

Modelling the Dynamic Evolution of System Workload During Pipelined Query Execution

Myra Spiliopoulou
Institut für Wirtschaftsinformatik
Humboldt-Universität zu Berlin
myra@wiwi.hu-berlin.de

Johann Christoph Freytag
Institut für Informatik
Humboldt-Universität zu Berlin
freytag@informatik.hu-berlin.de

Abstract

Database parallelism increases the complexity of query optimization. In particular, the query optimizer must take into account the impact of processor and network workload on competing and communicating processes. During query execution this additional overhead becomes even more important when exploiting pipelined parallelism, since the workload evolves over time dynamically as processes start and complete execution. We propose a model which incorporates the effect of the system workload on the execution of parallel and pipelined queries.

We extend the conventional notion of pipeline in two directions: First, we include pipes with multiple producer nodes into our model. Second, we consider the latency between the start time of the producing process and the start time of the consuming process. Based on this execution model, we derive a cost model that incorporates system workload. In particular, we calculate the effects of pipelining on processor multitasking and the effects of multitasking on the total execution time of the query. Additionally, we model the load on the network channels used to connect communicating processors, taking the fact into account that those channels may be changed dynamically by the routing mechanism. We consider the communication delays caused by this load and estimate their impact on pipelined execution.

Finally, the model integrates the cost of remote disk accesses to the computation of system workload. It is general enough to cover query processing for both shared-nothing and shared-disk architectures.

Keywords: query cost modelling, multi-producer pipelines, multitasking overhead, communication overhead, resource utilization, pipelined parallelism, query optimization, parallel databases

1 Introduction

Database parallelism offers the potential of efficient query execution, but it increases the complexity of query optimization. In particular, the impact of the workload of processors and network on competing and communicating processes must be taken into account. Both kinds of workload become even more important when exploiting pipelined parallelism, especially if the pipeline has multiple non-blocking producers. We introduce a query execution model that covers such pipelines. Based on this model, we derive a set of cost functions that incorporate the impact of system workload on pipelined query execution. Our model is general enough to cover bushy and pipelined parallelism for both shared-nothing and shared-disk architectures.

One of the most difficult issues in database parallelism is the efficient exploitation of pipelining, because it “does not easily lend itself to load balancing”, as pointed out by Graefe [Gra93]. Different aspects of pipeline exploitation on various query tree structures are studied in [Sch90, GHK92, LVZ93, SYT93, ZZBS93] and others. However, in all those models, the notion of pipeline is rather restricted, since, even for bushy trees, at most one of the children of a node is assumed to produce output in pipeline mode. While this is acceptable for some algorithms implementing relational algebra operators, it is not true for object-oriented methods with multiple inputs. Moreover, the delay between the start time of a node and the start time of its parent in a pipeline is almost uniformly ignored.

Hasan and Motwani [HM94, HM95] study the problem of pipelined parallelism from the viewpoint of communication tradeoff. They propose heuristics to minimize communication overhead taking multitasking into account [HM94], and a method of specifying the operator to processors mapping [HM95]. However, they do not study the impact of the network configuration. This configuration affects the selection of physical channels to connect two communicating processors, as well as the load on the channels. A router may even dynamically choose different routes between the same processors during query execution. Hence, the workload of a channel varies dynamically and differs from channel to channel.

If a cost function simply computes the cost of a query execution plan that is later parallelized [Hon92] or the cost of parallelizing it [HM94], then the effect of processor allocation on the quality of the original plan is overseen. Cost functions computing the processor utilization in pipelined execution are proposed in [GHK92, LVZ93, SYT93] for various tree types. However, the notion of pipelining has the aforementioned restrictions. Further, the impact of network usage is not addressed, and the transfer time unit between two arbitrary processors is assumed to be a constant.

We propose a model for pipelined execution that alleviates the above shortcomings. First, we generalize the conventional binary bushy tree structure into an n-ary query tree. A node of this tree can receive output in pipelined mode from any of its children. Second, we take into account the delay between the start times of adjacent nodes in a pipeline. Third, we incorporate both processor and network utilization into the cost of this generalized query tree. Forth, rather than simply estimating communication cost for an abstract network overlooking its configuration, we model the varying load on each network channel throughout query execution and calculate its impact on communication cost. The parallel architectures we cover may conform to both the shared-nothing and the shared-disk paradigms.

By incorporating more parameters of the optimizer’s search space into the cost function, the approximation of the actual execution time of a query schedule becomes more realistic. However, the new parameters expressing the influence of latency, multitasking and network load on query cost, increase the search space to be scanned by the optimizer. We therefore envisage optimization techniques based on randomized algorithms [IK91, LVZ93, SHC95] or genetic algorithms [MMS94]

for the exploration of the search space of our model.

In the next section, we introduce our base terminology, specify the problem of schedule cost computation and present our schedule execution model. In section 4, we present a mechanism computing the execution rate of multi-producer pipelines. This mechanism is essential, because it allows the accurate monitoring of resource utilization, without making the simplifying assumption of zero latency. In section 3, we compute the cost of a process incorporating the synchronization adjustments and taking the latency among pipeline participants into account.

Section 5 analyzes our cost mechanism, which computes the cost of a schedule by trapping the changes in the processor utilization as “events” and calculating the time of their occurrence. In section 6, we extend the base mechanism to also incorporate the network utilization on each channel. The modelling of remote disk accesses for temporary data storage differs for the shared-nothing and the shared-disk architecture. For the former, a simple extension of the mechanism is described in section 6. The modelling of remote I/O for the shared-disk architecture is more complex and is handled separately in section 7. Section 8 concludes our study.

2 Modelling the Query Execution Problem

There are two optimization approaches for the parallel query execution problem. The two-phase approach produces an optimized uniprocessor query execution plan using some cost function and then parallelizes it using another cost function [Hon92, HM94]. The single-phase approach incorporates information on the operator(s)-to-processor(s) mapping into the cost function and produces a complete schedule for the original query [GHK92, LVZ93, SYT93]. Those two approaches are shown in Fig. 1(a) and (b) respectively. In our own single-phase approach, shown in Fig. 1(c), we incorporate both the operator-to-processor mapping and the dynamically evolving load of processors and network channels into the cost function.

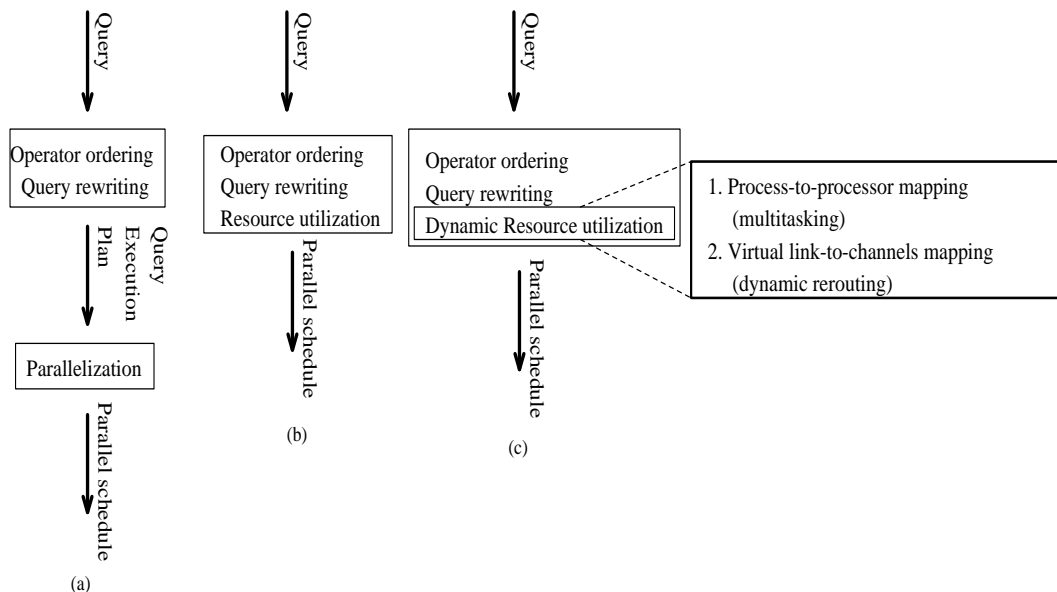


Figure 1: Query optimization and cost estimation methods for pipelined execution

2.1 Query Execution Plans with Multi-Producer Pipelines

Query Execution Plans. A query execution plan (QEP) is a query tree, whose nodes are query operators annotated with the execution algorithm and data access information. The edges connecting the nodes reflect the dataflow from the leaf nodes towards the root. The children of a node x are its “producers”; x is called “consumer”. According to the terminology of [HM94], a producer may be “blocking”, if it must complete execution before its consumer starts, or “pipelining”, if each tuple it outputs is immediately forwarded to the consumer. We term an edge from a pipelining producer as a “pipelining edge”, and an edge from a blocking producer as a “blocking edge”.

Conventional and multi-producer pipelines. Let x, y, z be three operators, with x consuming the output of y and z . On a bushy QEP, four cases may occur, as depicted in Fig. 2, where we denote a pipelining edge by an arrow pointing to the consumer.

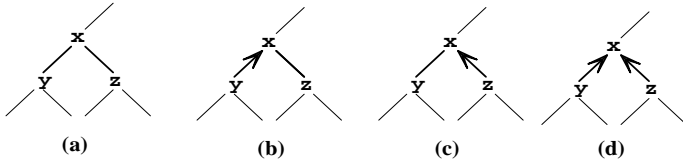


Figure 2: Three operators in a bushy QEP

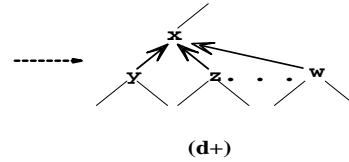


Figure 3: A Multi-Producer Pipeline

Case (a) has no pipeline. Cases (b) and (c) depict a switching of the inner relation input to x . If we allow that a pipelining edge may be the left or the right one, those cases are equivalent¹. Case (d) is the more general one: x receives input in pipeline mode from two producers.

In Fig.3, we extend case (d) by considering an n-ary bushy tree and an operator x with more than two producers y, z, \dots, w , any of which may operate in pipeline mode towards x . We term this pipeline a “multi-producer” pipeline. Process x could be a multi-input method that cannot be decomposed by the optimizer. Thus, our model is not limited to relational queries, since queries based on an object-oriented data model can also be handled.

2.2 Scheduling a QEP

Schedules. A schedule represents a node-to-processor mapping for a QEP. We do not consider node parallelism. Hence, a node is assigned to exactly one processor.

The nodes of the QEP correspond to “processes” in the schedule. According to the above terminology for producer and consumer nodes, a schedule has producer and consumer processes, connected with pipelining or blocking edges. For the rest of the paper, we consider pipelining only.

Example schedule. In Fig. 4(a) we show the query graph of a query having predicates with at least two operands. The edge x_2 indicates an operator with three operands. In Fig. 4(b), we present the configuration of the network in which this query is executed. For this query and configuration, a possible QEP and schedule are presented in Fig. 4(c), where the join ordering and the process-to-processor-mapping have been specified. We omit the execution algorithms, as well as the operators reading the input relations for brevity.

In the representation of the schedule, the horizontal axis is a time axis. We also depict the latency between the start time of a process and the start time of its consumer. The notion of latency is discussed in more detail below.

¹In a right-deep tree, only the right edge permits pipelined processing.

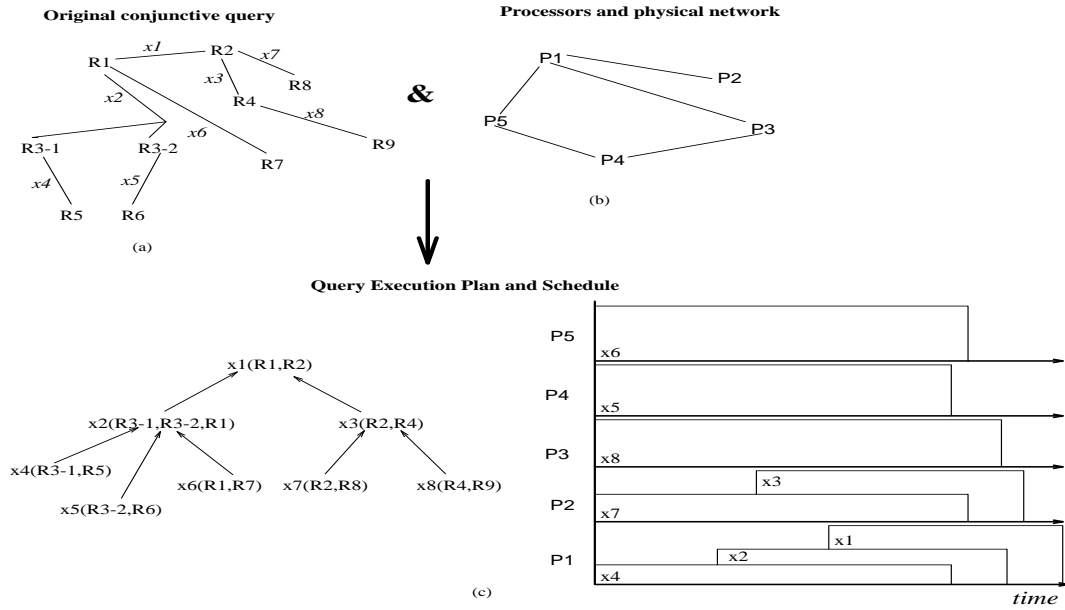


Figure 4: An example QEP and schedule for a conjunctive query on a network

2.3 Execution Model

Query execution starts by activating the “runnable” processes of the schedule. A process is runnable if it has enough input data to operate upon. If the process participates in a pipeline, it can start as soon as it has received the first data from its producers.

Latency time in pipelined execution. In a pipeline, producers and consumers run simultaneously. It is often assumed that all processes in a pipeline also start simultaneously [HM94]. However, there is always a “latency”, i.e. a delay between the start time of a process and the start of its consumer. This latency is due to local processing on the producer node, depending on the producer’s execution algorithm and on the predicate’s selectivity. Latency is also caused by transmission delays across the network, and by data buffering, if buffering is used for data transfers. Hence, the processes in a schedule do not run from the beginning to the end of query execution. Therefore, the system workload varies over time, as processes become runnable and as they complete execution.

A dynamically varying workload. At the beginning of query execution, the runnable processes are leaf nodes of the QEP, so they are executed without communicating. Processes running on the same processor compete for its resources. Assuming that the processor distributes its resources evenly among them, the execution time of a process is proportional to the number of processes running on the processor.

When non-leaf processes receive enough data to start execution, they are activated on the processors specified by the schedule. Their activation implies redistribution of processor resources. The activation of pipeline consumers also implies data transfers across the network, from the producers towards their consumer. The network channels used for this transfer may be changed dynamically during query execution by the system router, in order to exploit the full network bandwidth. The workload on the network channels affects the data transfer time in a similar way

as multitasking affects the process execution time.

Synchronization of processes in a pipeline. As pointed out in [HM94], a pipeline executes at the pace of its slowest process. However, due to latency, a process affects the execution pace of the pipeline only when it becomes runnable. Therefore, an explicit synchronization mechanism is needed, which adjusts the execution rates of pipelined processes on the fly.

We present such a dynamic synchronization mechanism in [SF95]. In summary, this mechanism computes the relative execution rates of all producers y_1, \dots, y_k of a node x , and identifies the slowest producer, say y_1 . Depending on whether the consumer or the producers must be slowed down, two adjustment functions are specified: $adjustConsumer(x, y_1)$ and $adjustProducer(x, y_i)$ for each producer y_i . These functions reduce the execution speed of the consumer, resp. producer, by a fraction in the range $(0, 1]$, where the value 1 means that the two processes do not need to be adjusted.

A query optimizer may decide to replace a pipelining edge with a blocking edge, if one of the producers is very slow. This case falls beyond the scope of our study, since we focus on pipelining edges only.

2.4 The Cost Computation Problem

The computation of the cost of a schedule in the aforementioned execution model can be specified as follows:

Input: a schedule with its accompanying QEP

Meta-information: the network configuration, consisting of possibly heterogeneous processors with or without access to local discs

Output: the total elapsed time from the beginning of executing the first runnable process until the final output is generated

In order to compare different schedules, an optimizer may need a special metric, since two schedules may differ both on the QEP structure and on the number of processors. The specification of the metric depends on the optimization objectives. The reader is referred to [GHK92, LVZ93, SYT93] for the description of such metrics.

In order to compute the cost of a schedule, we simulate its execution on the parallel machine, monitoring the changes in the processor and network channel utilization. Our model captures the parameters of the aforementioned execution mechanism. In the next section we consider the impact of latency and multitasking on the execution cost of a process in a pipeline. In section 5, we study the influence of processors' workload on the overall schedule. The effects of network workload are presented in section 6.

3 Process Cost in a Multi-Producer Pipeline

The cost of a schedule is the sum of the execution time of the root process and of the elapsed time until its slowest producer has generated enough output for the root to start. We apply this computation method recursively until reaching the leaf processes.

3.1 Cost of an Individual Process

We observe the cost function $C(\cdot)$ of a particular algorithm as a black box. An in-depth analysis of execution algorithms and their cost can be found in [Gra93]. We assume that $C(\cdot)$ incorporates the characteristics of the algorithm, including CPU time and, for the shared-nothing architecture, access to local discs. The cost of remote I/O access is studied separately, as described in subsection 6.2.

The cost of a process x is affected by multitasking. In particular, x shares the processor's resources with $n_x - 1$ other processes active at the same time. So, we denote the cost of x as $C(x, n_x)$. We assume that the processor distributes its services (CPU time slices and I/O request handling) fairly among the processes running on it ². Therefore:

$$C(x, n_x) = C(x, 1) \cdot n_x + ps$$

where ps is the overhead of process switching. This overhead should rather be a function on n_x . However, we assume that it is very small relatively to the value of $C(\cdot)$, so that an average constant value can be assigned to it, in order to simplify the formula.

If we have already computed the cost value $C(x, n_x)$, we can calculate the cost of executing x together with $n'_x - 1$ other processes as follows:

$$C(x, n'_x) = C(x, n_x) \cdot \frac{n'_x}{n_x} + ps \quad (1)$$

It should be stressed that n_x varies over time as processes start and complete execution. Hence, one of the objectives of our model is to reflect the evolution of n_x for each process x and its impact on the execution time of x . The formula in Eq.1 will be used to calculate the (remaining) execution time of a process whenever the number of processes running on a processor changes.

3.2 Cost of a Process in a Pipe

Let x be a process in the schedule. If x is a leaf process, it is runnable from the beginning of query execution. Otherwise, it must wait for its producers to output sufficient data for it to start. Let $C^{init}(x, n_x)$ be the "initialization time" of a process, i.e. the time it needs to produce the initial output needed by its consumer to start. C^{init} reflects the impact of latency, as described in subsection 2.3. For the root node, which has no consumer, we set C^{init} equal to zero.

We denote by $T^{init}(x, n_x)$ the elapsed time from query start to the end of the time span $C^{init}(x, n_x)$, by which x becomes runnable. If x is a leaf process, then $T^{init}(x, n_x)$ is equal to $C^{init}(x, n_x)$. Otherwise, let Y be the set of producers of x , and let y_1 be its slowest producer, i.e. the producer for which it holds that: $T^{init}(y_1, n_{y_1}) = \max_{y \in Y}(T^{init}(y, n_y))$. Then:

$$T^{init}(x, n_x) = \begin{cases} C^{init}(x, n_x) & , x \text{ is a leaf node} \\ T^{init}(y_1, n_{y_1}) + \mathit{adjustConsumer}(x, y_1) \cdot C^{init}(x, n_x) & , \text{otherwise} \end{cases} \quad (2)$$

If x is too fast for its producers, it is slowed down by the adjustment function $\mathit{adjustConsumer}(\cdot)$ in Eq.6.

The total execution time of x is $C(x, n_x)$, as described in subsection 3.1. We denote by $C^{end}(\cdot)$ the remaining execution time for x after $C^{init}(x, n_x)$. If the consumer z of x runs on the same

²Distribution of memory is more complex because it depends on the demand of each process. We are considering this problem but beyond the scope of this study.

processor p as x , then the resources of p must be redistributed among $n_x + 1$ processes. Hence, the “completion time”, i.e. the remaining execution time of x is, according to Eq.1:

$$C^{end}(x, n'_x) = \begin{cases} C(x, n_x) - C^{init}(x, n_x) & n'_x = n_x \quad , z \text{ does not run on } p \\ C(x, n_x) \cdot \frac{n'_x}{n_x} + ps - C^{init}(x, n_x) & n'_x = n_x + 1 \quad , z \text{ runs on } p \end{cases}$$

We denote the elapsed time from the beginning of query execution to the completion of x as $T^{end}(x, n'_x)$. We compute it as follows:

$$T^{end}(x, n'_x) = adjustProducer(z, x) \cdot C^{end}(x, n'_x) + T^{init}(x, n_x) \quad (3)$$

where the adjustment function $adjustProducer(\cdot)$ slows down x if it is faster than z ; otherwise it is equal to 1. This function is described in detail in [SF95].

3.3 Example

In Fig. 5, we consider a simple QEP scheduled on one processor. The cost of this schedule is the execution time of x and the elapsed time until it starts execution. This elapsed time is equal to the initialization time of y_1 , assuming that y_1 is slower than y_2 . Note that C^{init} and C^{end} are time spans, while T^{init} and T^{end} are points on the time axis, denoting the end of those spans.

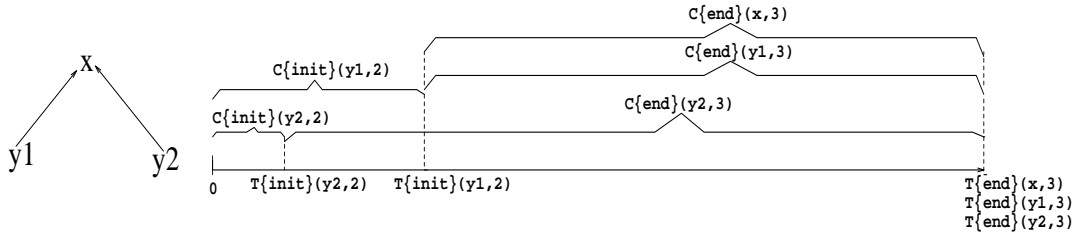


Figure 5: Execution of a pipeline with two non-blocking producers

We can see that the time span $C^{init}(y_1, 2)$ overlaps the time span $C^{init}(y_2, 2)$, because y_1 is slower. Since y_1 and y_2 are leaf processes, they are runnable at the beginning of query execution and the elapsed time $T^{init}(y_i, 2)$ coincides with $C^{init}(y_i, 2)$ for $i = 1, 2$. At $T^{init}(y_1, 2)$, x becomes runnable and the processor’s resources are redistributed among three processes. Ongoing execution of y_1, y_2 is overlapped by the execution of x . Since x has no consumer, its initialization time is zero.

In order to keep the example simple, we have assumed that the three processes finish together, so that the processor utilization does not change once x starts. In the general case, there will be a delay between the completion of y_1 and y_2 and that of x . This implies that the distribution of processor resources among the running processes will change again during the lifetime of x . The computation of the remaining execution time of a process, whenever processor utilization changes, is analyzed in the next section.

4 Synchronizing Processes in a Multi-Producer Pipeline

As pointed out in [HM94], a pipeline executes at the pace of its slowest process. However, as noted in subsection 2.3, there is a latency between the start time of a process and the start time of its

consumer. Therefore, an explicit synchronization mechanism is needed, which adjusts the execution rates of pipelined processes on the fly.

The synchronization effectively slows down processes. Hence, an optimizer may use our model to decide whether synchronization is still preferable to the alternative of breaking the pipeline by introducing a blocking edge. This case falls beyond the scope of our study.

4.1 Execution Rate of a Process

A process x produces output at a rate $outRate(x)$, which is determined by the operator semantics. x receives input from its producers at their different output rates. The total input rate of x , denoted by $producersRate(\cdot)$, is the summation of the output rates of its producers.

$$producersRate(x, y_1) = \sum_{i=1..k} outRate(y_i) \quad (4)$$

The relationship between the output rates of producers of the same consumer can be computed from the relationship between the cardinalities of the data sets they output and the selectivity of the consumer's operator towards each data set, assuming a uniform data distribution. For instance, if x is a join process equally selective towards both its input relations, and if one relation is twice as large as the other, then the input rate of the large relation must be half the input rate of the small one.

In a multi-producer pipe, we identify the slowest producer and we express the output rates of the other producers in terms of the output rate of that one. Let y_1, \dots, y_k be the producers of process x and let y_1 be the slowest one. There are a_2, \dots, a_k such that $outRate(y_i) = a_i \cdot outRate(y_1)$ for each $i = 2 \dots k$. Hence, we use Eq. 4 and express the producers' rate of x as a function of y_1 :

$$producersRate(x, y_1) = \sum_{i=1..k} a_i \cdot outRate(y_1) = a \cdot outRate(y_1) \quad (5)$$

where $a_1 = 1$ and $a = \sum_{i=1..k} a_i$.

Synchronization between x and its producers y_1, \dots, y_k means that the output rate of x must be equal to its producers' rate. We describe the synchronization mechanism through an example.

4.2 Example of Synchronization

Let y_1, y_2, y_3 be the producers of x , and w be the consumer of x , as shown in Fig. 6. Let y_1 be the slowest producer of x . The original values of the output rates, as determined by the semantics of the processes, are shown in $\langle Step1 \rangle$ of Fig. 6.

According to Eq.4 and Eq.5, $producersRate(x, y_1)$ is 18 rather than 20. So, when x starts execution in $\langle Step2 \rangle$, its output rate must be set to 18, to keep in pace with its producers. When w starts execution in $\langle Step3 \rangle$, the $producersRate(w, x)$ must be adjusted to the output rate of w , which is 8 rather than 18. Then, y_1, y_2, y_3 must be slowed down, too, because they are now faster than x . $\langle Step3 \rangle$ shows the output of the synchronization, when all nodes of the tree execute at the same rate.

4.3 Synchronization Functions

As shown in the example, synchronization is performed on the fly. If a runnable process is slower than its producers, it slows them down. This effect is propagated down to the leaf nodes. If the process is faster than its producers, then it slows down itself. We introduce two functions to

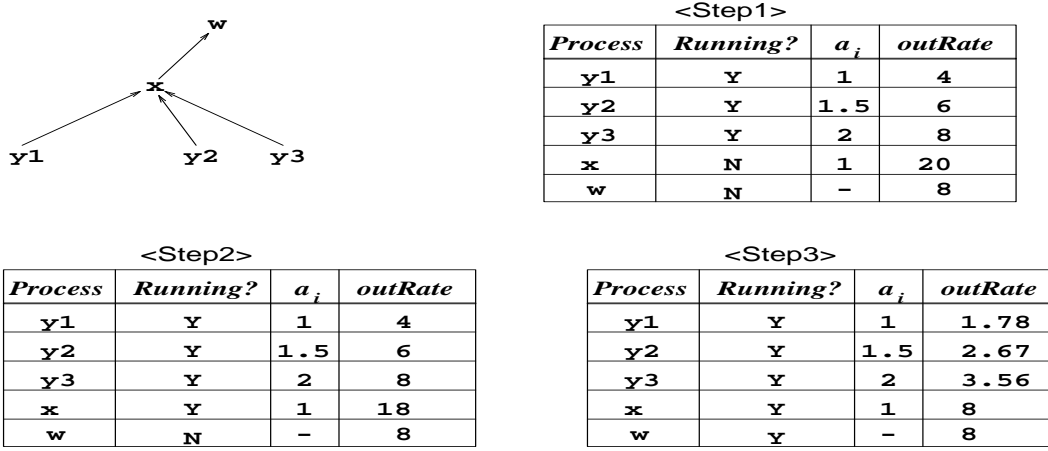


Figure 6: Output rate adjustment in a multi-producer pipeline

synchronize processes in a pipeline. The first one adjusts the consumer x to its slowest producer, say y_1 , if the consumer must be slowed down:

$$adjustConsumer(x, y_1) = \begin{cases} 1 & , outRate(x) \leq producersRate(x, y_1) \\ \frac{producersRate(x, y_1)}{outRate(x)} & , otherwise \end{cases} \quad (6)$$

The second function adjusts y_1 to x , if the producers must be slowed down:

$$adjustProducer(x, y_1) = \begin{cases} 1 & , outRate(x) \geq producersRate(x, y_1) \\ \frac{outRate(x)}{producersRate(x, y_1)} & , otherwise \end{cases}$$

After adjusting y_1 , the other producers y_2, \dots, y_k are adjusted using a_2, \dots, a_k . Therefore:

$$adjustProducer(x, y_i) = \begin{cases} 1 & , outRate(x) \geq producersRate(x, y_1) \\ a_i \cdot \frac{outRate(x)}{producersRate(x, y_1)} & , otherwise \end{cases} \quad (7)$$

for $i = 1 \dots k$.

4.4 Impact of Multitasking

The execution speed, and thus the output rate, of a process is not only affected by the speed of its producers and its consumer, but also by the number of runnable processes on the same processor. Hence, we extend our notation, so that the output rate of a process x coexisting with $n_x - 1$ other processes is $outRate(x, n_x)$. As in 3.1, we assume that a processor distributes services to these processes in a judicious way. Hence, the output rate of x is the $(\frac{1}{n_x})^{th}$ fragment of its rate when executed alone:

$$outRate(x, n_x) = \frac{outRate(x, 1)}{n_x}$$

A similar equation holds for each producer of x . The producers' rate becomes:

$$producersRate(x, y_1) = \sum_{i=1 \dots k} outRate(y_i, n_{y_i}) = \sum_{i=1 \dots k} a_i \cdot outRate(y_1, n_{y_1}) = a \cdot outRate(y_1, n_{y_1}) \quad (8)$$

This equation generalizes Eq.5. According to it, the output rates of all producers are estimated again, whenever the processor load changes. The a_i values are calculated anew and $producersRate(\cdot)$ is recomputed by Eq.8. Then, to adjust the pipeline in the presence of multitasking, we use the functions $adjustConsumer(\cdot)$ and $adjustProducer(\cdot)$ after replacing $outRate(x)$ by $outRate(x, n_x)$ in Eq.6 and Eq.7.

5 A Mechanism for Event Capturing

We compute the execution time of a schedule by simulating processing on each individual processor. Resource utilization changes as processes start and terminate execution on processors during evaluation of the query. We reflect these changes by an event capturing mechanism placing them on the time axis of each processor.

An “interesting event”, called “event” for short hereafter, is a change on resource utilization, occurring at some point of time on the processor’s time axis. Resource utilization changes when a process starts execution and thus competes for processor and network resources (“start event”), when it has produced enough data for its consumer to start and must transmit them to it (“initialization event”)³, and when it terminates and frees its resources (“completion event”). According to these definitions, the start event for a process coincides with the initialization event for its slowest producer.

The functions $T^{init}(\cdot)$ and $T^{end}(\cdot)$ defined in Eq.2 and Eq.3 of subsection3.2 mark the occurrence of the initialization event and the completion event respectively. In the following, we use the term “event” and the notion of the “time point at which the event occurs” interchangeably.

The load on a processor depends on the processes running on it. The load on a network channel depends on the processes using it for data transfer. By the above definition, utilization of these resources is constant between any two adjacent events. Thus, we measure the cost of a schedule by detecting the three aforementioned types of events and placing them on the time axis. This gives us a precise estimation of the execution time on each processor.

5.1 Events on one processor

We first describe the mechanism capturing the events occurring on a single processor, and then we discuss their interference with events on other processors.

Let S^i denote the set of processes assigned to processor i . The processor is active until the last of these processes has finished execution, i.e. for the time span $[0, t_{end}^i]$, where $t_{end}^i = \max_{x \in S^i}(T^{end}(x, n_x))$. This time span is divided into intervals by a series of points $0, t_1^i, t_2^i, \dots, t_{end}^i$, each one corresponding to some event.

Let A^i be the set of processes, for which the initialization event has not occurred yet, and let B^i be the set of processes running to completion after their initialization events. When the initialization events of all producers of a process have occurred, the process becomes runnable and is added to set A^i . When its initialization event occurs, the process migrates to set B^i . When its completion event occurs, the process is eliminated. Processor i completes execution when both A^i and B^i are empty. As A^i and B^i evolve over time, we denote the instance of A^i (B^i) at t_j^i as A_j^i (resp. B_j^i).

³The initialization event of a process will cause no change in resource utilization if the consumer process is located on the same processor and if this event does not coincide with its start event.

The first event. When the processor starts execution, it executes the processes in A^i . The first event that can occur is the initialization event of some process. This event is placed on the time axis as point t_1^i :

$$t_1^i = \min_{x \in A_1^i} (T^{init}(x, n_1))$$

where $n_1 = \text{card}(A_1^i)$ ⁴. Let x_1 be the process corresponding to this minimum.

At t_1^i , x_1 migrates from set A^i to B^i . If the consumer of x_1 , say y , has no other producer, then it can start execution, too. Then: $A_2^i = (A_1^i \cup \{y\}) - \{x_1\}$ and $B_2^i = B_1^i \cup \{x_1\}$.

General case. We want to estimate the time t_j^i at which the next event occurs. t_j^i corresponds to the earliest among the initialization events of the processes in A_j^i and the completion events of the processes in B_j^i :

$$t_j^i = \min \left(\min_{x \in A_j^i} (T^{init}(x, n_j)), \min_{x \in B_j^i} (T^{end}(x, n_j)) \right) \quad (9)$$

where $n_j = \text{card}(A_j^i) + \text{card}(B_j^i)$. Let x_j be the process corresponding to this minimum. Then:

1. If x_j belonged to A_j^i , the event indicates that x_j has completed its initialization phase. It must therefore migrate from A^i to B^i : $A_{j+1}^i = A_j^i - \{x_j\}$ and $B_{j+1}^i = B_j^i \cup \{x_j\}$.
If the consumer of x_j , say w , is runnable, i.e. if the initialization events of the siblings of x_j have already occurred, then w is added to A_{j+1}^i .
2. If x_j belonged to B_j^i , the event indicates that its execution is now completed. Then, it must be removed from B^i : $B_{j+1}^i = B_j^i - \{x_j\}$.

As already noted in subsection 3.2, the initialization event of the process at the root of the tree coincides with its start event. Hence, the migration of this process from set A_i to set B_i is instantaneous.

Impact of multitasking. The number of processes active after t_j^i is $n_{j+1} = \text{card}(A_{j+1}^i) + \text{card}(B_{j+1}^i)$. If $n_{j+1} \neq n_j$, then the processor must redistribute its resources. Let x be a process running after the event t_j^i . Then:

1. If t_j^i is the initialization event of x , the initialization time of x is $C^{init}(x, n_{j+1})$.
2. If the initialization event of x has not occurred before t_j^i , then its remaining initialization time is $T^{init}(x, n_j) - t_j^i$. This time must be adjusted according to Eq.1, into:

$$\underbrace{T^{init}(x, n_{j+1}) - t_j}_{\text{remaining initialization time}} = (T^{init}(x, n_j) - t_j^i) \cdot \frac{n_{j+1}}{n_j} + ps \quad (10)$$

For the computation of t_{j+1}^i , we need to compute the time point of the initialization event of x , given the new resource allocation. It is derived easily from the above formula as:

$$T^{init}(x, n_{j+1}) = (T^{init}(x, n_j) - t_j^i) \cdot \frac{n_{j+1}}{n_j} + ps + t_j$$

⁴ $\text{card}(\cdot)$ denotes set cardinality.

3. If t_j^i corresponds to the initialization event of x , its completion time is $C^{end}(x, n_{j+1})$.
4. If the initialization event of x has occurred before t_j^i , its remaining completion time is $T^{end}(x, n_j) - t_j^i$. This time must be adjusted into:

$$\underbrace{T^{end}(x, n_{j+1}) - t_j}_{\text{remaining completion time}} = (T^{end}(x, n_j) - t_j^i) \cdot \frac{n_{j+1}}{n_j} + ps \quad (11)$$

Similarly to the second case above, the completion event for x under the new resource allocation will occur at:

$$T^{end}(x, n_{j+1}) = (T^{end}(x, n_j) - t_j^i) \cdot \frac{n_{j+1}}{n_j} + ps + t_j$$

Example. In Fig. 7, we show the placement of events on the time axis for a processor executing a multi-producer pipeline, and the evolution of the sets A and B . We have omitted the index of the processor number for notational simplicity. The slowest producer is y_1 , so that it holds: $outRate(x, 5) = (1 + a_2 + a_3 + a_4) \cdot outRate(y_1, 5)$. When x starts execution, the execution rates of all 5 processes are adjusted, if necessary, by setting the output rate of the consumer x equal to the producer's rate $producersRate(x, y_1)$, given by Eq.8.

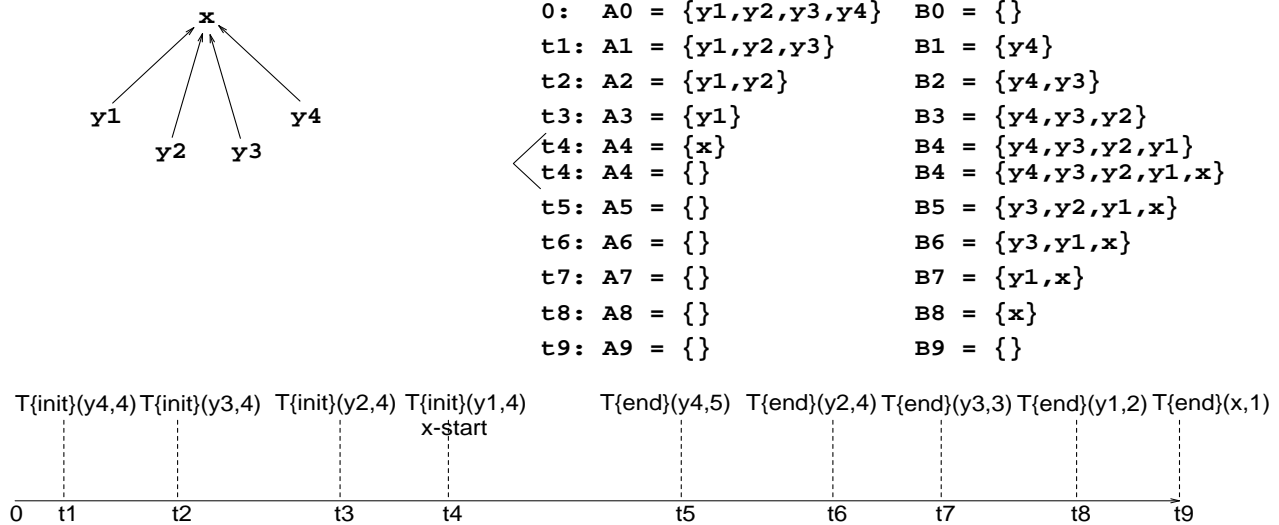


Figure 7: Execution of a multi-producer pipeline on one processor

At the beginning of query execution, only the four leaf processes are runnable. When the initialization events of processes y_4, y_3, y_2 occur, the resource utilization does not change because x is still waiting for data from process y_1 . At t_4 , the initialization event of y_1 occurs. Thereafter, all five processes are running on the processor.

At t_4 , the remaining execution time of $y_i, i = 2, 3, 4$, is adjusted to $T\{end\}(y_i, 5) - t_4$ according to Eq.11, while the completion time of y_1 is $C\{end\}(y_1, 5)$. As process x is the root, its initialization event coincides with its start event. We depict this fact in the Figure by repeating point t_4 , at which x instantaneously migrates from set A_4 to set B_4 . The completion time of x is initially $C\{end\}(x, 5)$.

Points t_5, \dots, t_9 mark the completion events of the processes. After each such event, the number of running processes is reduced by one and the remaining completion times of the processes still active are readjusted.

5.2 Interleaving Events on Different Processors

If a process is assigned to another processor than its consumer, then the producer's initialization event must be "projected" onto the time axis of the other processor, because data must be transferred from the producer to the consumer. This data transfer affects the network load. We initially describe this projection mechanism without considering the delay caused by data transfer. In the next section, we generalize our formulae to incorporate this delay.

Each processor has its individual clock. The interference of process execution, though, forces us to place the events occurring on different processors in total order on a common time axis. We introduce therefore a virtual "reference time axis" on which we place the events of all p processors. The execution time span for the query on the reference axis is $[0, t_{end}]$, where $t_{end} = \max_{i=1\dots p}(t_{end}^i)$. At point 0 (beginning of time axis), we assume without loss of generality, that *all* processors start processing the query.

The first and second event. The first event placed on the reference time axis is the earliest event among all processors:

$$t_1 = \min_{i=1,\dots,p} (t_1^i)$$

Let x^k be the process corresponding to this minimum, where k is the processor on which x^k runs. Further, let w^l be the consumer of x^k , where $l \neq k$. x^k must send data to w^l , thus affecting the network load. Hence, t_1 must be projected onto the time axis of processor l .

The first event placed on the time axis of l at point t_1^l occurs after t_1 , by definition of the latter. So, t_1 becomes the new t_1^l and the counter of events on l is advanced. If t_1 is also the start event of w^l , then the remaining initialization times of the running processes must be recalculated according to Eq.10, to take into account the resource demand of w^l . Then, the next event t_2^l is computed according to Eq.9. If w^l does not start execution, then the old t_1^l is the next event, renumbered as t_2^l .

Since t_1^k and t_1^l appear already on the reference time axis, the next event to be placed as point t_2 is:

$$t_2 = \min(\min_{i=1\dots p, i \neq k, i \neq l} (t_1^i), t_2^k, t_2^l)$$

General case. We compute the time t_i of the next event, by identifying the earliest event that is not already placed on the reference time axis.

$$t_j = \min_{i=1,\dots,p} (t_{j_i}^i) \tag{12}$$

where we denote as $t_{j_i}^i$ the time of the earliest event on processor i , which is not already placed on the reference time axis. We use the double index j_i , because the event counters on the different processors are not advanced at the same pace. This is already clear for the second event on the reference axis.

Let $t_{j_k}^k$ be the event corresponding to t_j . If $t_{j_k}^k$ is the initialization event of a process whose consumer is located on a different processor l , then $t_{j_l}^l$ becomes the time of this initialization event, i.e. it becomes equal to t_j , and the counter j_l is advanced. If this initialization event caused a

process to start, then the time point of the next event on l must be recomputed according to Eq.10 and Eq.11. Otherwise it only needs to be renumbered, because the event counter has been advanced.

Example. In Fig. 8, we show a QEP and schedule on two processors $P1$ and $P2$. Each processor has its own time axis, and the events on it are projected onto the reference axis. We denote by $x1_init$ the event occurring at $T^{init}(x_1, 3)$, and similarly for the other events. In the enumeration of events, $t\{i\}j$ stands for t_j^i .

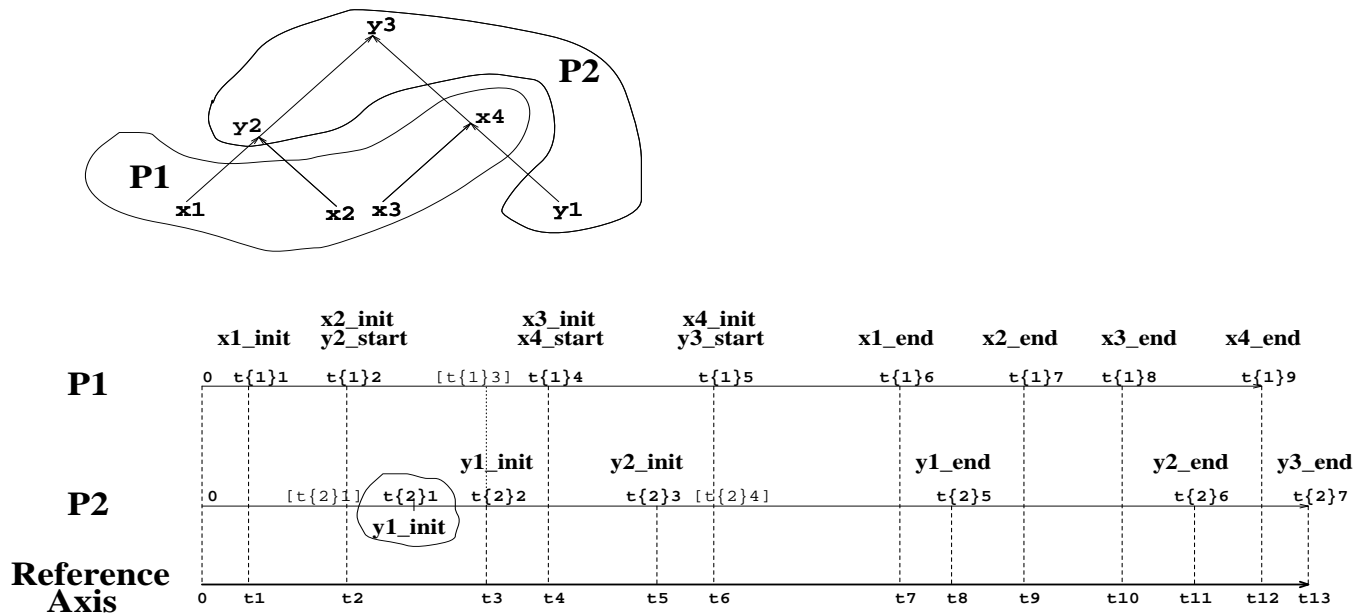


Figure 8: Execution of a bushy QEP on two processors

Initially, processes x_1, x_2, x_3 run on $P1$ and y_1 runs on $P2$. The first event occurs at $t\{1\}1$, so that t_1 is set equal to $t\{1\}1$. The second event occurs at $t\{1\}2$. It is projected onto the time axis of $P2$, as $[t\{2\}1]$, where the square brackets denote the projection operation. This event is also the start event of y_2 . Hence, the resources of $P2$ must be redistributed among two running processes. The original event that occurred at $t\{2\}1$ is placed in a circle, because it was computed without considering y_2 : it is renumbered into $t\{2\}2$ and its value is recomputed.

Event t_3 corresponds to $t\{2\}2$. Since y_1 is one of the producers of x_4 , t_3 is projected onto the time axis of processor $P1$. This event simply causes the renumbering of subsequent events on $P1$ but no recomputation because x_4 cannot start yet. Subsequent events are computed similarly, and occasionally projected onto the time axis of one of the processors: completion events are never projected, because they only affect the resource usage of the processor on which they occur.

6 Data Transfer Cost

The network causes a delay between the time an initialization event occurs and the time it is perceived on the processor of the consumer. This delay is caused by the data transfer across the network channels and must be added to the value of the initialization event, when projecting it onto the time axis of the consumer's processor.

The data transfer speed of the network channels depends on their load. This load is composed of data transfers between producer and consumer processes and between processes and disk peripherals.

6.1 Channel Load due to Dataflow

We represent the network configuration as a graph $G(V, E)$, where V represents the set of processors and E the set of channels connecting them. A query schedule assumes a “conceptual” topology, i.e. a set of “virtual” links among the processors that must interact. These links are materialized on the network’s channels by a routing mechanism. We make the following assumptions on the behaviour of this router:

- The router may change the mapping of links dynamically during execution, in order to evenly distribute the load over all channels.
- The router chooses the channels to materialize a link according to some criterion. The optimizer knows this criterion, so that it can simulate the router’s decisions and compute the resulting channel load.
- The path of channels materializing a link may not change between two consecutive events on the reference time axis.

According to our assumptions, the load on a channel may change during execution, but it remains constant in the time span between any two consecutive events. We can approximate this load for this time span by counting the processes transmitting data to their consumers across this channel during that time span, and by calculating the amount of data being transferred.

Let $Path_j(i, k)$ denote the set of channels selected by the router to materialize the link between processors i, k at event t_j of the reference time axis; this path will not change prior to event t_{j+1} . Further, let L^e denote the set of processes communicating across channel $e \in E$ with their consumers, and let L_j^e denote its instance at t_j . The sets L^e ($e \in E$) reflect the dynamic change of network load, similarly to the sets A^i and B^i ($i \in V$) reflecting processor load.

We denote by $t_{comm}(e)$ the “ideal transfer time” on a channel $e \in E$, defined as the time required to transfer a data unit (byte, page, block) across channel e , when e has no other data to transfer. By making $t_{comm}(\cdot)$ dependent on e , we cover also heterogeneous networks, having channels of different transfer speeds. The “actual transfer time” across e at the time point of event t_j is $card(L_j^e) \cdot t_{comm}(e)$.

If a process x^i sends data to its consumer x^k , let $block(x^i, x^k)$ be the amount of data units sent at each transmission. If buffering is used, the block will consist of some data pages. If no buffering is used, the block will be a tuple. We assume that those data are accumulated on processor k , and stored locally if the start event of x^k has not occurred yet. This prevents processor i from filling its local storage with foreign data.

If t_j is the initialization event of x^i , then an initial transfer of $block(x^i, x^k)$ data units must be performed from processor i to k . Its cost is:

$$transferCost(x^i, x^k, t_j) = initData(x^i, x^k) \cdot \sum_{e \in Path_j(i, k)} card(L_j^e) \cdot t_{comm}(e) \quad (13)$$

where we sum up the actual transfer time across each channel in the path $Path_j(i, k)$.

In order to project the initialization event of x^i onto the time axis of k , this transfer delay must be taken into account. Therefore, the event is projected onto the time axis as point $t_{j_k}^k$ computed as:

$$t_{j_k}^k = t_j + \text{transferCost}(x^i, x^k, t_j) \quad (14)$$

Example. In Fig. 9, we extend the example of Fig. 8 to incorporate the data transfer cost. The events at t_2, t_4, t_6 and t_{10} are projections of events onto some local axis, perceived with a delay caused by the data transfer. The underlined timepoints t_2 and t_6 are worth noting: they correspond to initialization events that are not start events, but simply indicate a change in the network load. \square

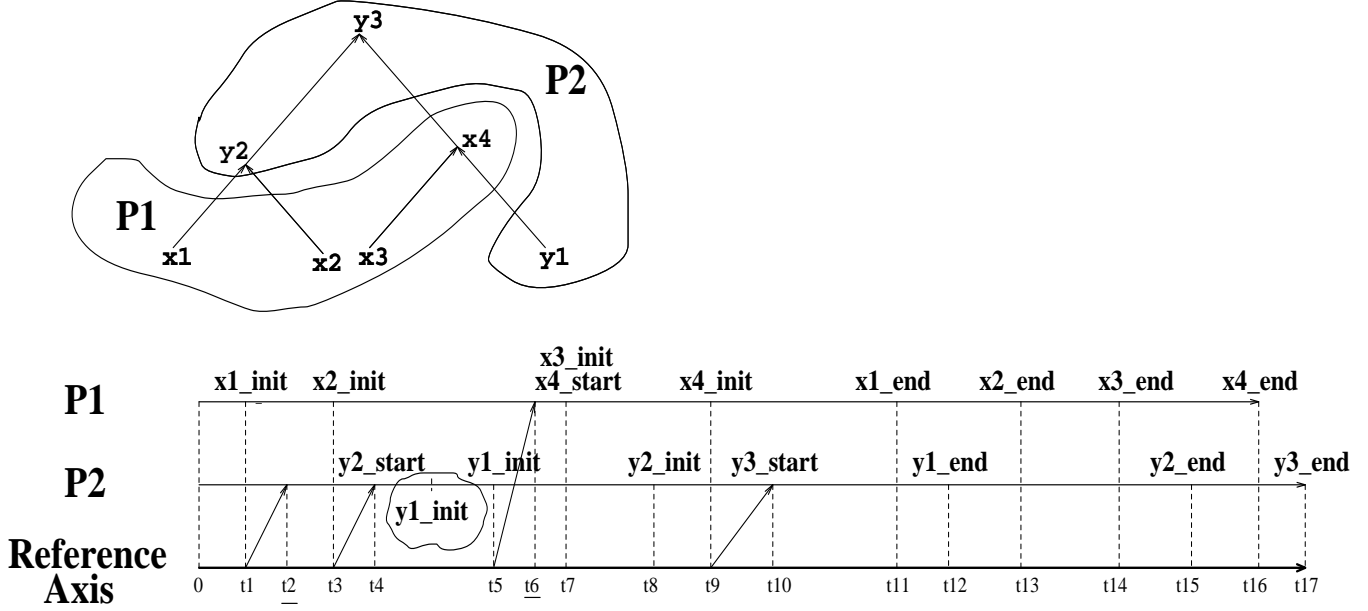


Figure 9: Impact of communication cost on query execution time

The above discussion referred to initialization events. The completion events are not projected onto anyother time axis: the completion event of a process only signals that the process releases the resources it held on the local processor i and on the channels it used for data transfer. This is reflected in the contents of the B^i instances describing the processor load and of the L^e instances describing the network load.

6.2 Remote I/O Cost

The network load is further affected by data transfers from and to remote peripherals. If each processor has its own local storage, as in the shared-nothing architecture, then intermediate results are stored locally and remote disk accesses may only occur at leaf nodes acquiring data from remote disks. We model this case by introducing “I/O processes”, dedicated to retrieve data from local discs and to transmit them to the network. This transmission can take place in pipeline mode. Thus, remote I/O cost is a special case of dataflow cost.

If some processors do not have access to local storage, as in the shared-disk architecture, then all their I/O requests are remote. Remote disk accesses for data transmission are caused by producers

sending data to their consumers and can be modelled by the I/O processes described above. Remote disk accesses to store and recover intermediate results must be modelled differently, because they can occur at any time, even between adjacent events, depending on the memory demand of the process at that particular time. An initial solution to this problem is described in [SF95].

7 Transfers of Temporary Data in the Shared-Disk Architecture

In a shared-disk architecture, temporary data need to be transferred across the network and stored to remote disks, whenever memory is not adequate. This activity is orthogonal to the occurrence of events, in the sense that remote data transfers may occur between adjacent events. Before we model this situation, we stress certain characteristics of the shared-disk system that affect parallel query execution.

7.1 Data management in the shared-disk architecture

According to [Rah93], each disk of a shared-disk architecture can be accessed by each processor via an interconnection network. This structure is depicted in Fig. 10, where we have attached a dedicated IO-processor on each disk, for convenience. This IO-processor is responsible for all I/O requests towards the disk.

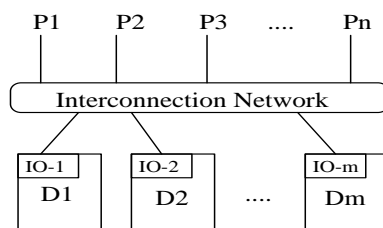


Figure 10: Shared-disk database architecture

Each processor has its own main memory, which is shared among the processes running simultaneously on it. Clearly, a multitasking scheme based on memory swapping is not appropriate for this kind of architecture, since the data traffic caused by swapping would be prohibitive. In shared-disk systems, like the SuperclusterTM of PARSYTEC, memory is therefore distributed among the processes. Among the possible memory assignment patterns, we select the following:

Let M^i be the total memory space of processor i . Further, let n_j^i be the number of processes active on this processor by the time of event t_j . Then, the memory space available to one of these processes, say x , is $memSpace(x, j) = \frac{M^i}{n_j^i}$. The second parameter of this function denotes the dependence of the memory space on the events placed on the reference axis: this is necessary, since we detect changes in the number of runnable processes via the events.

7.2 Memory space demand of a process

The memory assignment mechanism implies that there is an upper limit to the space that may be consumed by a process at any time. This limit changes over time, as processor utilization changes.

Influence of process semantics. The memory assignment mechanism is based on the assumption that a process fully utilizes the memory space available to it. However, several refinements can be made:

- A selection algorithm needs only one input data page to perform its comparisons. If buffered data transfer is performed, one further output page is needed, where data are stored before being transmitted.
- A nested loops algorithm does not need more space than required to accommodate the inner relation and one page of the outer relation.
- For the hybrid hash join algorithm of [Sha86], the memory space required should be sufficient to accommodate just the square root of the inner relation's size.
- A merge sort algorithm depends on the memory space available, but once the size of the merge run is determined, the demand does not change.
- Some algorithms can adjust themselves to changes in the available memory size (as the nested loops algorithm), while for others the memory demand is fixed upon initiation (as for the merge sort algorithm) and cannot be changed later on.

Evolution of memory demand. For each process x , we can determine a minimum and a maximum memory space size limit, $minMem(x)$ and $maxMem(x)$, according to the semantics of the used algorithm and the size(s) of the relation(s) being processed. The actual memory demand $actMem(x)$ depends on the memory available to the process at its start event, say $t_{j(x)}$:

$$actMem(x) = \begin{cases} maxMem(x) & , \quad maxMem(x) \leq memSpace(x, j(x)) \\ minMem(x) & , \quad maxMem(x) > memSpace(x, j(x)) \text{ and} \\ & x \text{ does not expand over the available memory} \\ memSpace(x, j(x)) & , \quad \text{otherwise} \end{cases}$$

At a later event t_j the memory usage of x is:

$$memUsage(x, j) = \begin{cases} \min(memSpace(x, j), maxMem(x)) & , \quad x \text{ can adjust itself} \\ & \text{to the available memory} \\ \min(actMem(x), minMem(x)) & , \quad x \text{ can neither adjust itself nor} \\ & \text{expand over the available memory} \\ actMem(x) & , \quad \text{otherwise} \end{cases} \quad (15)$$

The classic merge sort algorithm is quite inflexible to changes of memory availability, so that its memory demand must be considered fixed. This does not preclude the usage of more sophisticated algorithms: an algorithm that can adjust itself to memory demand will obtain the maximum needed and available. In the other cases, if the minimum demand is not available, the QEP is infeasible, in the sense that the selected processor cannot be used by that process.

7.3 Memory faults and I/O-requests

The memory space available to a process can only change at the start event and completion event of coexisting processes. However, a process does not utilize the available memory to the same extent during its whole lifetime: a sorting process requires a larger amount of memory when it

constructs the sort runs than it needs when it produces the sorted output. So, the time at which a process asks for disk access depend both on the available memory space and on the semantics of the process.

Occurrence of I/O requests. The problem can be formulated as follows: when does a process ask to write data to disk, when is it ready to read them back, and what is the amount of data being transferred each time?

In our model, we have treated processes as black boxes. Identifying the relationship between disk access requests and process semantics implies the removal of this simplification, and the close study of the algorithm's behaviour. This is a formidable task even for classic relational algorithms: the algorithm has to be broken into phases, the cost of each phase estimated and mapped to a range of events, taking into account the impact of multitasking on the execution time. For processes implementing object-oriented methods this is impossible, as very little meta-information is available. Therefore, we adopt the following simplified scheme of a process's behaviour.

A process x will *not* ask for disc access to transfer temporary data, if and only if its actual memory demand is the maximum it needs and this maximum is always available. This means that a process will never ask for temporary I/O, if at its start event it obtained the maximum memory it needed, and it keeps obtaining that maximum thereafter. Algorithms like hash join and nested loops join will adjust themselves to less memory than the maximum required by using the disk at temporary storage. So, the probability that process x will make an I/O request at the j^{th} event t_j is $io(x, j)$, computed using Eq.15 as:

$$io(x, j) = P(memUsage(x, j) < maxMem(x))$$

If this probability is greater than zero, the process x^i sends data to an IO-process, say x^k , in pipeline mode. The cost function of the data transfer is trivial, the input rate of the consumer x^k is equal to the output rate of the producer x^i , and the total amount of data being transferred is equal to $maxMem(x^i)$. Those data are not written at once; obviously, they are transferred in data units of size no larger than $memUsage(x^i, j)$.

Once the transfer has been completed, x^i reads the data back into its memory. The total amount being read is again $maxMem(x^i)$, the transfer block size is $memUsage(x^i, j)$, the cost function is the same as above and the output rate of the producer x^k is equal to the input rate of the consumer x^i .

Amount of data being transferred. Disk write operations may occur whenever an event makes the above probability more than zero. So, the amount of data transferred from process x^i to IO-process x^k at some event t_j is:

$$ioData(x^i, x^k, t_j) = (io(x^i, j(x^i)) + io(x^i, j)) \cdot memUsage(x^i, j) \quad (16)$$

where $j(x^i)$ is the ordinal number of the start event for x^i . The probability $io(x^i, j(x^i))$ is computed when x^i starts execution and is used as a constant thereafter.

It should be noted that Eq.16 gives a pessimistic estimate of the I/O cost of a process: if the process obtains less memory than its maximum when it starts, it will constantly require disk access until it terminates. On the other hand, assuming I/O access only at events reducing the memory space (as are the start events) is too optimistic: it overlooks the facts that data cached have to be read back and that some algorithms cannot adjust themselves to dynamic changes of memory size after their startup. Our pessimistic approximation amends the above facts.

Finally, we would like to mention that the knowledge of the exact cost formula of an algorithm is still not adequate to calculate the I/O cost as required by our cost function. In order to calculate the network workload, we need to estimate the I/O data transfers *at each event*, while the cost formulae for the algorithms calculate the *total* amount of data transferred. Assuming a uniform distribution of the requests throughout the execution time is not realistic, both because some algorithms (as the hash join) have peaks of I/O requests, and because the event are not necessarily distributed uniformly over the reference time axis.

7.4 Transfer Cost for I/O requests

Remote I/O requests may occur at any time during the life of a process. We can assume however that they are fully overlapped with communication and local execution cost. So, we reduce the impact of I/O to its contribution on the workload of the channels.

As described in subsection 7.1, an I/O request is routed towards the IO-processor responsible for the target disk. Those IO-processors are part of the network configuration described as a graph $G(V, E)$ in subsection 6.1 and the notion of $Path_j(i, k)$ covers also the case where either of i, k is an IO-processor. Therefore, if two processes communicate by the time of a certain event, the cost function estimating channel workload must incorporate the load caused by remote I/O accesses across all channels.

If process x^i requests disk access via IO-processor k at event t_j , then x^i is added to all sets $L_j^e, e \in Path_j(i, k)$. The amount of data to be transferred is $ioData(x^i, x^k, t_j)$. Hence, using Eq.16:

$$transferIOcost(x^i, x^k, t_j) = ioData(x^i, x^k, t_j) \cdot \sum_{e \in Path_j(i, k)} card(L_j^e) \cdot t_{comm}(e) \quad (17)$$

Hence, we merge Eq.13 for the communication between the non IO-processors i, k with Eq.17 for the communication with an IO-processor, into one common formula for the transfer cost between processes x^i and x^k :

$$transferCost(x^i, x^k, t_j) = \sum_{e \in Path_j(i, k)} card(L_j^e) \cdot t_{comm}(e) \cdot \begin{cases} initData(x^i, x^k) & , \text{ by Eq.13} \\ ioData(x^i, x^k, t_j) & , \text{ by Eq.17} \\ ioData(x^k, x^i, t_j) & , \text{ by Eq.17} \end{cases} \quad (18)$$

In the second case above, k is the IO-processor, while in the third case, the IO-processor is i . All paths $Path_j(\cdot)$ considered in the above formulae contain also the channels used for disk data transfer.

8 Conclusions

We have presented a model incorporating resource utilization in parallel query execution. Our model covers the exploitation of bushy parallelism and pipelining in both shared-nothing and shared-disk architectures. We have extended the conventional pipeline scheme by supporting pipes with multiple non-blocking producers and by integrating the impact of latency in pipelines. Further, we do not require that processes in a pipe are synchronized from the beginning to the end of query execution, but rather provide a scheme for the adjustment of their execution rates on the fly.

In order to provide the optimizer with a realistic estimate of schedule cost, we study the effect of dynamically occurring changes of resource utilization during query execution. As resource utilization we consider the evolving workload of the processors and the network channels. For the

processor load, we trace the influence of multitasking on the execution time of each process and its propagation on the other processors. For the network load, we compute the data transfer cost and the impact of multiple data transfers across the same channel on the channel's transfer speed. In our computations, we distinguish between remote disk accesses for data transmission and for the storage of intermediate results. By measuring the load on the physical channels, our cost evaluation scheme takes into account the effect of the network configuration on the query cost.

We compute the system workload by simulating the execution of the schedule, as it would take place on each processor involved. This simulation mechanism is based on the detection of interesting events, namely events that signal a change in the system load, and on their placement on a time axis. The time span covered by those events is the total duration of schedule execution.

A cost model may be evaluated in terms of its expressiveness and reliability. Our model is more expressive than previous ones describing pipelined parallelism [GHK92, LVZ93, SYT93], because it incorporates the impact of network configuration and of the dynamically evolving system load on the query execution cost. It is also more general, because it covers both the shared-nothing and the shared-disk architectures and because it considers a more generalized notion of pipeline. By capturing more factors of query execution cost, our model is also more realistic.

For the experimental evaluation of our model we plan to integrate it in a generic parallel environment, as proposed in [Gra90]. Since the search space of our cost function is very large, we will couple our model with a combinatorial optimization technique.

The generality of our model makes it particularly attractive for the study of parallel search spaces. The cost functions used in recent works on parallel spaces [LVZ93, LOY94, SHV95] simplify the actual search space by omitting certain cost factors. Studies comparing alternative strategies, as [SMK93, SG88], use even simpler cost functions. We argue that the complexity of the cost function affects the shape of the search space and the behaviour of the strategies exploring it. We intend therefore to evaluate the behaviour of different optimization strategies for our cost function, and compare it with their behaviour for simpler cost functions.

References

- [GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *SIGMOD Int. Conf. on Management of Data*, pages 9–18, San Diego, CA, 1992. ACM.
- [Gra90] Goetz Graefe. Parallelizing the Volcano database query processor. In *35th CompCon Conf., Digest of Papers*, pages 490–493, San Francisco, CA, 1990. IEEE.
- [Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
- [HM94] Waqar Hasan and Rajeev Motwani. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *Int. Conf. on Very Large Databases*, pages 36–47, Santiago, Chile, 1994.
- [HM95] Waqar Hasan and Rajeev Motwani. Coloring away communication in parallel query optimization. In *Int. Conf. on Very Large Databases*, pages 239–250, Zurich, Switzerland, 1995.
- [Hon92] Wei Hong. Exploiting inter-operation parallelism in XPRS. In *SIGMOD Int. Conf. on Management of Data*, pages 19–28, San Diego, CA, 1992. ACM.

- [IK91] Yannis Ioannidis and Y.C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications on query optimization. In *SIGMOD Int. Conf. on Management of Data*, pages 168–177, Denver, Colorado, 1991. ACM.
- [LOY94] E.T. Lin, E.R. Omiecinski, and S. Yalamanchili. Large join optimization on a hypercube multiprocessor. *IEEE Trans. on Knowledge and Data Engineering*, 6(2):304–315, 1994.
- [LVZ93] Rosana Lanzelotte, Patrick Valduriez, and Mohamed Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *Int. Conf. on Very Large Databases*, pages 493–504, Dublin, Ireland, 1993.
- [MMS94] Tadeus Morzy, Maciej Matysiak, and Silvio Salza. Tabu Search optimization of large join queries. In *EDBT'94 Int. Conf.*, pages 309–322, Cambridge, UK, 1994. Springer Verlag.
- [Rah93] Erhard Rahm. Parallel query processing in shared disk database systems. *ACM SIGMOD Record*, 22(4):32–37, 1993.
- [Sch90] Donovan A. Schneider. Complex query processing in multiprocessor database machines. Technical Report TR965, University of Wisconsin, Madison, Wisconsin, 1990.
- [SF95] Myra Spiliopoulou and Johann Christoph Freytag. Modelling the dynamic evolution of system workload during pipelined query execution. Technical Report ISS-20, Institute of Information Systems, Humboldt University of Berlin, Berlin, Germany, 1995. In preparation.
- [SG88] Arun Swami and Anoop Gupta. Optimization of large join queries. In *SIGMOD Int. Conf. on Management of Data*, pages 8–17, Chicago,IL, 1988. ACM.
- [Sha86] L.D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. on Database Systems*, 11(3):239–264, 1986.
- [SHC95] Myra Spiliopoulou, Michalis Hatzopoulos, and Yannis Cotronis. Parallel optimization of large join queries with set operators and aggregates in a parallel environment supporting pipeline. *IEEE Trans. on Knowledge and Data Engineering*, 1995. To appear.
- [SHV95] Myra Spiliopoulou, Michalis Hatzopoulos, and Costas Vassilakis. A cost model for the estimation of query execution time in a parallel environment supporting pipeline. *Computers & Artificial Intelligence*, 1995. To appear.
- [SMK93] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Optimizing join orders. Technical Report MIP9307, Faculty of Mathematic, University of Passau, Passau, Germany, 1993.
- [SYT93] Eugene J. Shekita, Honesty C. Young, and Kian-Lee Tan. Multi-join optimization for symmetric multiprocessors. In *Int. Conf. on Very Large Databases*, pages 479–492, Dublin, Ireland, 1993.
- [ZZBS93] Mikal Ziane, Mohamed Zaït, and Pascale Borla-Salamet. Parallel query processing with zigzag trees. *The VLDB Journal*, 2(3):277–301, 1993.