EXPLOITING INTER-OPERATION PARALLELISM IN XPRS

by

Wei Hong

Memorandum No. UCB/ERL M92/3

6 January 1992

EXPLOITING INTER-OPERATION PARALLELISM IN XPRS

by

Wei Hong

Memorandum No. UCB/ERL M92/3

6 January 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

EXPLOITING INTER-OPERATION PARALLELISM IN XPRS

by

Wei Hong

Memorandum No. UCB/ERL M92/3

6 January 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Exploiting Inter-Operation Parallelism in XPRS *

Wei Hong

Computer Science Division

EECS Department

University of California at Berkeley

hong@postgres.berkeley.edu

January 3, 1992

Abstract

In this paper, we study the scheduling and optimization problems of parallel query processing using inter-operation parallelism in a shared-memory environment and propose our solutions for XPRS. We first study the scheduling problem of a set or a continuous sequence of independent tasks that are either from a bushy tree plan of a single query or from the plans of multiple queries, and present a clean and simple scheduling algorithm. Our scheduling algorithm achieves maximum resource utilizations by running an IO-bound task and a CPU-bound task in parallel with carefully calculated degrees of parallelism and maintains the maximum resource utilizations by dynamically adjusting the degrees of parallelism of the tasks whenever necessary. Real performance figures are shown to confirm the effectiveness of our scheduling algorithm. We then revisit the optimization problem of parallel execution plans of a single query and extend our previous results to also consider inter-operation parallelism by introducing a new cost estimation method to the query optimizer based on our scheduling algorithm.

*This research was sponsored by the National Science Foundation under contract MIP 8715235.

1 Introduction

There have been growing research efforts in the area of parallel database systems during the past few years. Several research systems have been designed and/or constructed including *shared-nothing* [STON86] systems such as GAMMA[DEWI90] and BUBBA[COPE88] and shared-memory systems such as XPRS[STON88] and Volcano[GRAE90]. XPRS is a multi-user parallel database management system that we are developing based on the Postgres next-generation DBMS[STON91]. It is implemented on a shared-memory multiprocessor and a disk array as shown in Figure 1. In XPRS, all relations are striped sequentially, block by block, in a round-robin fashion across the disk array to allow maximum i/o bandwidth. A shared-memory system has two major advantages over a shared-nothing system. First, there are no communication delays because messages are exchanged through shared memory and synchronization can be accomplished by hardware spin locks. Second, the operating system can automatically perform load balancing by allocating the next ready process to the first available processor. Simulation results in [BHID88] show that the potential win of a shared-memory system over a shared-nothing system to be as much as a factor of two. As we will show in this paper, XPRS has been built to fully utilize these advantages of a shared-memory system.

There are two forms of parallelism that can be exploited in a parallel database system: *intra-operation* parallelism and *inter-operation* parallelism. Intra-operation parallelism is achieved by partitioning the input data to a certain operation and allocating multiple processors to perform the same operation on subsets of the input data, while inter-operation parallelism is achieved by allocating some processors to one operation and some other processors to another operation. In this paper, we continue our study on parallel query processing in a shared-memory environment which has previously been reported in [HONG91] and present a more complete approach that exploits both intra-operation parallelism and inter-operation parallelism.

[HONG91] is unique in its *two-phase* optimization strategy to overcome the enormous search space in the problem of optimizing parallel query execution plans. In the two-phase

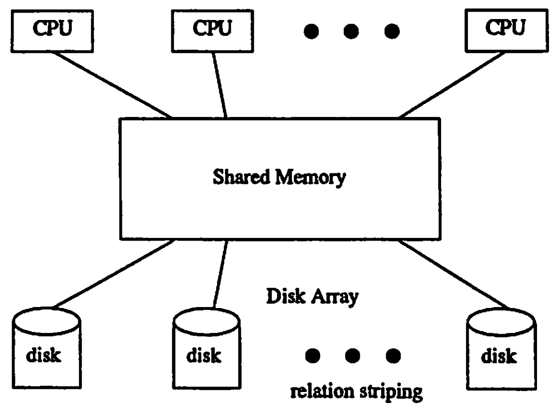

Figure 1: **The Parallel Environment of XPRS**

optimization strategy, we first, at compile time, optimize sequential query execution plans and then in a second phase, at run time, optimize the parallelizations of the optimal sequential plan chosen in the first phase. Obviously this two-phase optimization strategy greatly reduces the plan search space because it only explores parallelizations of the optimal sequential plan instead of the parallelizations of all possible sequential plans. It is also shown experimentally that this two-phase optimization strategy does not significantly compromise optimality of the resulting parallel plan. However, [HONG91] has only considered intra-operation parallelism and *left-deep tree* plans (i.e., plans that always join the result of a join with a base relation). Due to the sequential nature of left-deep tree plans, no inter-operation parallelism can be exploited within a left-deep tree plan. On the other hand, in a *bushy tree* plan (i.e., plans that allows joins of results of joins), we may be able to perform two joins in parallel before joining them together. In this paper, we will consider bushy tree plans and both intra-operation and inter-operation parallelism. Since the implementation and performance of intra-operation parallelism have already been studied in [HONG91], we will focus on inter-operation parallelism in this paper. Specifically, we will address the following two problems:

- *processor scheduling:* given a set or a continuous sequence of runable operations, what operations to execute in parallel and how many processors to allocate to each parallel operation;

- *query optimization:* how to extend the two-phase optimization algorithm in [HONG91] to handle bushy tree plans and inter-operation parallelism.

Most previous work on parallel query processing has been done on intra-operation parallelism only, e.g., [DEWI90] and [HONG91]. Some recent work has proposed to also apply inter-operation parallelism to query processing. [SCHN90] presents an experimental analysis of the query processing tradeoffs among *left-deep* and *right-deep* tree plans in a shared-nothing enironment. An important finding is that right-deep trees are superior given sufficient memory resources. However, there is no analytical cost expression which can be used by an optimizer to decide whether and when to switch from a left-deep tree to a right-deep tree. Moreover, no algorithms are proposed for determining the degree of parallelism for each parallel operation. [PIRA90] shows through an example query the use of inter-operation parallelism in query processing and models the processor scheduling problem as a modified version of the well-known *bin packing* problem. However, no general scheduling algorithm is proposed. [LU91] proposes an optimization algorithm that considers bushy tree plans and inter-operation parallelism. The algorithm only handles *synchronized* bushy tree plans, i.e., those without pipelining between joins, and uses a greedy algorithm to choose a bushy tree plan that maximizes the opportunities of inter-operation parallelism. Processor scheduling is not specifically addressed in the paper.

In this paper, we present a clean and simple algorithm for processor scheduling given n runable operations. Our main idea is to use inter-operation parallelism to combine IO-bound and CPU-bound tasks to increase system resource utilizations. Our algorithm matches up IO-bound and CPU-bound tasks with appropriate degrees of intra-operation parallelism to make both the processors and the disks operate as close to their full capacities as possible and thus to minimize the elapsed time. In order to avoid an NP-hard packing problem in the optimization of task schedules, we have also developed a mechanism, taking advantage of the low communication delay of a shared-memory system, to dynamically adjust the degree of intra-operation parallelism of a running task so that the system stays at the IO-CPU balance point as tasks start and finish. Having solved the scheduling problem, we revisit the optimization problem of parallel execution plans of a single query and extend the two-phase optimization strategy to consider bushy tree plans and inter-operation parallelism by introducing a new cost estimation method to the query optimizer based on our scheduling algorithm.

The rest of this paper is organized as follows. Section 2 describes our adaptive processor scheduling algorithm including calculation of IO-CPU balance point, dynamic adjustment to intra-operation parallelism and task re-ordering heuristics. Section 3 examines variations of our scheduling algorithm and compares their performances through experimental results. Section 4, then extends the two-phase optimization strategy based on our scheduling algorithm, and last, we conclude this paper in Section 5.

2 Adaptive Scheduling Algorithm for XPRS

In this section, we present our adaptive scheduling algorithm for XPRS. First, we describe the architecture of XPRS query processing and define the scheduling problem that we are solving. Then, we describe our classification of IO-bound and CPU-bound tasks and the calculation of the IO-CPU balance point. Then, a mechanism for dynamic adjustment of parallelism is presented. Last, we integrate our ideas into an adaptive scheduling algorithm.

2.1 Problem Definition

The architecture of XPRS query processing is given in Figure 2. There are one master Postgres backend and multiple slave Postgres backends. The master backend is responsible for all the optimization and scheduling, while the slave backends process whatever tasks that are assigned to them by the master backend. XPRS query processing consists of two phases. In the first phase, the optimizer takes one or more user queries and generates certain sequential plans for each query. In the second phase, the parallelizer parallelizes the sequential plans chosen in the first phase. In XPRS, a sequential plan is represented as a binary tree of the basic relational operations, e.g., sequential scan, index scan, nestloop join, mergejoin and hashjoin. First, the sequential plans are decomposed into *plan fragments*, i.e., a group of operations that do

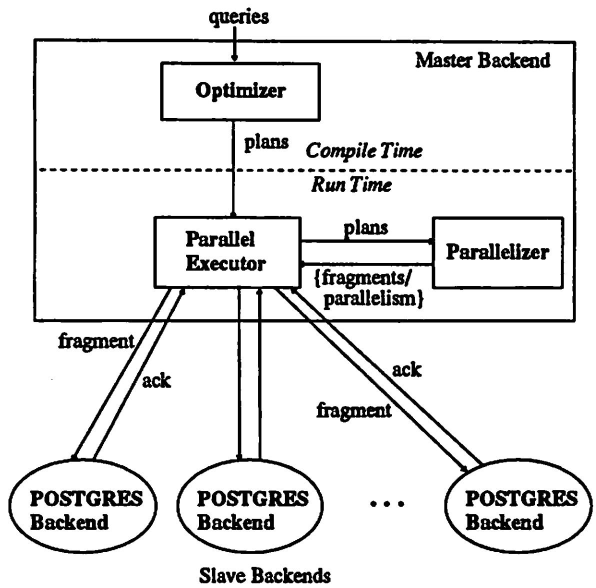

Figure 2: **Architecture of XPRS Query Processing**

not contain any blocking edges. Blocking edges are those between two operations where one operation must wait for the other to finish producing all the tuples before it can proceed. In other words plan fragments are the maximum pipelineable subgraphes of a sequential plan. Plan fragments are used as the units of parallel execution and are also called *tasks*. By inter-operation parallelism, we in fact mean *inter-fragment* or *inter-task* parallelism. After identifying all the plan fragments, the parallelizer has to find a processing schedule for the plan fragments and choose a degree of parallelism for each plan fragment. If we only consider intra-operation parallelism, i.e., we only execute one plan fragment at a time, the responsibility of the parallelizer is very simple. As presented in [HONG91], intra-operation parallelism in XPRS achieves near-linear speedup until it runs out of either available processors or the disk bandwidth and there are severe performance penalties for excessive parallelism. Therefore, the parallelizer only needs to choose a runable plan fragment, i.e., one for which all input data are ready, and choose the maximum parallelism according to the current number of available processors and disk bandwidth. However, the parallelizer's responsibility becomes much more complicated when inter-operation (inter-fragment) parallelism is taken into account. Here is the scheduling problem that we are solving for the parallelizer:

Given n runable plan fragments (tasks), $f_1, f_2, \ldots, f_n$, where the plan fragments may be from a bushy tree plan of the same query or from different queries that are simultaneously submitted,

1. decide a processing schedule for $f_1, f_2, \ldots, f_n$;

2. choose a degree of parallelism for each f_i,

such that the total elapsed time of processing $f_1, f_2, \ldots, f_n$ is minimized.

As we will see, our solution to the above described problem works for both a fixed set of tasks or a continuous sequence of tasks. Namely, we also allow n to be infinity in the above description.

2.2 IO-bound and CPU-bound tasks

The key of our solution to the above defined scheduling problem is to combine IO-bound and CPU-bound tasks through inter-operation parallelism so that the utilization of both the processors and the disks is maximized, thus the elapsed time is minimized. Before we describe our solution, we need to define our classification of IO-bound and CPU-bound tasks.

Suppose that if task f_i is processed sequentially, it generates i/o requests at rate C_i (ios/second). When f_i is executed with parallelism x, its i/o rate becomes $IO_i(x) = C_i \times x$. Suppose that the total disk i/o bandwidth is B (ios/second) and the total number of processors is N. We call task f_i *IO-bound* if $C_i > B/N$ and *CPU-bound* if otherwise. Obviously, the function $y = IO_i(x)$ is a straight line with slope C_i. If we draw the line with the rectangle bounded by B and N as in Figure 3, we can see that IO-bound tasks are those corresponding to the lines above the diagonal line and CPU-bound tasks are those corresponding to the lines below the diagonal line. As we can see, the parallelism of a task is limited by the rectangle boundaries and the maximum parallelism of a task is achieved at the intersection between the line corresponding to the task and one of the rectangle boundaries. An IO-bound task will run out of disk bandwidth before it runs out of processors. Its maximum parallelism $maxp(f_i) = B/C_i$. On the other hand, the parallelism of a CPU-bound task is only bounded

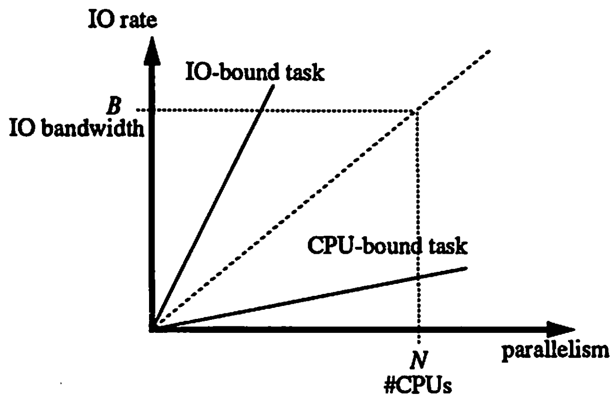

Figure 3: **IO-bound and CPU-bound tasks**

by the number of processors N, i.e., $maxp(f_i) = N$.

2.3 Calculation of IO-CPU Balance Point

Intuitively, in order to maximize the utilization of both the disks and processors, we want the system to be running at the upper right corner of the rectangle in Figure 3, i.e., the point with coordinate (N, B). Obviously, if we run one task at a time using only intra-operation parallelism, unless the line corresponding to a task is exactly the diagonal line, the system will not be running at the upper right corner of the rectangle. When we run two tasks f_i with parallelism x_i and f_j with parallelism x_j together, the system is running at the point with coordinate $(x_i + x_j, C_i x_i + C_j x_j)$. We can maximize the system resource utilization by choosing x_i and x_j according to the following equations:

$$\begin{cases} x_i + x_j & = N \\ C_i x_i + C_j x_j & = B \end{cases}$$

By solving the above equations, we get,

$$\begin{cases} x_i & = (B - C_j N)/(C_i - C_j) \\ x_j & = (C_i N - B)/(C_i - C_j). \end{cases}$$

We call (x_i, x_j) the *IO-CPU balance point* for tasks f_i and f_j.

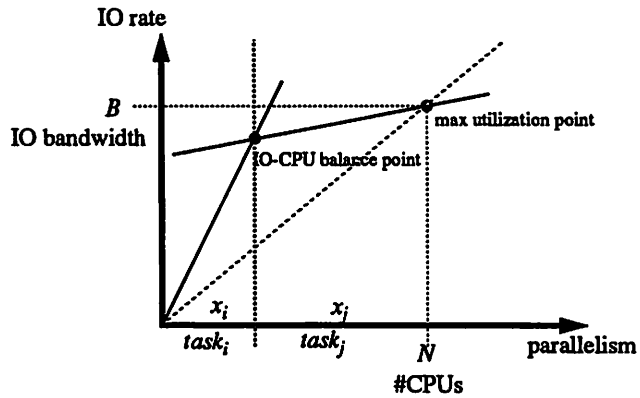

Figure 4: **IO-CPU Balance Point**

Suppose that $C_i > C_j$, in order to make x_i and x_j both positive, we have to have $C_i > B/N$ and $C_j < B/N$. In other words, one task has to be IO-bound and the other CPU-bound. This formula also tells us that one IO-bound task plus one CPU-bound task can always achieve maximum system resource utilization with appropriate parallelism assignment. Although a combination of more than two tasks may also achieve the same effect, it complicates the scheduling algorithm and consumes more memory. Therefore, in exploiting inter-operation parallelism, it is sufficient to only run two tasks at a time, i.e., we never need to run more than two tasks in parallel. This result significantly simplifies the scheduling problem.

Figure 4 gives a graphical interpretation to the above analysis. Given an IO-bound task and a CPU-bound task, if we draw the line corresponding to the IO-bound task through the origin and the line corresponding to the CPU-bound task through the upper right corner of the rectangle, there will always be an intersection within the rectangle between the two lines, which is the IO-CPU balance point. We can see that if we run task i with parallelism x_i and task j with parallelism x_j, the system will be running at the maximum utilization point.

The above calculation assumes that the disk bandwidth B is a predefined constant. However, in reality, disks have two bandwidths, i.e., a sequential i/o bandwidth and a random i/o bandwidth, where the random i/o bandwidth is about 1/3 of the sequential i/o bandwidth. We have to take this into account and do a more careful calculation for parallelism involving sequential i/o tasks.

Suppose that tasks f_i and f_j both generates sequential i/o's. Let B_s be the sequential read bandwidth and B_r be the random read bandwidth. Unfortunately the real effective bandwidth B can not be pre-computed because it depends on the ratio of the time that the disks spend in handling i/o's from each of the tasks. If the disks spend *most* of their time handling i/o's from one task, then in effect the disks still do mostly sequential i/o's, so $B \approx B_s$. However, if the disks spend half the time on one task and half on the other, then the disks have to seek between the blocks of one task and those of the other, so $B \approx B_r$. From the above equations, we know that the ratio is given by $C_i x_i / C_j x_j$, or $C_j x_j / C_i x_i$. Therefore we can calculate the effective bandwidth B as below,

$$B = \begin{cases} B_r + (1 - C_i x_i / C_j x_j)(B_s - B_r) & \text{if } C_i x_i \leq C_j x_j \\ B_r + (1 - C_i x_i / C_j x_j)(B_s - B_r) & \text{otherwise.} \end{cases}$$

For inter-operation parallelism between two sequential i/o tasks, we need to add this third equation to the above equations to compute the correct IO-CPU balance point. Similarly, we can also compute the correct IO-CPU balance point between a sequential i/o task and a random i/o task. However, because of the disk bandwidth drop, inter-operation parallelism may lose its advantage over intra-operation parallelism. In other words, for sequential scan tasks it may be better to run the tasks one by one using only intra-operation parallelism to avoid disk seeking between tasks. Therefore, in considering running two sequential scan tasks in parallel, we need to compare the estimated time of execution using inter-operation parallelism based on the above equations and the estimated time of execution using only intra-operation parallelism and decide whether inter-operation parallelism is worthwhile.

2.4 Dynamic Adjustment of Parallelism

Even though we have known how to calculate the IO-CPU balance point given an IO-bound task and a CPU-bound task, we have not yet solved the scheduling problem, because by running the two tasks at their IO-CPU balance point only guarantees full resource utilization while both tasks are running, when one task finishes first and there is no other job to fill in the newly available resources, resources are still wasted. Now the question is how to order the execution of tasks so that resource waste is minimized. We can model this problem as a modified version of the bin packing problem or the multi-processor scheduling problem in combinatorial optimization as in [PIRA90]. However, given the NP-hard complexity of the problem, it will not be efficient to try to solve this problem directly. In XPRS, this combinatorial problem is avoided by a mechanism of dynamic adjustment of intra-operation parallelism taking advantage of the low communication overhead in a shared-memory environment.

As described in [HONG91], in XPRS, intra-operation parallelism is implemented in two ways, *page partitioning* and *range partitioning*. In page partitioning, we partition relations across disk page boundaries and assign a subset of disk pages to each participating processors to work on. Specifically, given n processors, processor i processes disk pages $\{p \mid p \ mod \ n = i\}$, where $i = 0, 1, \ldots, n-1$. In range partitioning, we partition relations according to the value of a certain attribute. We try to find a balanced range partition with data distribution information in the system catalog or in the root node of an index. Page partitioning is used for sequential scans while range partitioning is used for index scans. Joins are parallelized using either page partitioning or range partitioning depending on the type of scans in their inner and outer plans. Different parallelism adjustment mechanisms have been designed for page partitioning and range partitioning operations.

Suppose that the current degree of parallelism for a task is n and we want to adjust it to a new degree of n', where n' can be greater than n, which means that we are putting in more available processors to work on this task, or smaller than n, which means that we are taking some processors away from this task to work on another task. We have implemented dynamic parallelism adjustment in XPRS as follows:

- For Page Partitioning

  The master backend sends a signal to all participating slave backends on the task. Upon receiving the signal, each slave backend, $i = 0, 1, \ldots, n-1$, sends back the current page number $curpage_i$ that it is scanning. After receiving all the page numbers from the slave backends, the master backend computes the maximum page number,

$$maxpage = max\{curpage_i\}, \ i = 0, 1, \ldots, n-1.$$

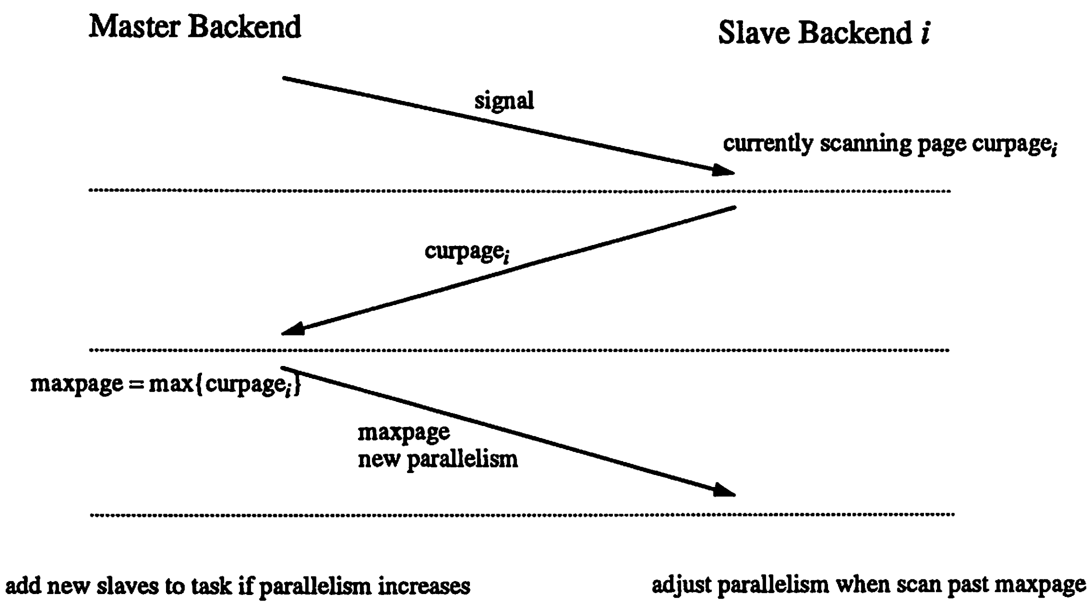

Figure 5: **Page Partitioning Parallelism Adjustment**

Then the master backend sends *maxpage* and new parallelism n' to all the slave backends, which completes the communications between the master and the slaves for the parallelism adjustment. If $n' > n$, the master backend will start $n' - n$ free slave backends to work on the task and make each of them start scanning after page *maxpage*. After receiving *maxpage* and n', all the slave backends will resume their work until they finish scanning all the pages before *maxpage*, at which point, they will change from scanning every nth page to scanning every n'th page and complete the parallelism adjustment. If $n > n'$, upon scanning past *maxpage*, the slave backends i, $i >= n' - 1$ will finish processing the current task and report back to the master backend of their availability. The communication process between the master backends and the slave backends for page partitioning parallelism adjustment is shown in Figure 5.

- For Range Partitioning

The master backend sends a signal to all the participating slave backends on the task. Upon receiving the signal, each slave backend sends back the intervals of values that remain for them to scan. For example, if a slave backend is assigned to scan for values

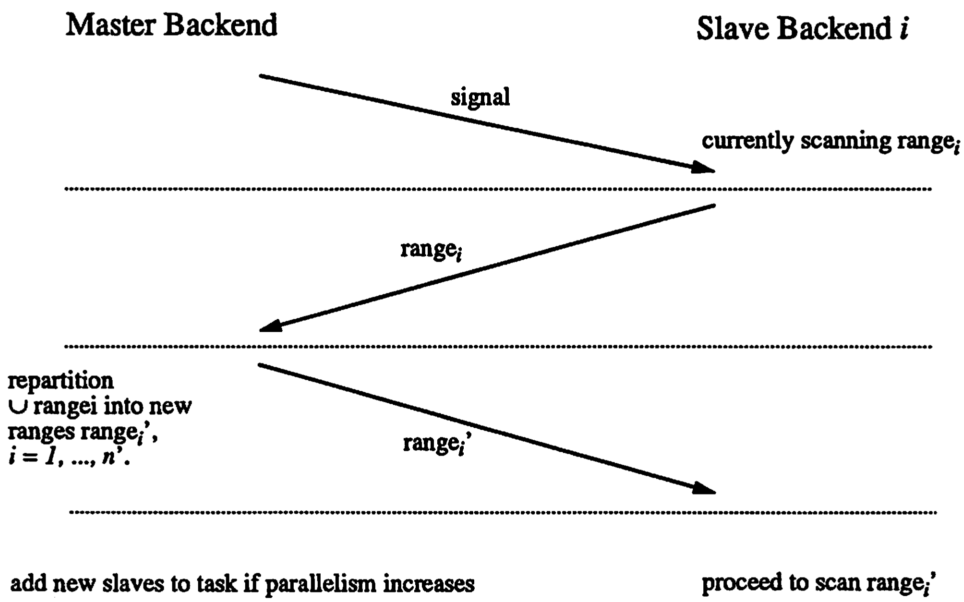

Figure 6: **Range Partitioning Parallelism Adjustment**

in interval $[l, h]$ and the current value that is being examined is c, the interval that will be sent back to the master backend is $[c, h]$. After parallelism adjustment, each slave backend may get more than one intervals to scan instead of only one contiguous interval. Upon receiving all the intervals from the slave backends, the master backend redistributes the intervals among n' slave backends and sends each slave backend a new set of repartitioned search intervals. If $n' > n$, the master backend will start $n' - n$ newly available slave backends to work on the task. Meanwhile, the old slave backends will resume their processing with the new search intervals. If $n' < n$, the extra slave backends will finish the processing of the current task and report back to the master backend as available. Figure 6 shows the communications between the master backend and the slave backends for range partitioning parallelism adjustment.

As we can see, our parallelism adjustment mechanism is made possible only by the low communication delay advantage of a shared-memory system.

2.5 Adaptive Scheduling Algorithm

The main idea of our scheduling algorithm is to use our dynamic parallelism adjustment mechanism to keep the system running at the IO-CPU balance point, i.e., at the maximum system resource utilization. At the same time, the algorithm also considers the possibility that inter-operation parallelism may be disadvantageous when sequential i/o is involved because of the disk seeks between tasks, in which case only intra-operation parallelism will be used to take advantage of the sequential disk bandwidth.

Given a set of n runable tasks, $S = \{f_1, f_2, \ldots, f_n\}$, suppose that the sequential execution time of f_i is T_i. And suppose that $T_{intra}(f_i)$ is the execution time of f_i using only intra-operation parallelism and $T_{inter}(f_i, f_j)$ is the execution time of f_i and f_j running at their IO-CPU balance point (x_i, x_j) with inter-operation parallelism. We have,

$$T_{intra}(f_i) = T_i/maxp(f_i),$$
$$T_{inter}(f_i, f_j) = min(T_i/x_i, T_j/x_j) + T_{ij}/maxp_{ij}.$$

where T_{ij} is the execution time of the remaining task when either f_i or f_j finishes first and $maxp_{ij}$ is the maximum intra-operation parallelism of the remaining task. We have,

$$T_{ij} = \begin{cases} T_i - T_j x_i/x_j & \text{if } T_i/x_i > T_j/x_j \\ T_j - T_i x_j/x_i & \text{otherwise,} \end{cases}$$

$$maxp_{ij} = \begin{cases} maxp(f_i) & \text{if } T_i/x_i > T_j/x_j \\ maxp(f_j) & \text{otherwise.} \end{cases}$$

The following is the description of our algorithm.

1. Divide S into S_{io} and S_{cpu} such that $S = S_{io} \bigcup S_{cpu}$, S_{io} contains all the IO-bound tasks and S_{cpu} contains all the CPU-bound tasks.

2. Choose $f_i \in S_{io}$ and $f_j \in S_{cpu}$, $S_{io} = S_{io} - \{f_i\}$, $S_{cpu} = S_{cpu} - \{f_j\}$.

3. Calculate the IO-CPU balance point between f_i and f_j, (x_i, x_j) and $T_{inter}(f_i, f_j)$.

4. If $T_{inter}(f_i, f_j) < T_{intra}(f_i) + T_{intra}(f_j)$, execute f_i with parallelism x_i if f_i is a new task or adjust the current parallelism of f_i to x_i if f_i is a running task; execute f_j with parallelism x_j if f_j is a new task or, adjust the current parallelism of f_j to x_j if f_j is a running task. Otherwise, execute f_i alone with parallelism $maxp(f_i)$ until completion, then execute f_j alone with parallelism $maxp(f_j)$ until completion.

5. If there is no task running, go to 2.

6. If f_i finishes first while f_j is still running, choose new $f_i \in S_{io}$, $S_{io} = S_{io} - \{f_i\}$, go to 3.

7. If f_j finishes first while f_i is still running, choose new $f_j \in S_{cpu}$, $S_{cpu} = S_{cpu} - \{f_j\}$, go to 3.

8. If $S_{io} = \emptyset$ or $S_{cpu} = \emptyset$, execute remaining tasks using intra-operation parallelism only.

In the above algorithm, we use an obvious strategy to choose the pair of IO-bound and CPU-bound tasks, f_i and f_j to execute in parallel, namely, to pair up the most IO-bound task, i.e., the task with the greatest i/o rate, and the most CPU-bound task, i.e., the task with the smallest i/o rate. In this way, we can keep the system running closer to the maximum utilization point (the upper-right corner of the rectangle in Figure 4) when either IO-bound or CPU-bound tasks run out first, because the remaining tasks will be those corresponding to lines closer to the diagonal in Figure 4. In a multi-user environment, if we want to minimize the response time of individual queries instead of the the total elapsed time, a shortest-job-first heuristic can be used, i.e., to execute the tasks from shortest queries first.

The above algorithm can be easily extended to handle a continuous sequence of tasks $\{f_1, f_2, f_3, \ldots\}$ instead of a fixed set of tasks. All we need to do is to represent S_{io} and S_{cpu} as queues. When a task arrives, it is put into either the S_{io} queue or the S_{cpu} queue according to its i/o rate. The rest of the algorithm still work as described.

3 Evaluation of Scheduling Algorithms

In this section we evaluate the performance of our scheduling algorithm described in the previous section through XPRS benchmark experiments. Currently XPRS is running on a Sequent Symmetry system with 12 processors and 4 disks. In the experiments, we compare the performance of the following three scheduling algorithms.

- **INTRA-ONLY** – No Inter-Operation Parallelism

  Execute tasks one by one using intra-operation parallelism only.

- **INTER-WITHOUT-ADJ** – Inter-Operation Parallelism without Dynamic Adjustment

  Almost the same as the algorithm in the previous section, except that when one task finishes first, no dynamic parallelism adjustment is performed. The master backend will simply start the task that can get closest to maximum utilization point if executed using the currently available processors in parallel with the running task.

- **INTER-WITH-ADJ** – Inter-Operation Parallelism with Dynamic Adjustment

  The algorithm described in the previous section.

We will run the following four workloads against each of the three algorithms:

- all IO-bound tasks,
- all CPU-bound tasks,
- extremely IO-bound tasks with extremely CPU-bound tasks,
- random-mix tasks.

Each workload consists of ten tasks. Since our algorithms only depend on the i/o rate of each task and other details of the operations in the tasks do not affect the performance. we choose all the queries to be one-variable selection queries for simplicity. Hence, all the tasks will be either a sequential scan or an index scan. The length of each task is randomly chosen between scanning 100 tuples and scanning $10,000$ tuples. We adjust the i/o rate of each task by varying the size of tuples that are scanned. All relations in the workloads have the same schema: $r_i(a = int4, b = text)$, where attribute b is a variable-size string and is used to adjust the tuple sizes. All queries will be a selection on $r_i.a$. An unclustered index may be created on a to make index scans possible. For sequential scans, the i/o rate is determined by the tuple size. There is a fixed per-tuple overhead (evaluation of query qualifications) after each tuple is read into memory from disks. Therefore, the time between two i/o requests is equal to the time to read in a disk page plus the time to process all the tuples that reside in the read-in disk page. When tuple size is small, many tuples can be packed into one disk page, thus it takes longer to process all the tuples in a page and the i/o rate is lower, so the task is likely to be CPU-bound. On the other hand, if tuple size is large, the i/o rate is higher and the task is likely to be IO-bound. For index scans on an unclustered index, however, the i/o rate is always high because index scans can follow the pointer in an index to a qualified tuple on a disk page and hence the time between two i/o requests is small. Index scans on an unclustered index are most likely IO-bound. For index scans on a clustered index, it is more or less the same situation as that of sequential scans.

In our experiments, the most CPU-bound task is a sequential scan on relation r_{min} in which the b attribute in all the tuples is set to NULL so that the tuple size is the smallest. The most IO-bound task is a sequential scan on relation r_{max} in which the b attribute in all the tuples is set large enough so that each disk page can only hold one tuple. In XPRS, the disk page size is 8K bytes. We have measured the i/o rate of sequential scans on both r_{min} and r_{max}. The r_{min} i/o rate is 5 (ios/second) and the r_{max} i/o rate is 70 (ios/second). All other tasks will have i/o rate in between. We have measured the bandwidth of our disks (bandwidth after file system overhead) to be 97 io's/second for sequential reads, 60 io's/second for almost sequential reads and 35 io's/second for random reads. In parallel executions, we at most see the almost sequential read bandwidth because even for parallel sequential scans, the reads may become unordered due to the asynchronousness of the parallel backends. Since we use 4 disks, we have a total i/o bandwidth of $4 * 60 = 240$ io's/second, and because we use 8 processors in our experiments, according to our definition, those tasks with i/o rate above 240/8 = 30 are IO-bound and those below 30 are CPU-bound. We choose the i/o rate of the tasks in our experiments as in the following table.

Type of Tasks	IO Rate (ios/second)
CPU-bound	randomly chosen in [5, 30)
IO-bound	randomly chosen in (30, 60]
Extremely CPU-bound	randomly chosen in [5, 15]
Extremely IO-bound	randomly chosen in [60, 70]

We run each workload in XPRS using each of the above three algorithms and measure the turnaround time of each run. Our experiment result is presented in Figure 7. As we can see, when the workload is all IO-bound or CPU-bound, all three algorithms have roughly the same performance and inter-operation parallelism does not help. It is sufficient to use intra-operation parallelism only. However, when there is a mixed workload of IO-bound and CPU-bound tasks, our proposed scheduling algorithm INTER-WITH-ADJ can improve performance by as much as 25% over INTRA-ONLY. We can also see that INTER-WITHOUT-ADJ loses to INTRA-ONLY because without parallelism adjustment a task may have to run with a low parallelism even when other tasks have finished and more processors are available.

4 Optimization of Bushy Tree Plans for Parallelism

In the previous sections, we have studied the scheduling problem of a set or a sequence of runable tasks. Our algorithm can be used regardless of whether the parallel tasks are from a bushy tree plan of the same query or from different queries. In this section, we will concentrate on the optimization problem of parallel execution plans for a single query and propose an optimization strategy based on the scheduling algorithm in the previous sections.

Since we have shown that a proper combination of intra-operation parallelism and inter-operation parallelism wins over only intra-operation parallelism given a workload of mixed IO-bound and CPU-bound tasks, the left-deep-tree-only and intra-operation-parallelism-only

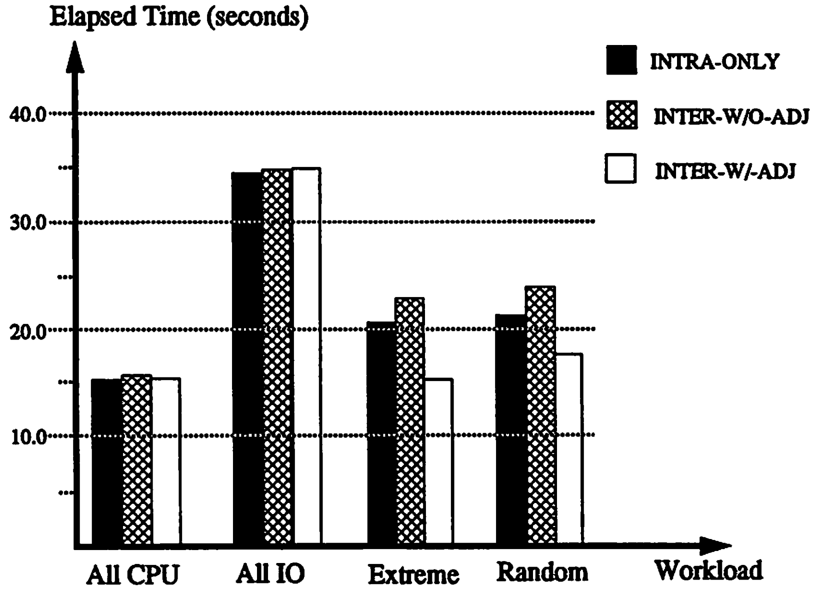

Figure 7: **Experiment Results of Scheduling Algorithms**

optimization strategy proposed in [HONG91] obviously cannot always take full advantage of all available resources and thus cannot guarantee the optimality of the execution plan chosen. However, in a multi-user environment, this problem can be easily solved by combining the two-phase optimization strategy in [HONG91] with our scheduling algorithm. We still find the best parallel plan for each query using only intra-operation parallelism with the algorithm in [HONG91], but we rely on the tasks from different queries submitted by multiple users to achieve maximum resource utilizations using our scheduling algorithm. In this section, we will focus on the optimization problem of a single query in a single-user environment where we have to depend on the the tasks within a same plan to achieve IO-CPU balance and where bushy tree plans have to be considered. Our idea is to preserve the same optimization scheme as in [HONG91], but use a new cost estimation method to estimate and compare the costs of bushy tree plans.

Since use of parallelism only helps reduce response time of a query execution, we only consider response time as our cost measurements in the following discussions. For each sequential plan p, let $seqcost(p)$ be the cost of sequential execution of p and $parcost(p, n)$ be the cost of parallel execution of p on n processors. As described before, plan p can be decomposed into a set of plan fragments which are what we call tasks in parallel executions. Unlike the situation in the previous sections, the tasks here have order-dependencies among themselves because some task may take the output of another task as input. However, obviously our scheduling algorithm can be easily modified to deal with the order-dependencies. It only needs to check if a task is ready before choosing it to execute and only execute the ready tasks. Suppose that $F(p) = \{f_1, f_2, \ldots, f_k\}$ is the set of plan fragments (tasks) of plan p. Using the cost estimation methods in conventional query optimization, we can estimate the sequential execution time of each task i, T_i. We can also estimate the number of i/o's of each task i, D_i. Thus, we can estimate the i/o rate of each task i as $C_i = D_i/T_i$. Let $T_n(S)$ be the elapsed time of executing a set of tasks, S with n processors. We can compute $T_n(S)$ with the following recursive formula:

$$T_n(S) = \begin{cases} T_i/maxp(f_i) + T_n(S - \{f_i\}) & \text{if } f_i \text{ is run alone,} \\ min(T_i/x_1, T_j/x_2) + T_n((S - \{f_i, f_j\}) \bigcup \{f_{ij}\}) & \text{if } f_i \text{ and } f_j \text{ is run in parallel.} \end{cases}$$

where f_i and f_j are two ready tasks chosen in S according to our scheduling algorithm to execute in parallel at IO-CPU balance point (x_1, x_2), f_{ij} is the remaining task of the longer of f_i and f_j when one of them finishes first. This formula is derived directly from our scheduling algorithm and is self-explanatory. We compute parallel execution cost of a plan as,

$$parcost(p, n) = T_n(F(p)).$$

Now for each plan p, we can estimate $parcost(p, n)$ given n and since we are assuming a single-user environment, n is known beforehand. Therefore, we can find the plan that minimizes $parcost(p, n)$. The optimal plan can be found by a conventional query optimization algorithm with $parcost(p, n)$ replacing $seqcost(p)$. Note that the calculation of $parcost(p, n)$ depends on the structure of the entire plan tree of p which makes local pruning, a common complexity-reducing technique in conventional query optimization infeasible. Aside from this factor, we can solve the parallel optimization problem with the same algorithm complexity as in conventional query optimization.

5 Conclusion

In this paper, we have presented our approach to exploit inter-operation parallelism in a shared-memory environment. We have first studied the scheduling problem of a set or a sequence of independent tasks that are either from a bushy tree plan of a single query or from the plans of multiple queries and proposed a clean and simple scheduling algorithm. Our scheduling algorithm achieves maximum resource utilizations by running two carefully selected tasks in parallel at their IO-CPU balance point, and avoids the combinatorial optimization problem by dynamically adjusting the degree of parallelism of the tasks to keep the system running with maximum resource utilizations. It takes full advantage of the low communication overhead feature of a shared-memory system, which a shared-nothing system does not have. We have also studied the optimization problem of parallel executions of a single query and extended our previous result to also consider inter-operation parallelism by introducing a cost estimation method for parallel execution costs of a sequential plan based on our scheduling algorithm.

In this paper, we have neglected the memory constraints on parallelism. For example, we cannot run two hashjoins in parallel unless there is enough memory for both hash tables. As future work, we will integrate memory constraints into our scheduling and optimization algorithms. So far, we have only studied the parallel optimization problem of a single query. We also plan to extend our results to deal with parallel optimization of multiple queries.

References

[BHID88] Bhide, A. and Stonebraker, M., "A Performance Comparison of Two Architectures for Fast Transaction Processing," Proc. 1988 IEEE Data Engineering.

[COPE88] Copeland, G., et. al., "Data Placement in Bubba," Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, IL, June 1988.

[DEWI86] Dewitt, D., et. al., "GAMMA: A High Performance Dataflow Database Machine," Proc. 1986 VLDB.

[DEWI90] Dewitt, D., et al, "The Gamma Database Machine Project," IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990.

[GRAE90] Graefe, G., "Encapsulation of Parallelism in the Volcano Query Processing System," Proc. 1990 ACM-SIGMOD.

[HONG91] Hong, W. and Stonebraker, M., "Optimization of Parallel Query Execution Plans in XPRS," Proc. 1st IEEE International Conference on Parallel and Distributed Information Systems, Dec. 1991.

[LU91] Lu, H., et al, "Optimization of Multi-Way Join Queries for Parallel Execution," Proc. VLDB, 1991.

[PIRA90] Pirahesh, H., et al, "Parallelism in Relational Database Systems: Architectural Issues and Design Approaches," Proc. of the 2nd International Symposium on Database in Parallel and Distributed Systems, July 1990.

[SCHN90] Schneider, D. and Dewitt, D., "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines," Proc. 1990 VLDB Conference.

[STON86] Stonebraker, M., "The Case for Shared Nothing," Proc. 1986 IEEE Data Engineering.

[STON88] Stonebraker, M., et. al., "The Design of XPRS," Proc. 1988 VLDB.

[STON91] Stonebraker, S., et al, "The Postgres Next Generation DBMS," CACM vol. 34 no. 10, October 1991.