Proceedings of International Conference
on Algorithms

Scheduling and Processor Allocation for Pipeline Execution of Multijoin Queries

Yin-Fu Huang, Jyh-Her Chen, and Longson Lin
Department of Electronic Engineering
National Yunlin Institute of Technology
Touliu, Yunlin, Taiwan 640, R.O.C.
Email: huangyf@el.yuntech.edu.tw

Abstract

In this paper, we explore the scheduling and processor allocation for pipeline execution of multijoin queries. To improve the pipeline execution of hash joins in a multiprocessor system, a cost model is proposed to compute the execution time of a pipeline segment. Based on the model, two heuristic scheduling algorithms that produce a join sequence with shorter execution time for a pipeline segment, are presented. As for processor allocation, we also present two heuristic processor allocation algorithms to make the processors have least idle time. To show the performance of our algorithms, a testbed system has been implemented on a multiprocessor machine, 16 transputers connected with a ring. In our experiments, LPC_pm has better performance than LPC_sm when the intermediate results of pipeline segments are small. And LIT_stage always performs better than LIT_seg for all cases.

1 Introduction

When queries become more complex and relations grow larger, the performance of a database system will become a critical issue. Some complex queries may take hours or even days to complete, and system performance will degrade by those complex queries. In a relational database system, a join operation is the most expensive one, especially with the increases in database size and query complex [2]. Multiprocessor-based parallel machines had been used to improve system performance due to their high potential for parallel execution of complex database operations [2]. Thus parallelism is the only primary solution to increase system performance.

Different join methods can be parallelized with the pipelined approach. These methods includes nested loop, sorted merge, and hash based. The nested loop method has been found to be useful only when the relations to be joined are relatively small [6]. For the sorted merge method, it has better performance when the relation size is small and the join operation has low selectivity. The hash based method is faster under most conditions and easier to parallelize [9]. The hash based method has two phases. The first one is to build a hash table. The smaller relation (inner relation), say R, is fitted into memory by hashing on the join attribute of each tuple. The next one is to probe the hash table. Another relation (outer relation), say S, is hashed on the join attribute of each tuple, based on the hash function used in the first phase. The hash value is used to probe the hash table. If tuples in the table match the hash value, both tuples in relations R and S will be combined.

Note that different execution sequence of join operators in a query will result in different execution cost. Nevertheless, generating an optimal processing strategy for a multi-join query has been proven to be NP hard [8]. Thus several heuristic algorithms have been proposed to solve the problem. In [8], a two-way semi-join (forward reduce and backward reduce) was proposed to reduce data transmission and processing on a LAN environment. The algorithm makes the query site free from having to store and process intermediate results, and this accounts for signification saving in I/O. But the join order was not explored there. In [10], Srivastava and Elsesser proposed a new heuristic algorithm that breaks the cycle by constructing a plan tree layer by layer in a bottom-up manner. Nevertheless, the search space is very huge when the system memory is large or the relation size is small. In [2], an analytical model for the execution of a pipeline segment was first derived, and then two heuristic schemes to build the segmented right-deep tree for efficient query execution was proposed. The two algorithms focus on how to minimize the amount of works. But minimizing the amount of works does not necessarily minimize the execution time of a join query in a parallel machine.

In this paper, we explore the scheduling and

processor allocation for pipeline execution of multijoin queries. In the next section, the system models and cost formulas are described. In Section 3, the scheduling algorithms of a muti-join query are proposed. Then the processor allocation for each pipeline segment is explored in Section 4. To show how better our algorithms perform than other ones, the experiments are conducted in Section 5. Finally, conclusions are given in Section 6.

2 System Models

The system we propose here is a homogeneous one; i.e. all processors are identical. We focus on intra-query parallelism; thus a join operator is executed within only one processor. Furthermore. we assume the relations involved in a join operator can be fitted into the memory of a processor, and the intermediate results that each pipeline segment produces, will have to be written back to disks.

2.1 The Description of Pipeline Segments

The execution of a query can be denoted by a segmented right-deep tree [2], as shown in Fig. 1, that allows the implementation of a pipelined approach. The segmented right-deep tree is composed of several pipeline segments that are executed one by one in a bottom up manner. Then the pipeline segment includes several stages during which hash joins are executed.

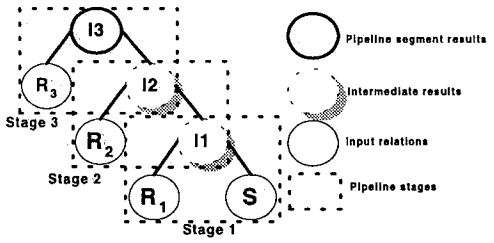

Fig. 1 A Pipeline Segment

2.2 The Cost Model

Before evaluating a pipeline segment, all associated relations must be loaded into target nodes. Thus the distribution time of relations from disks to nodes must be derived. Here a relation assigned to pipeline stage i is called the i-th relation. In the centralized type, the coordinator retrieves relations from disks one by one, and distributes them to target nodes. Owing to this sequential activity, the distribution time of stage i includes disk retrieval time, data transmission time, and the distribution time of stage i-1. However the distribution is performed in parallel in the distributed type. Thus the distribution time of a stage only contains its local disk retrieval time and data transmission time. Finally, the distribution time of stage i can be expressed as follows where the value of I/O_type is dependent on the storage architecture of the system.

$$T_{distribution}(i) = T_{distribution}(i-1) \times I/O\_type + \lceil Size(R_i)/B_{size} \rceil \times t_{I/O\_r} + \lceil Size(R_i)/P_{size} \rceil \times t_c, \quad 1 \le i \le N$$

$$T_{distribution}(0) = \lceil Size(S)/B_{size} \rceil \times t_{I/O\_r} + \lceil Size(S)/P_{size} \rceil \times t_c$$

After the distribution, each stage will build its hash table, and perform the probing according to the data from the previous stage. We assume the building time (or probing time, or even combining time) of all tuples is identical, respectively. The processing time in the probing phase consists of probing time and combining time, and it can be described as follows. A tuple in the outer relation is first probed through the hash function, and then compared with the tuples in the hash table if it hits the table. Finally, the two tuples are combined as an intermediate tuple if both match. Both building time formula and processing time formula are expressed as follows :

$$T_{build}(i) = Card(R_i) \times t_{build}, \quad 1 \le i \le N, \quad T_{build}(0) = 0$$

$$T_{process}(i) = Card(R_i) \times t_{probe} + Card(I_i) \times t_{combine}, \quad 1 \le i \le N, \quad T_{process}(0) = 0$$

There are two types of extra cost for pipeline synchronization. One is idle time that is the time for a stage to wait for the first packet from the previous stage. If a node has built its hash table and no packets from the previous stage have arrived yet, it must wait. It occurs when the i-th relation is much smaller than the (i-1)th relation and the i-th pipeline stage completes building the hash table before the (i-1)th pipeline stage. This case is shown in Fig. 2. Absolutely there may be no idle time for a stage when the first packet from the previous stage has arrived and it is building its hash table. Fig. 3 shows another case. Thus the idle time for the i-th stage may be expressed as follows:

$$T_{idle}(i) = \begin{cases} T_{distribution}(i-1) + T_{build}(i-1) + T_{channel}(i-1) \\ \quad - T_{distribution}(i) - T_{build}(i), & \text{if idle} \\ 0, & \text{if not idle} \end{cases}$$

$$1 \le i \le N$$

$$T_{idle}(0) = 0$$

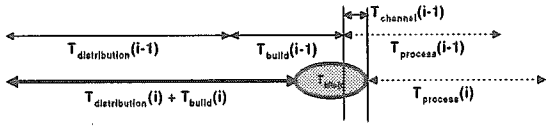

Fig. 2 Idle Time Case

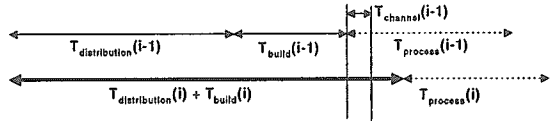

Fig. 3 No Idle Time Case

Another extra cost for pipeline synchronization is overhead. For flow control, a stage may wait for packets from the previous stage or suspend transmitting packets to the next stage during pipeline processing. It occurs when the previous stage is too slow to produce packets for this stage or the next stage has no free buffer to receive its intermediate results. Fig. 4 shows the case. Absolutely there may be no overhead for a stage when the flow in the pipeline is smooth. Fig. 5 shows another case. Thus the overhead for the i-th stage may be expressed as follows where T_{node} denotes the sum of $T_{distribution}$, T_{build}, T_{idle}, and $T_{process}$.

$$T_{overhead}(i) = \begin{cases} T_{node}(i-1) + T_{overhead}(i-1) + T_{channel}(i-1) \\ \quad - T_{node}(i), & \text{if overhead} \\ 0, & \text{if no overhead} \end{cases}$$

$$1 \le i \le N$$

$$T_{overhead}(0) = 0$$

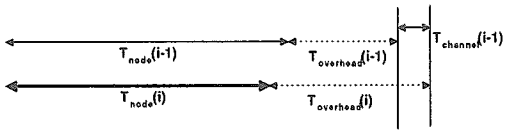

Fig. 4 Overhead Case

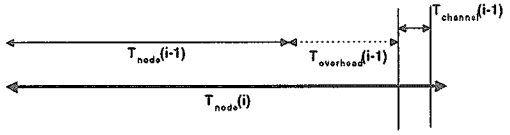

Fig. 5 No Overhead Case

Finally, it is obvious that the execution time of a pipeline segment is the completion time of the last pipeline stage. The cost formula is expressed as follows:

$$T_{execution}(PS) = T_{node}(N) + T_{overhead}(N)$$

3 Scheduling of a Multijoin Query

Two heuristic scheduling algorithms to produce pipeline segments are proposed here. One is LPC_sm (i.e. least processing cost based on sequential mode) whose solution allows only one pipeline segment processed at a time. Another one is LPC_pm (i.e. least processing cost based on parallel mode) whose solution allows more than one pipeline segment processed concurrently. Within both algorithms, two different strategies can be applied to find the join sequence for a pipeline segment, respectively; those are bottom up and top down approaches.

3.1 Basic Concepts

The objective of a scheduling algorithm is to produce a join sequence with shorter execution time for a multijoin query. We can use an exhaustive method to find an optimal join sequence, but the scheduling time is definitely enormous since the number of join sequences for a query with n joins is n!. Thus heuristic scheduling algorithms are proposed to find a join sequence with shorter execution time.

A query can be represented as a **join query graph** denoted by G = (V, E) where V is the set of nodes and E is the set of edges. Each node in a join query graph represents a relation. Two nodes are connected with an edge if there exists a join predicate on some common attribute of two corresponding relations. As shown in Fig. 6, there are nine relations and thirteen joins involved in the query. In our algorithms, a join query graph is partitioned into a few **star join graphs** as shown in Fig. 7. One of the relations, called **main relation**, is connected to all other relations. Other relations are called **satellite relations**. As shown in Fig. 7, R4 is main relation, and R1, R2, R3, R6, and R7 are satellite relations of a star join graph.

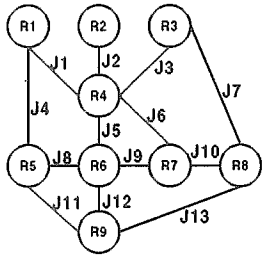

Fig. 6 A Join Query Graph

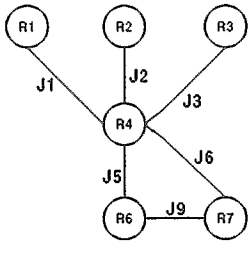

Fig. 7 A Star Join Graph

Our algorithms to find a pipeline segment are based on evaluating the processing cost per byte. Here we focus on how to find a star join graph first. As for the join sequence of a star join graph will be discussed later. In the algorithm LPC_sm, a star join graph with the least processing cost per byte is chosen at a time. Then these relations in the star join graph are clustered as a relation in the new join query graph. The process to find a star join graph with the least processing cost per byte continues until the new join query graph reduces to only one relation. As shown in Fig. 8(a), we have several star join graphs

embedded in the join query graph, such as {R1, R2, R3, R4, R6, R7}, {R3, R7, R8, R9}, {R4, R5, R6, R7, R9} etc. Provided the processing cost of {R1, R2, R3, R4, R6, R7} is the least, it will be clustered as a relation named I1. Then the join query graph is transformed into the new graph as shown in Fig. 8(b). Following the same way, the only one star join graph {I1, R5, R8, R9} is clustered and named I2 as shown in Fig. 8(c). The joins in a star join graph constitute a pipeline segment after a join sequence is determined. Here the pipeline segments producing relation I1 and I2 can not be processed in parallel. That is why we call this algorithm LPC_sm.

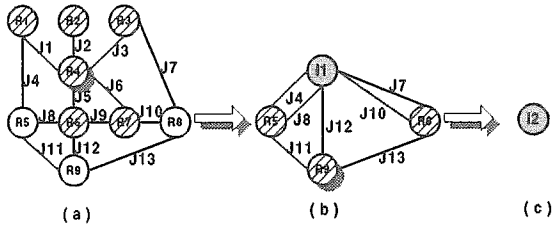

Fig. 8 Transformation of Join Query Graphs by Algorithm LPC_sm

To allow more than one pipeline segment processed in parallel, we have another algorithm called LPC_pm. The algorithm LPC_pm tries to find independent pipeline segments at a time. Here a join query graph **with layer i** is introduced to describe the independence between pipeline segments. That is the pipeline segments with the same layer can be processed in parallel. Like the algorithm LPC_sm, the evaluation is still based on the processing cost per byte. Fig. 9(a) shows a join query graph with layer 1 where the star join graph {R1, R2, R3, R4, R6, R7} is first found, and then {R5, R8, R9}. They can be processed concurrently. Through a transformation, Fig. 9(b) shows a new join query graph with layer 2 where relation I1 and relation I2 are produced by {R1, R2, R3, R4, R6, R7} and {R5, R8, R9}, respectively. Finally, relation I3 is produced by relations I1 and I2, as shown in Fig. 9(c). The detailed procedures about algorithms LPC_sm and LPC_pm are described in Section 3.2.

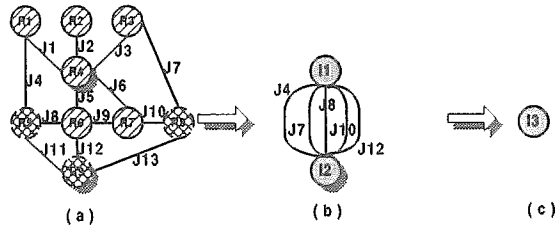

Fig. 9 Transformation of Join Query Graphs by Algorithm LPC_pm

Here we discuss the join sequence of a star join graph. As mentioned before, a star join graph becomes a pipeline segment after its join sequence is determined. For example, the join sequence of a star join graph, (((((R4 join R6) join R3) join R7) join R1) join R2), and its corresponding pipeline segment are shown in Fig. 10. The outer relation of pipeline stage 1 is actually the main relation of a star join graph. Therefore, the join sequence we want to determine is the order of satellite relations. The strategy BOTTOM_up uses a greedy method to find the first join with the least execution time, then the second join with the next least execution time, and so. In pipeline stage 1 as shown in Fig. 10, the outer relation is R4, and the inner relation is chosen from relations R1, R2, R3, R6, and R7 according to which join with R4 has the least execution time. Here relation R6 is chosen first. Then relation I1 produced by R4 and R6 acts as the outer relation of pipeline stage 2. Following the same way, the inner relation of each pipeline stage is chosen until all satellite relations are emptied, and the pipeline segment is then built.

Fig. 10 A Star Join Graph and the Corresponding Pipeline Segment by Strategy BOTTOM_up

The strategy TOP_down is another one to find the join sequence of a star join graph. In most cases, the last pipeline stage could be the bottleneck of a pipeline segment. Thus the strategy TOP_down finds a join sequence where the last pipeline stage has the least work cost. Like the strategy BOTTOM_up, the outer relation of pipeline stage 1 is definitely the main relation of a star join graph. Therefore, the join sequence to be found is the order of satellite relations. The difference between strategies BOTTOM_up and TOP_down is the building way. BOTTOM_up builds a pipeline segment in a bottom up manner, whereas TOP_down uses a top down method to construct a pipeline segment. An example is shown in Fig. 11. First, let R4 be the outer relation of pipeline stage 5 temporarily, and the inner relation is chosen from relations R1, R2, R3, R6, and R7 according to which join with R4 has the least work cost. Let R2 be the inner relation selected. Next, R4 acts as the outer relation of pipeline stage 4 again, and relation R1 is chosen to join R4 and R2 (i.e. ((R4 join R1) join R2)), so that the last pipeline stage still has the least work cost. Following the same way, the inner relation of each pipeline stage is chosen until all satellite relations are emptied. Finally, main relation R4 is assigned to be the outer relation of pipeline stage 1. The detailed procedures about strategies BOTTOM_up and TOP_down are described in Section 3.2.

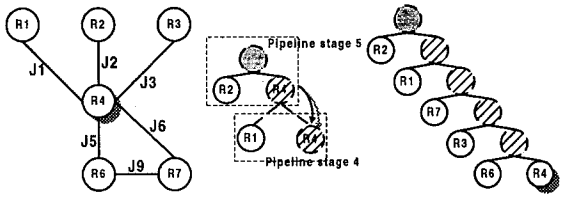

Fig. 11 A Star Join Graph and the Corresponding Pipeline Segment by Strategy TOP_down

3.2 The Heuristic Algorithms for Scheduling

3.2.1 Least Processing Cost Based on Sequential Mode

Algorithm LPC_sm finds a pipeline segment with the least processing cost per byte at a time. It has two basic steps. First, all star join graphs embedded in the join query graph are found and select the star join graph with the least processing cost per byte. Next, the join query graph is reduced. Repeat these two steps till the new join query graph has only one relation. The detailed algorithm is omitted here.

3.2.2 Least Processing Cost Based on Parallel Mode

This algorithm is similar to algorithm LPC_sm except that more than one independent pipeline segment is found and they could be processed in parallel. Algorithm LPC_pm has three steps. The first step is the same as that in algorithm LPC_sm. Next, the relations constituting the pipeline segment are removed from the query join graph so that the next independent pipeline segment can be found from that graph. Last, collect the new relation defined in step 2 in order to find pipeline segments in the next layer. We have double nested loops in this algorithm. Repeat these three steps to find pipeline segments of different layers till the new join query graph has only one relation. The detailed algorithm is omitted here.

3.2.3 Join Strategy: BOTTOM_up

Strategy BOTTOM_up finds a join sequence of a star join graph, with the least execution time. It has two basic steps. First, select a relation from all satellite relations such that its join with the relations in R_{order} has the least execution time. Next, the relation chosen above is removed from the set of satellite relations. Repeat these two steps till no satellite relations exist. Within this procedure, $T_{execution}$ has been defined in Section 2.2, which is different from $T_{exec\_per\_byte}$. The detailed procedure is omitted here.

3.2.4 Join Strategy: TOP_down

Strategy TOP_down also finds a join sequence of a star join graph, but uses a different appoach. It has three basic steps. First, select a relation from satellite relations such that the last pipeline stage always has the least work cost. Second, the relation chosen above is removed from the set of satellite relations. Repeat the first and second steps till no satellite relations exist. Last, add the main relation to output set and reverse the order of output set. The detailed procedure is omitted here.

3.2.5 Common Functions Used in the Scheduling Algorithms

Three common functions called in above algorithms are described as follows. Function GROUP combines relations as a star join graph, but the number of combined relations must be less than that of processors in the system. Each time a join with the smallest selectivity is considered. That is because the results produced by each pipeline segment must be written back to disk, and we want the results as small as possible to save disk I/O time. Next, function LOAD calculates the work cost in the last pipeline stage per byte. Last, function Texec_per_byte calculates the execution time of a pipeline segment per byte.

4 Processor Allocation for Each Pipeline Segment

The relations must be distributed to processors before the execution of a pipeline segment. The processor allocation algorithms decide that the relations of a pipeline segment will be assigned to which processors. Although processor allocation problem depends on the underlying platform, the general performance measure is based on minimal transmission time or minimal idle time. In this section, two processor allocation algorithms are proposed: one is called LIT_seg, i.e. least idle time based on a pipeline segment, and another is called LIT_stage, i.e. least idle time based on a pipeline stage.

4.1 Basic Concepts

In the processor allocation algorithms, a formula called Transmission Cost Estimation (TCE) is used to calculate the transmission time of a pipeline segment. It can be expressed as follows.

$$TCE(NODE\_SET) = \sum_{i=1}^{m-1} tc_i + OVERHEAD_{distribution} + OVERHEAD_{write\_back}$$

Within this formula, NODE_SET is a set of nodes assigned to a pipeline segment, and tc_i is the transmission time of the i-th node. Before the execution of the pipeline segment, it takes some time for Coordinator to distribute data to Workers, and the time is called $OVERHEAD_{distribution}$. Similarly $OVERHEAD_{write\_back}$ is the transmission time when the results produced by the pipeline segment are written back to disks. Given an example NODE_SET = {4, 5, 7} in Fig. 12, its TCE will be (1+2)+(4+5+7)+(1)=20. The minimal transmission time can be found as follows. Let the total number of idle nodes be N, and m is the number of nodes required for a pipeline segment. Then we will have N-m+1 ways to select m nodes in order from the N idle nodes. Among theses candidates, the allocation with the minimal transmission time will be found.

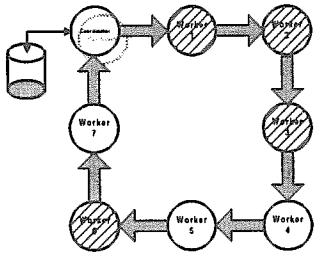

Fig. 12 An Example for TCE

In addition to find the minimal transmission time described above, we expect the node set found to have the least idle time. According to the strategy "The pipeline segment with the least relations is the first one served", algorithms LIT_seg and LIT_stage assign idle nodes to the pipeline segment with the least relations first. After the assignment, the pipeline segment with the next least relations is assigned repeatedly until the number of idle nodes in the system are less than that required by the pipeline segment. Since the execution time of each node involved in a pipeline segment can be calculated, we know when a busy node will be released. Then these released nodes and original remainder idle nodes will be considered for the next pipeline segment. The difference between algorithms LIT_seg and LIT_stage is when busy nodes will be released. In algorithm LIT_seg, all busy nodes are released together when the entire pipeline segment is completed. And in algorithm LIT_stage, a busy node is released immediately once its corresponding pipeline stage is completed. In sequential mode mentioned in Section 3.1, each layer has only one pipeline segment, so there is at most one pipeline segment executed at a time. Thus algorithm LIT_stage is never applied in sequential mode. As for parallel mode, both algorithms LIT_seg and LIT_stage can be applied. Due to the feature that the tasks of previous nodes will be finished earlier than the latter ones in the pipeline processing, LIT_stage should perform better than LIT_seg. This is because the idle time of previous nodes is not utilized in algorithm LIT_seg until the entire pipeline segment is completed. The detailed procedures of algorithms LIT_seg and LIT_stage will be described in Section 4.2.

4.2 The Heuristic Algorithms for Processor Allocation

4.2.1 Least Idle Time Based on a Pipeline Segment

Algorithm LIT_seg releases all busy nodes together when the entire pipeline segment is completed. It has five basic steps. First, test whether there exists a pipeline segment in the current layer, which can be executed using the remainder idle nodes in the system. If yes, select a pipeline segment with the minimal relations and allocate nodes to make it have the minimal transmission time. Next, record completion time of each node allocated to the pipeline segment. If no pipeline segment can be executed, release the nodes allocated to a pipeline segment whose completion time is the nearest. Repeat node allocation and node release till all pipeline segments are assigned. The detailed algorithm is omitted here.

4.2.2 Least Idle Time Based on a Pipeline Stage

This algorithm is similar to algorithm LIT_seg except that the completion time of each node is recorded according to each pipeline stage. The detailed algorithm is omitted here.

5 The Experiments

5.1 The Experimental Environment

The experiments are conducted on a testbed system constructed by 16 transputers connected with a ring, where the different heuristic algorithms proposed in Section 3 and 4 are implemented. In the system only root node can access a disk, and the involved relations are arranged to fit the main memory on each node. In all experiments, the system parameters are set or measured as shown in Table I.

Table I The System Parameters

Packet Size	10 KBytes
Router Buffer Capacity	10 packets
Worker Buffer Capacity	10 packets
Tuple Length	50 bytes
Disk Transfer Rate	354 μs/packet
Channel Transfer Rate	171 μs/packet
Processing Time	213 μs/tuple
Building Time	57 μs/tuple

To observe the performance of proposed algorithms under different situations, the workload parameters used in the experiments are defined. These workload parameters provide us to clarify which algorithm has better performance under what environments. The workload parameters consists of join selectivity, relation cardinality, relation number, processor number.

5.2 The Experimental Results

We explored join selectivity in Experiment 1, relation size in Experiment 2, and the interrelations between relation number and processor number in Experiment 3.

Experiment 1: join selectivity
Case 1: High, Case 2: Low
Selectivity : 0.001 - 0.005, 0.0001- 0.0005
Cardinality : 750 - 1000
Relation no. : 10
Processor no. : 8

Algorithms	Case 1	Case 2
Pm-Bottom-Seg	6042	4186
Pm-Bottom-Stage	6027	4154
Pm-Top-Seg	5405	4189
Pm-Top-Stage	5370	4161
Sm-Bottom-Seg	5580	5016
Sm-Top-Seg	5403	5052
Opt-RD	5209	4178

Experiment 2: relation cardinality
Case 1: Small, Case 2: Middle, Case 3: Large
Selectivity : 0.0001 - 0.0005
Cardinality : 500 - 750, 750-1000, 1000-1250
Relation no. : 10
Processor no. : 8

Algorithms	Case 1	Case 2	Case 3
Pm-Bottom-Seg	3216	4186	5768
Pm-Bottom-Stage	3198	4154	5744
Pm-Top-Seg	3236	4189	6036
Pm-Top-Stage	3215	4161	6001
Sm-Bottom-Seg	3375	5016	6152
Sm-Top-Seg	3648	5052	6218
Opt-RD	3218	4178	5327

Experiment 3: relation number and processor number
Case 1, Case 2, Case3, Case4
Selectivity : 0.0001 - 0.0005
Cardinality : 500 - 750
(Relation, Processor) : (6,8), (12,8), (12,16), (20,16)

Algorithms	Case 1	Case 2	Case 3	Case 4
Pm-Bottom-Seg	2395	5035	5292	8554
Pm-Bottom-Stage	2364	5002	5051	7491
Pm-Top-Seg	2401	4717	4875	7935
Pm-Top-Stage	2368	4709	4747	7360
Sm-Bottom-Seg	2966	5098	5436	6889
Sm-Top-Seg	2369	4749	4752	6717
Opt-RD	1912	4162	3722	6242

6 Conclusions

In this paper, we explore the scheduling and processor allocation for pipeline execution of multijoin queries. To improve the pipeline execution of hash joins in a multiprocessor system, a cost model is proposed to compute the execution time of a pipeline segment. Based on the model, two heuristic scheduling algorithms that produce a join sequence with shorter execution time for a pipeline segment, are presented. Algorithm LPC_sm processes only one pipeline segment at a time, whereas algorithm LPC_pm allows more than one pipeline segment to be processed at the same time. Within both algorithms, two different strategies can be applied to find the join sequence for a pipeline segment, respectively; those are bottom up and top down approaches. Strategy BOTTOM_up finds the join sequence for a pipeline segment with less execution time, whereas strategy TOP_down finds the last pipeline stage of a pipeline segment with minimum work cost. As for processor allocation, we also

present two heuristic algorithms to make the processors have the minimal transmission time and the least idle time. Algorithm LIT_seg assigns processors to the next pipeline segment when the current pipeline segment is completed, whereas algorithm LIT_stage does based on a pipeline stage. Although LIT_stage is always better than LIT_seg in our experiments, LIT_stage must be based on the preknowledge of completion time of each pipeline segment.

To show the performance of our algorithms, a testbed system has been implemented on a multiprocessor machine, 16 transputers connected with a ring. As shown in the experiments, LPC_pm has better performance than LPC_sm when the intermediate results of pipeline segments are small. When the processing time of a query is long or a query is decomposed into more layers, TOP_down has better performance than BOTTOM_up. LIT_stage always performs better than LIT_seg for all cases. In addition, especially when relation cardinality is small or middle, some of our algorithms are better than Opt-RD.

References

[1] M.-S Chen, P. S. Yu, and K.-L Wu, "Scheduling and processing allocation for parallel execution of multi-join queries," Proc. Eighth Int'l Conf. on Data Engineering, pp. 58-67, Feb. 1992.

[2] M.-S Chen, P. S. Yu, and H. C. Young, "Applying segmented right-deep trees to pipelining multiple hash join," IEEE Transactions on Knowledge and Data Engineering, vol. 7, no. 4, Aug. 1995, pp. 656-668.

[3] Computer System Architects, *Transputer Architecture and Overview*, Prentice Hall, 1990.

[4] S. M. Deen, D. N. P. Kanangara, and M. C. Taylor, "Multi-join on parallel processors," Proc. Second Int'l Symp. on Database in Parallel and Distributed Systems, pp. 92-102, July 1990.

[5] L. Harada and N. Akaoshi, "Evaluation of linear join processing tree in shared-nothing database environment," Proc. Fifth Int'l Conf. on Computing and Information, pp. 413-417, May 1993.

[6] M. S. Lakshmi and P. S. Yu, "Effectiveness of parallel joins," IEEE Transactions on Knowledge and Data Engineering, vol. 2, no 4, Dec. 1990, pp. 410-424.

[7] K. P. Mikkilineni and S. Y. W. Su, "An evaluation of relational join algorithms in a pipelined query processing environment," IEEE Transactions on Software Engineering., vol. 14. no. 6, June 1988, pp. 838-848.

[8] N. Roussopoulos and H. Kang, "A pipeline N-way join algorithm based on the 2-way semijoin program," IEEE Transactions on Knowledge and Data Engineering, vol. 3, no. 4, Dec. 1991, pp. 486-795.

[9] D. A. Schneider and D. J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," Proc. ACM-SIGMOD Int'l. Conf. on Management of Data, pp. 110-121, June 1989.

[10] J. Srivastava and G. Elsesser, "Optimizing multi-join queries in parallel relational databases," Proc. Second Int'l Conf. on Parallel and Distributed Information system, pp. 84-92, Jan. 1993.