links (using the store-and-forward routing method) is:

$$t_{comm} = t_s + (mt_w + t_h)l$$

and

$$t_{comm} = t_s + mlt_w$$

whenever t_h is negligible (as is in most platforms) [parallel comp].

But, in the proposed DTR routing method, t_s (including C_{O_Sch}, C_{T_Sel}, C_{T_Fet}, C_{O_Exec} and C_{T_Rout}) is spent repetitively in each processing unit, and t_w (including C_{T_Wrt} and C_{T_Trans}) depends on the communication model that is used (i.e., near to zero for shared memory and a relatively considerable value for message passing).

So, assuming cost metrics of t_s to be equal for different nodes, communication cost for a tuple of size m to traverse $|E'|$ edges of the *QMG* (number of links in the *UACP–MIN*) can be stated as in (4):

$$t_{comm} = |E'| \times t_s \times m \times t_w \tag{4}$$

Also, time complexity of the DTR method (based on the pseudo code in Fig. 11) is as follows: In *DTR*, each processor participates in dynamic tuple routing in addition to executing the query plan operators that is assigned to it. Time complexity of executing a data stream operator over an input queue of size $\boldsymbol{n}$ tuples is assumed $O(n^2)$ where n is operator's window size [41–43]. Finding the edge with minimum cost and sending result tuple along with the header *Op_Id* to its destination has time complexity of $\theta(k^2)$. The reason is for each outgoing edge, destination processor's costs is computed using its $\boldsymbol{K}$ input edges, and is repeated for $\boldsymbol{K}$ outgoing edges (number of outgoing edges of an operator is $\boldsymbol{K}$ in *QMG*); in other words, according to the above pseudo code, selecting the min-cost processor and sending the result tuple has time complexity of $O(k^2)$, since outer loop has $(k-1)$ iterations and the inner loop has $(k-1)$ iterations.

So, time complexity of *DTR* for each operator in each processor (which being executed in parallel) is of $O(n^2 + K^2)$ where n is its operator's widow size and $\boldsymbol{K}$ is number of logical machines participating in parallelism, and both are small in practice.

5 Implementing on multi-core systems

In order to implement parallel system, as the multiprocessing environment for employing the proposed DTR method, some important considerations regarding to the environment's properties should be noticed; if we want to use a processor for each of

the processing units (nodes in the *QMG*), a large number of processors is needed but many of them might be idle or under-utilized (e.g., only one of the processing units from $\boldsymbol{K}$ ones in each stage would be employed). So, assigning one processor to each of the processing units leads to processing resources' waste.

Moreover, number of processing units (nodes in the *QMG*) depends on the semantic of the query and may vary for different queries (it is not a fixed value).

On the other hand, we need $\boldsymbol{K}$ logical machines that can work in parallel (simultaneously) to provide the ability of parallel processing.

Accordingly, a *Hybrid* parallel processing model is used in which each of $\boldsymbol{K}$ logical machines is considered as a (physical/hardware) processor whilst each of processing units (nodes in the *QMG*) considered as a thread in the corresponding processor. In fact, it is implemented as threads running in cores of a multi-core system.

The other important consideration in regard to the cost of the message passing communication model is the necessity of comparing it toward the shared memory approach. Although we have implemented pure message passing (nothing is shared—that causes costs to increase), using a hybrid communication model can be more effective. For example, a lot of messages are transferred to notify upstream processing units of downstream ones' cost. Storing their costs in a shared place could reduce message passing overheads considerably. Anyway, both communication models are implemented and compared in the performance evaluation (Sect. 6).

Whether using cache memory is beneficial or not is another challenge. In general, data reuse is critical for cache performance. Supporting state-full data stream query operators such as the Join raises the need for using cache memories in the parallel underlying system. The presented DTR method is implemented and compared with event-driven scheduling method proposed in [1] and Eddies algorithm proposed in [16].

The codes are implemented in Linux *kernel 2.6.32* Running on a multi-core processor *Core-i7* Machine. Linux kernel is used for the implementation mainly because it supports many low level APIs (e.g., for scheduling, messaging and etc.). In both DTR and the event-driven scheduling methods, every virtual machine is considered as a Posix thread implemented in C++. In the DTR method, communication between threads has been implemented in two following ways:

- *Shared memory*: To implement shared memory, every thread defines public variables so other threads can access them.
- *Message passing*: In order to implement messaging, the Message Passing Interface (MPI) has been used.

Event-driven scheduling method is implemented with shared memory access and Eddies [16] implemented using message passing.

Since the aim of implementing these methods was to evaluate their performances, the queries are hard-coded in the implemented software and no parser for query language has been provided in these experiments.

6 Evaluation

6.1 Experimental setup

The dataset used for the evaluation is Dec-Pkt-1 [17]. This dataset contains 3600 seconds of TCP and UDP packet traffics. Each tuple consists of the following 6 attributes: *Time stamp*, *Source Address*, *Destination Address*, *Source Port*, *Destination Port* and *data size*. The arrival rate of tuples is 770 tuple per seconds and the arrival distribution over time is uniform.

The queries consist of selection, projection, join and group-by (average) operators. Totally, 10 different query types have been used for evaluation and each query type occurs twice in the time duration that is 3600 seconds. Queries are arrived to and removed from the system dynamically during the run time. In Table 3 each query type is shown according to its operators and the start and ending time by minutes for both times it processes.

The arrival distribution of queries is uniform over the time and the average of 3.3 queries is getting process at each sampling time. Each query resides for average duration of 1210 seconds in the system. The distribution of queries over time is presented in Fig. 12.

Table 3 Query set specifications

	Selection	Join	Projection	Group-by
Q1	√	√	√	√
Q2	√	–	√	√
Q3	√	√	√	–
Q4	√	√	–	–
Q5	–	–	√	√
Q6	–	√	–	√
Q7	–	–	√	–
Q8	√	–	√	√
Q10	–	–	–	√

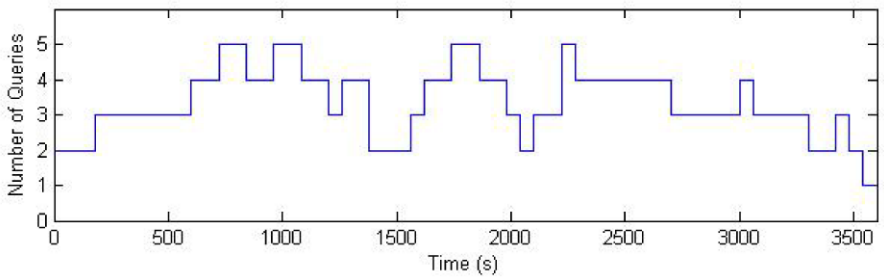


Fig. 12 Number of queries resides in the systems in each sampling time

The proposed DTR method with both two communication models (Shared Memory called *DTR-SM* and Message Passing called *DTR-MP*) is compared with the event-driven scheduling (called EDS) method [1] and the Eddies [16].

Based on [18], measured parameters are:

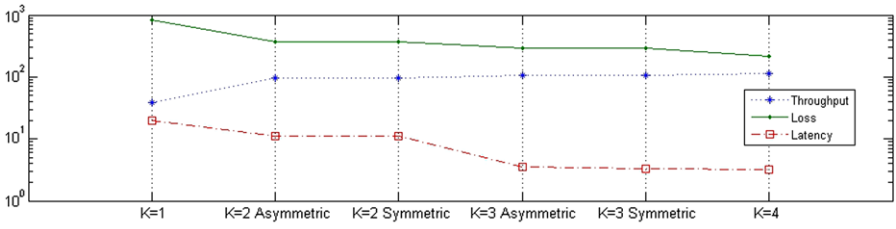
- *tuple latency (or response time)*: Time in milliseconds that takes a tuple to pass through the System
- *memory usage*: The size of tuples stored in the buffers
- *system throughput*: number of output tuples
- *tuple loss*: number of tuples that fail to process due to buffer overflow

6.2 Experimental results

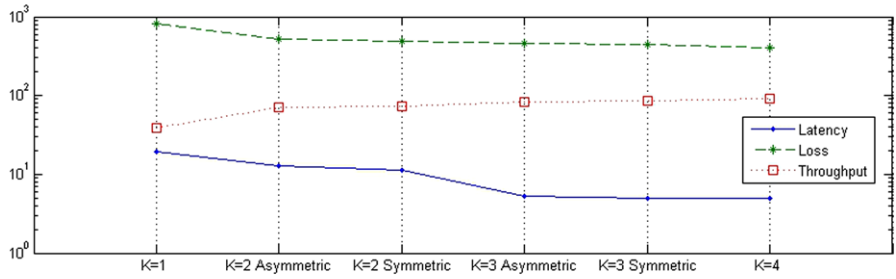
Before analyzing performance of the proposed *DTR* method and comparing it with the event-driven scheduling method and the Eddies, system specifications must be determined. One of the most important specifications of the *DTR* is the number of machines/cores (i.e., $\boldsymbol{K}$) that are used. Figure 13 shows values of performance parameters based on number of cores. Since the scheduling and memory management task is done by the Linux kernel, two mode of symmetric and asymmetric load balancing has been evaluated.

Generally, $\boldsymbol{K}$ represents the number of cores which are running the *DTR*. In symmetric mode, the Linux kernel and every other process run on another core and which is not taken into the account in $\boldsymbol{K}$. In asymmetric mode, those process run on the same cores that handles the *DTR* method.

A tradeoff must be made between performance and resources required. Respect to Fig. 13, when there are 3 asymmetric cores, the *DTR* method has better performance

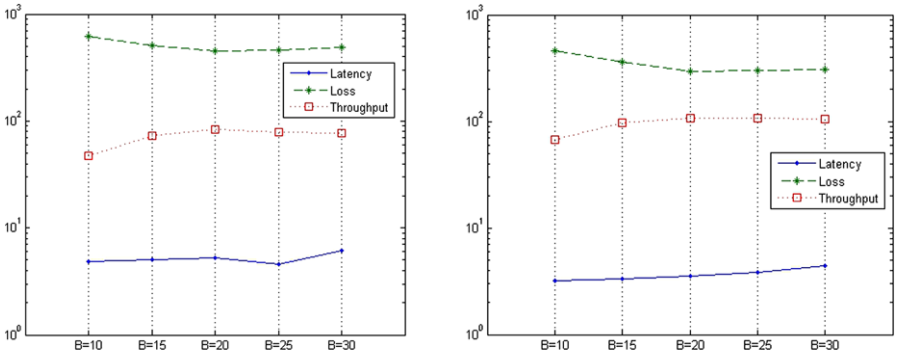


(a) DTR-SM



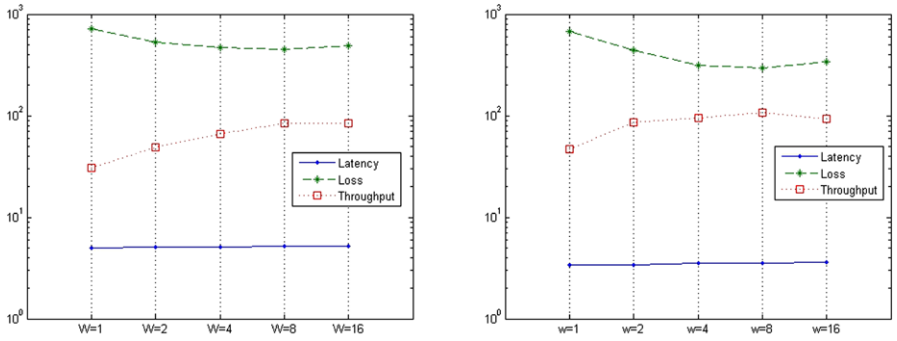
(b) DTR-MP

Fig. 13 Performance parameters vs. number of cores



(a) DTR-MP *(b)* DTR-SM

Fig. 14 Performance parameters vs. buffer size



(a) DTR-MP *(b)* DTR-SM

Fig. 15 Performance parameters versus $\boldsymbol{\omega}$

whilst using minimum resources. It means that in case of using 3 symmetric cores, we have 25% resources wasting due to the fourth one. So, number of cores in our system is set to three, hereafter ($\boldsymbol{K} = 3$).

Furthermore, size of operators queue (buffer size) is analyzed in Fig. 14.

Since buffer size has reverse impact on the two parameters (tuple latency and tuple loss), a tradeoff buffer size of 20 is selected ($\boldsymbol{B} = 20$).

In order to decrease the context-switching overhead, dynamic tuple routing is performed for each $\boldsymbol{\omega}$ tuples (instead of per-tuple). To determine the proper value of $\boldsymbol{\omega}$, system performance parameters are analyzed while increasing $\boldsymbol{\omega}$ (Fig. 15).

Based on the Fig. 15, the value of $\boldsymbol{\omega} = 8$ is used in the DTR method.

Accordingly, henceforward, the proposed *DTR* method with three asymmetric cores ($\boldsymbol{K} = 3$), buffer size of 20 ($\boldsymbol{B} = 20$)) and using the proposed *DTR* for each 8 tuples ($\boldsymbol{\omega} = 8$) is compared with the EDS method and the Eddies.

Figure 16 illustrates tuple latency, throughput, memory usage and tuple loss for these methods w.r.t. the time duration.

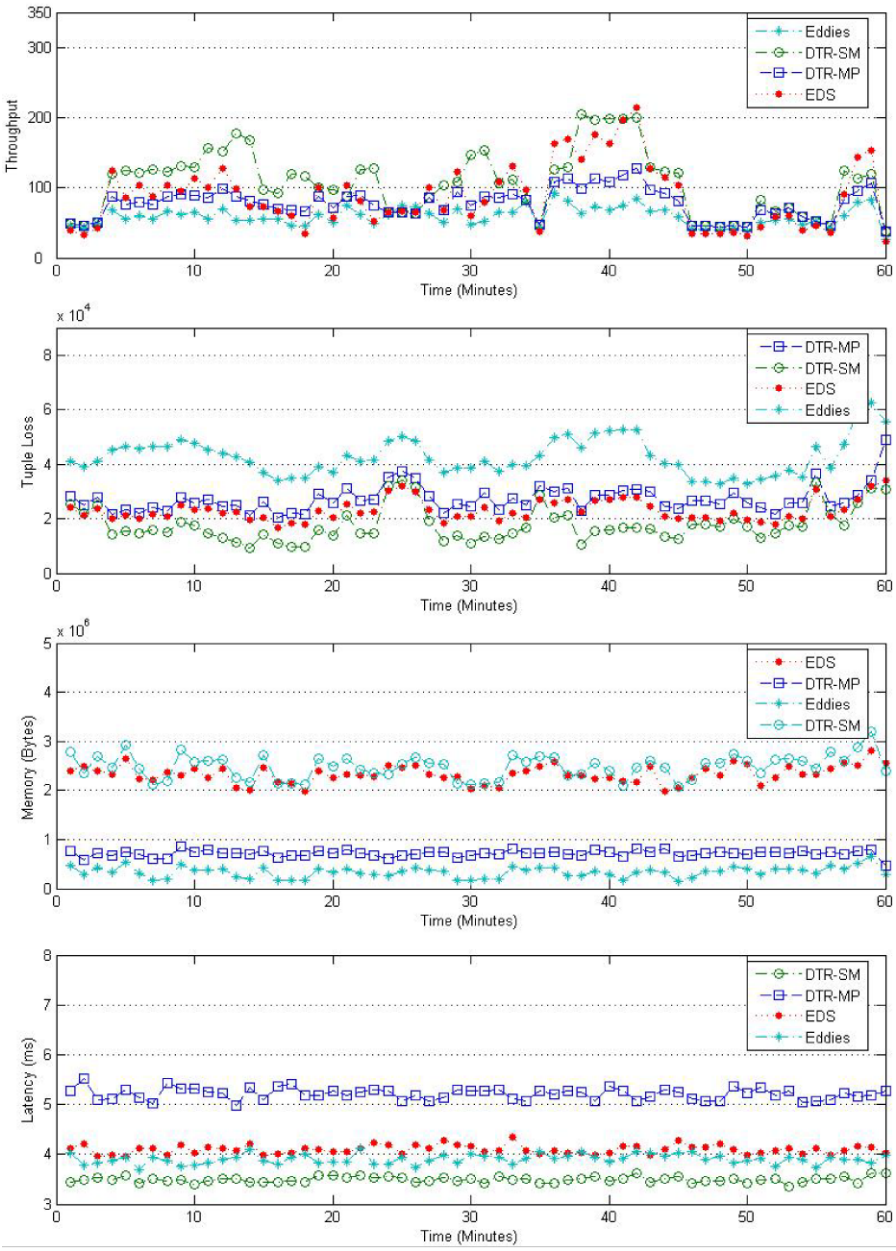


Fig. 16 Performance parameters vs. time duration

Charts in Fig. 16 show that the *DTR-SM* method (the proposed DTR method with Shared Memory) outperforms the EDS (Event-Driven Scheduling) method, the Eddies and event DTR-MP (DTR with Message Passing) in terms of tuple latency,

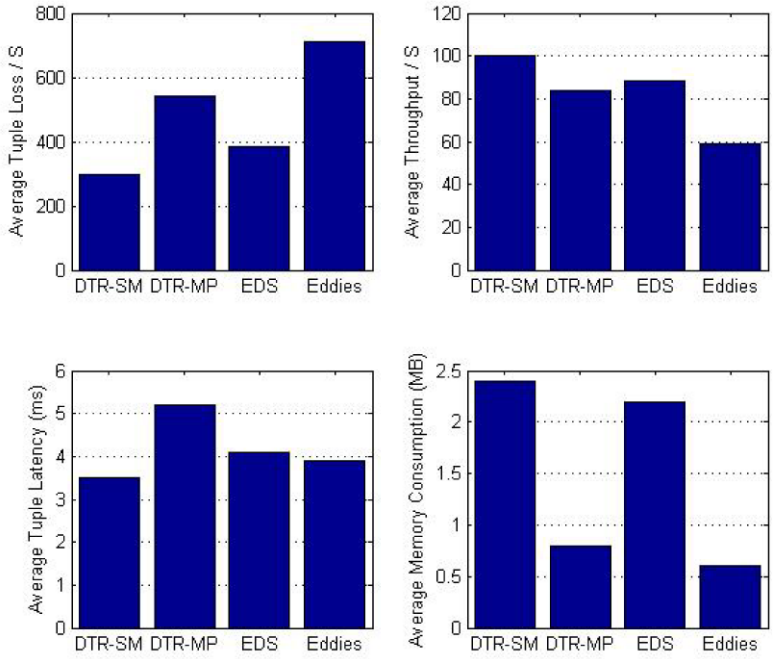


Fig. 17 Average performance parameters

throughput, tuple loss and average of those parameters is shown in Fig. 17. But the following notes should be considered, too:

- Tuple latency is measured in milliseconds. So, differences e.g., between DTR-SM and DTR-MP (2 ms) are not significantly considerable.
- Tuple loss in DTR-SM is near to half of the tuple loss in DTR-MP. Also, number of tuples which are lost is considerable (300 tuples in former whilst 600 in the later).
- The DTR-SM method requires to have shared memory which may be not possible in all of the parallel system architectures. This is one of the cons of the shared memory communication model rather than the message passing.
  Of course, size of the required memory modules is not a disturbing limitation for the system, even for the DTR-SM (i.e., 3 MB).
- Although message passing communication model can be employed in most of the parallel system architectures, its major disadvantage is its costs and overheads.
  It is shown in [19] that an implementation based on the shared memory might operate worse than the one based on the message passing, especially due to its communication overheads.
- Commonly, overheads in the DTR-MP are more compared to DTR-SM, and these overheads mainly regard to transferring control messages (metadata).
  We have implemented pure message passing (nothing is shared) in our system. Using a hybrid communication model can be more effective. Since, for example, a lot of messages are transferred to notify upstream processing units of downstream

one's cost; storing their costs in a shared place could reduce message passing overheads considerably.
– On the other hand, multi-core systems are designed for UMA system architectures, and hence have better performance while using the shared memory. Also, multithread parallel processing model uses a variant of the shared memory communication model for interaction between the threads.
– Therefore, implemented parallel system that is in fact a multithreaded multi-core system provides better performance, as it was expected.

As stated before, shared memory may not be applicable (supported) in some parallel system architectures. So, in this paper, both of the shared memory and message passing models are analyzed and evaluated.

6.3 System utilization monitoring

In order to evaluate the overhead of algorithms, for each core (i.e., CPU core), CPU utilization is monitored based on its variation of tasks (Fig. 18).

Processing Time is the percentage of time the core spends to handle the operator that is assigned to it. Higher percentage means the core spent less time on handling the overheads. Clearly, *DTR* method uses the best of CPU utilization among these methods. *Memory Access* time indicates the overall percentage of time in which the core is allocating or freeing the memory or reading from or writes to a buffer. *Routing* time is the time when core spends to find the next destination of a tuple. *Messaging* time is the sum of the times that core takes to communicate via other threads through messaging API. This time includes the time of sending and receiving tuples as well as communicating with other threads to estimate the load of a machine.

Experimental observation shows that messaging is a slow API and each message takes at least 300 μs to reach its destination which explains the reason why DTR-MP

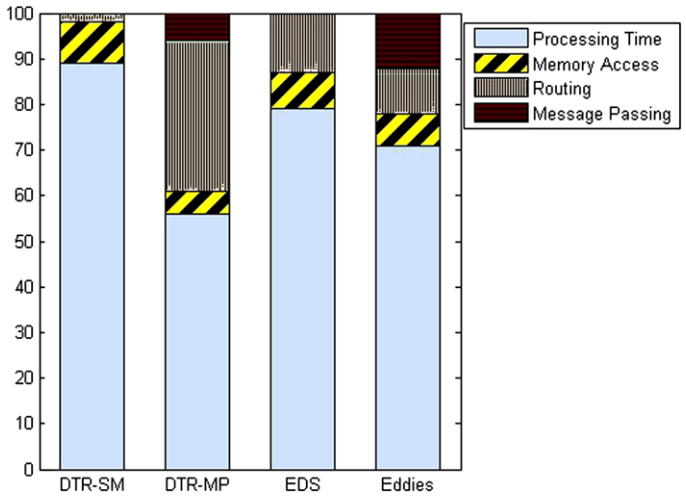


Fig. 18 Average CPU time based on its tasks

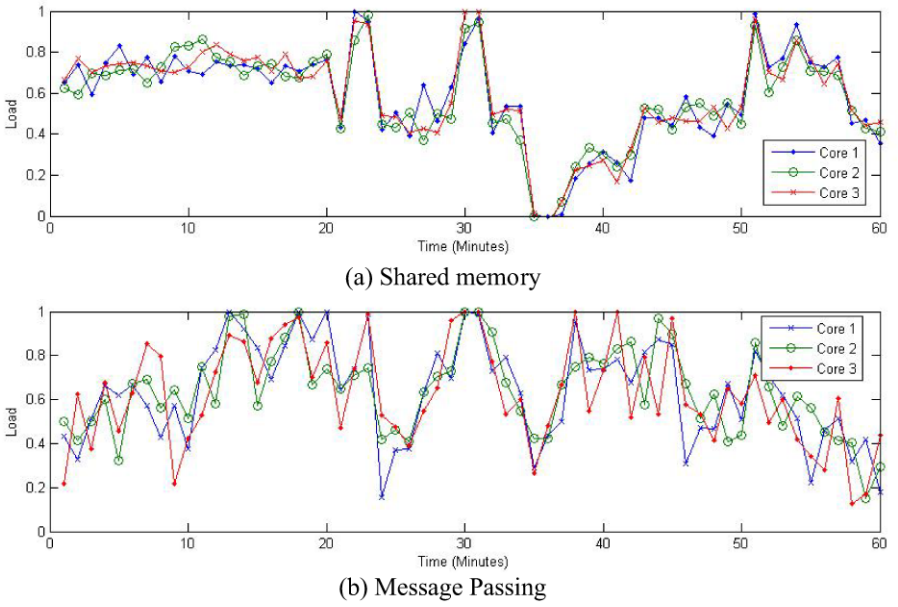


Fig. 19 Loads of each core in time

and Eddies algorithm have low performance (lightweight operators take less than 1 μs in average to handle a tuple).

In Fig. 19 the CPU utilization of 3 cores in a run is shown.

The above diagram in Fig. 19 shows the load of cores when DTR-SM runs. The bottom diagram shows the loads of 3 cores when DTR-MP is used. As it's shown in above diagram, when no heavy operators are in the query set, cores are free of load and even when they are under loads, they balance the load perfectly between each other. But in the bottom diagram, messaging takes most of the time of threads waiting for the reply to their requests, this way the routing algorithm won't commit their tasks completely; hence, the load between cores become imbalance.

7 Related work

Various scheduling strategies have been proposed for stream processing, ranging from simple (e.g., Round Robin [3], Chain [20], Greedy [3]) to more complex ones [21, 22]. Some address operator scheduling to optimize a single performance metric [18, 23] such as memory usage [3, 24]. Some others aim to optimize a complex performance metric [25, 26]. Generally, static scheduling strategies are weak in handling unpredictable, aperiodic and bursty workload [24].

For an overview on query processing and most related work on parallel query processing in databases, we refer the readers to [27–31].

In [1], parallel processing of continuous queries over logical machines is proposed which outperform serial execution as well as the min-latency scheduling [24] used in

the Aurora [32] to minimize tuple latency. Scheduling method employed in [1] is dynamic but event-driven (in overload situation). With respect to the continuous nature of the queries as well as data streams, the need for compatibility with this nature and adaptivity with time varying characteristics of data streams seems to be very important. A variant of such high-frequency (short-term) operator scheduling schema called *Dispatching* is proposed in [2] which provides a significant improvement on system's performance.

In the *Eddy* architecture [16], the query plan is replaced with an operator called Eddy which dynamically determines how each data tuple should be routed among operators of the eliminated query plan. In order to handle the drawback of this architecture (e.g., single source of failure, bottleneck, overheads and etc. [33]), a distributed version of Eddies was proposed in [34]; although the drawbacks are still considerable, especially for employing in a distributed system. The Eddy architecture, as a similar method, is compared with the proposed DTR method.

Dynamic operator placement in distributed query processing systems is discussed in [35, 36]. In [36], due to distribution of system over a local network, load balancing w.r.t. reducing communication costs is focused. Similar to the flux, [37] partitions data streams according to the query, but it is not performed dynamically.

In [38], different routing strategies are compared and the *WSCO* (Weighted Selectivity-Cost-Qlength) that considers all of selectivity, execution cost and queue's length of operator is concluded to be the best. However, in routing strategy employed in our *dispatching* method, updating cost of edges to select the minimum one is based on the destination machines' workload. Since the destination machine must process all of tuples waiting in its operators input queues, selectivity of operators is not taken into account.

In contrast with our reactive failure handling approach, a predictive failure management approach for data stream processing systems is proposed in [39]. Online learning methods are used to continuously classify operators at run time. Such approaches impose considerable overheads to the underlying system during its normal operation. So, these are not desirable for parallel systems in which failure can be tolerated by redundancies, such as in our proposed *UAPC–MIN*.

A topological classification of Multistage Interconnection Networks (MINs) and performance evaluation metrics of MINs is provided in [8].

8 Conclusion and future work

Online and fast query processing is of the major requirements and challenges in data stream systems. Single processor DSMSs are not capable of providing the required speed [1, 37]. We proposed parallel processing of queries in data stream systems over a multiprocessing environment in [1]. Operator scheduling in [1] is equal to finding the shortest path (routing) in the *QMG*; that is performed in an event-driven manner using the Dijkstra's shortest path algorithm. To be event-driven causes system performance to have too many fluctuations.

In order to reduce performance fluctuations, scheduling (routing in *QMG*) can be performed with higher frequency e.g., continuously and dynamically for each input tuple.

In this paper, Dynamic Tuple Routing (*DTR*) method is presented as a solution. Each processing unit (node of the *QMG*), after executing its assigned query operator, selects its next proper (min-cost) processing unit to forward the processed tuple to. Tuple latency—as the selection metric—is computed w.r.t. the cost model that contains processing, buffering and communication time delay.

Architecture of desired parallel system as the multiprocessing environment is analyzed and proposed. The interconnection network is a class of the MINs [5, 15], called *UACP–MIN*, which is based on the *QMG*.

Due to the following properties of the proposed multiprocessing environment, a priory-fixed parallel system topology is not a good choice for implementation:

- Number of processing units (i.e., nodes in the *QMG*) depends on the semantic of query and may vary per each one (*not-fixed*).
- All the processing units would not be employed all the time (*under-utilization*).

So, a hybrid parallel processing model is preferred to be used for implementing the practical parallel underlying system; in such system, each of $\boldsymbol{k}$ logical machines are considered as a physical (hardware) processor whilst each of nodes in the QMG implemented as a thread in the corresponding processor. Implementation of such system on multi-core systems is provided and used for evaluation. Performance evaluation results show that the proposed DTR method outperforms the event-driven scheduling [1] and Eddy [16] in terms of tuple latency, throughput and tuple loss. As expected, pure message passing (nothing is shared) communication model has a considerable overhead; So, employing a hybrid communication model would be very beneficial. For example, if we can use some shared memories e.g., for storing cost of each of downstream processing units, a large amount of messaging overheads will be removed. In a multi-core system, performing such enhancements is very common while not complicated. This is a part of our future works. Some of the others are listed below:

- Intra-operator parallelism for heavy operators especially the Join.
- Context-switching and message passing overheads reduction.
- Analyzing and implementing distributed version of a desired parallel system.

References

1. Safaei, A.A., Haghjoo, M.S.: Parallel processing of data stream query operators. Distrib. Parallel Databases **282**, 93–118 (2010). doi:10.1007/s10619-010-7066-3
2. Safaei, Ali A., Haghjoo, Mostafa S.: Dispatching stream operators in parallel execution of continuous queries. J. Supercomput. (2011). doi:10.1007/s11227-011-0621-5
3. Babcock, Brian, et al.: Operator scheduling in data stream systems. VLDB J. **13**, 333–353 (2004)
4. Replicate and migrate objects in the runtime, not cache lines or pages in hardware (Invited Plenary Lecture). In: Barcelona Multicore Workshop 2010, Barcelona, Spain, 21–22 Oct. (2010)
5. El-Rewini, H., Abd-El-Barr, M.: Advanced Computer Architecture and Parallel Processing. Wiley, Hoboken (2005). doi:10.1002/0471478385.index
6. Feng, T.Y.: A survey of interconnection networks. Computer **14**, 12–27 (1981)
7. Singah, B.: On multistage interconnection network. M.Sc. thesis (2000)
8. Aljundi, C., Chadi, A., Jundi, A., Dekeyser, J.-l., Scherson, I.D.: An interconnection networks comparative performance evaluation methodology: the case of delta and over-sized delta multistage interconnection networks. In: Proc. of the 16th International Conference on Parallel and Distributed Computing Systems (2003)

9. Lawrie, D.H.: Access and alignment of data in an array processor. IEEE Trans. Comput. **C-24**, 1145–1155 (1975)
10. Thomas, R.H.: Behavior of butterfly parallel processor in the presence of memory hot spots. In: Proc. of the 1986 Int. Conf. Parallel Processing, pp. 46–50 (1986)
11. Lin, W., et al.: A conflict routing scheme on multistage interconnection networks. IEEE Trans. Comput. **38**(8), 1086–1097 (1989)
12. Tian, H., Katangur, A.K., Yipan, J.Z.: A novel multistage network architecture with multicast and broadcast capability. J. Supercomput. **35**, 277–300 (2006)
13. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proc. of the 6th OSDI Symp. (2004)
14. Upadhyaya, P., Kwon, Y., Latency, A., Balazinska, M.: Fault-tolerance optimizer for online parallel query plans. In: Proceedings of the ACM SIGMOD (2011)
15. Grama, A., Karypis, G., Kumar, V., Gupta, A.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley, Reading (2003)
16. Avnur, R., Hellerstein, J.M.: Eddies: continuously adaptive query processing. In: Proceedings of the ACM SIGMOD (2000)
17. The Internet traffic archive, http://ita.ee.lbl.gov/html/contrib/DEC-PKT.html
18. Chakravarthy, S., Pajjuri, V.: Scheduling strategies and their evaluation in a data stream management system. In: Lecture Notes in Computer Science, vol. 4042. Springer, Berlin (2006)
19. LeBlanc, T.J.: Shared memory versus message passing in a tightly coupled multiprocessor: a case study. In: Proc. 1986 Int. Conf. Parallel Processing, pp. 463–466 (1986)
20. Babcock, B., et al.: Chain: operator scheduling for memory minimization in data stream systems. In: Proceedings of the ACM SIGMOD International Conference (2003)
21. Sharaf, M.A.: Preemptive rate-based operator scheduling in a data stream management system. In: IEEE/AICCSA (2005)
22. Soliman, M.S., Tan, G.: Operator-scheduling using dynamic chain for continuous-query processing. In: IEEE Int. Conference on Computer Science and Software Engineering (2008)
23. Sharaf, M.A., et al.: Scheduling continuous queries in data stream management systems. In: PVLDB (2008)
24. Don Carney, et al.: Operator scheduling in a data stream manager. In: Proceedings of the 29th International Conference on Very Large Data Bases, Germany, pp. 838–849 (2003)
25. Ghalambor, M., Safaeei, Ali A., Azgomi, M.A.: DSMS scheduling regarding complex QoS metrics. In: IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), 10–13 May (2009)
26. Srivastava B., Widom: exploiting k-constraints to reduce memory overhead in continuous queries over data streams. Technical report, November 2002
27. Graefe, G., et al.: Extensible query optimization and parallel execution in volcano. In: Query Processing for Advanced Database Systems. Morgan Kaufman, San Mateo (1994)
28. DeWitt, D.J., Gray, J.: Parallel database systems: the future of high performance database processing. Commun. ACM **36**(6), 85–98 (1992)
29. Graefe, G.: Volcano—an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng. **6**(1), 120–135 (1994)
30. Apers, P.M.G., et al.: PRISMA/DB: a parallel, main memory relational DBMS. IEEE Trans. Knowl. Data Eng. **4**(6), 541–554 (1992)
31. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**, 73–170 (1993)
32. Abadi, D., et al.: Aurora: a new model and architecture for data stream management. VLDB J. **2**, 120–139 (2003)
33. Deshpande, A.: An initial study of overheads of eddies. SIGMOD Rec. **33**, 44–49 (2004)
34. Tian, F., DeWitt, D.J.: Tuple routing strategies for distributed eddies. In: Proceedings of the 29th VLDB (2000)
35. Osman, A., Ammar, H.: Dynamic load management for distributed continuous query systems. In: Proceedings of the ICDE (2005)
36. Zhou, Y., et al.: Efficient dynamic operator placement in a locally distributed continuous query system. In: Lecture Notes in Computer Science, vol. 4275 (2006)
37. Johnson, T., et al.: Query-aware partitioning for monitoring massive network data streams. In: Proceedings of the ACM SIGMOD (2008)
38. Tian, F., DeWitt, D.J.: Tuple routing strategies for distributed eddies. In: Proceedings of 29th VLDB Conference, September 2003, pp. 333–344 (2003) (ISBN 0-12-722442-4)

39. Gu, X., et al.: Online failure forecast for fault-tolerant data stream processing. In: Proceeding of ICDE (2008)
40. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall International Series in Computer Science. Prentice-Hall, New York (1996). ISBN: 0-13-948472-8
41. Babu, S.: Adaptive query processing in data stream management systems. Ph.D. thesis, Stanford University (2005)
42. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive ordering of pipelined stream filters. In: Proc. SIGMOD Conference, pp. 407–418 (2004)
43. Das, A., Gehrke, J., Riedewald, M.: Approximate join processing over data streams. In: Proc. SIGMOD Conference, pp. 40–51 (2003)