# Optimization of Parallel Query Execution Plans in XPRS

Wei Hong and Michael Stonebraker
Computer Science Division, EECS Department
University of California at Berkeley
Berkeley, CA 94720

## Abstract

*In this paper, we describe our approach to the optimization of query execution plans in XPRS* [1]*, a multi-user parallel database machine based on a shared-memory multiprocessor and a disk array. The main difficulties in this optimization problem are the compile-time unknown parameters such as available buffer size and number of free processors, and the enormous search space of possible parallel plans. We deal with these problems with a novel two phase optimization strategy which dramatically reduces the search space and allows run time parameters without significantly compromising plan optimality. In this paper we present our two phase strategy and give experimental evidence from XPRS benchmarks that indicate that it almost always produces optimal plans.*

## 1 Introduction

XPRS (eXtended Postgres on Raid and Sprite) is a multi-user parallel database machine based on a shared-memory multiprocessor and a disk array. An outline of the initial design of XPRS can be found in [1]. The underlining reliable disk array RAID is described in [2].

Shared-memory multiprocessors are a very cost-effective way to achieve high performance. They have two major advantages over the *shared-nothing* architectures [3] adopted by most other parallel database machines such as GAMMA [4]. First, there are no communication delays because messages are exchanged through shared memory, and synchronization can be accomplished by cheap, low level mechanisms. Second, load balancing is much easier because the operating system can automatically allocate the next ready process to the first available processor. Simulation results in [5] show that the potential win of a shared-memory system over a shared-nothing system to be as much as a factor of two.

Database applications are often I/O intensive. In order to keep up with the I/O requests from multiple CPU's, XPRS uses a disk array to eliminate the I/O bottleneck. All relations are striped sequentially, block by block, in a round-robin fashion across the disk array to allow maximum I/O bandwidth.

[1]This research was sponsored by the National Science Foundation under contract MIP 8715235.

In this paper, we address the problem of parallel query optimization in a shared-memory environment and describe the XPRS approach. We will concentrate on two specific issues, namely dealing with compile-time unknown parameters and the large search space of parallel plans. In a multi-user environment, parameters such as the amount of available buffer space and number of free processors are unknown at compile time. Therefore, compile-time optimization must generate plans for an uncertain run-time environment. Second, the number of possible parallel query execution plans is so enormous that any exhaustive search algorithm is impractical. As a result, the search space must be heuristically reduced.

Our strategy is to divide query optimization into two phases. The first phase only deals with sequential query execution plans and fixed parameters and is performed at compile time. The second phase is performed at run time and finds the optimal parallelization of the best sequential plan chosen in the first phase. Obviously this two-phase optimization approach greatly reduces the plan search space because it only explores parallel versions of the best sequential plan. In this paper, we will present experiment results that show that this approach does not compromise optimality of the resulting parallel query execution plan.

Most previous work on parallel query optimization has been done for a shared-nothing environment, e.g., [6]. Earlier work on a shared-memory environment (e.g., [7]) only considers single operations, mainly joins and sorts. On the other hand we address the optimization of entire queries. [8] proposes an algorithm to parallelize a given sequential plan to achieve minimum duration time with computational resource requirements less than the given system bounds in a shared-memory environment. It does not address the problem of how to choose a sequential plan to parallelize, and it assumes that the amount of available resources is known and therefore the algorithm can not be used at compile time. In this paper, we will describe a way to handle unknown parameters at compile time and strategies to choose optimal sequential plans and their parallelizations. [9] proposes a general approach for dealing with unknown parameters in compile-time optimization by introducing *choose-plan* nodes to generate multiple query plans, consequently, the run-time system must go through a decision tree to choose a plan according to the current

0-8186-2295-4/91 $01.00 © 1991 IEEE

system parameters. Although this general approach can be applied to XPRS, our solution is much simpler because we are concerned only with unknown buffer sizes and number of processors. [10] shows through an example of a three-way nestloop joins for which the optimal sequential query plan may change if the buffer size changes. However, the example does not consider hashjoin as a join algorithm. In XPRS, on the other hand, we always assume that there is enough main memory (approximately the square root of the size of the smaller join relation [11]) to use a hashjoin algorithm, which has been shown to be optimal without the use of indices [11]. Our results show that with sufficient main memory to support hashjoins, the choice of the optimal query plan remains very stable with varying buffer sizes.

The rest of this paper is organized as follows. Section 2 explores the search space of parallel query plans and shows the complexity of the problem. Section 3 studies how buffer size affects the choice of optimal sequential plans, introduces a *choose* node that can dynamically switch join or scan methods based on real buffer sizes, and shows that the choice of optimal sequential plans augmented with *choose* nodes is insensitive to buffer size changes. Section 4 then discusses the performance of various parallelizations of a sequential plan and justifies our two-phase optimization approach. Section 5 gives an overview of the whole process of XPRS parallel query processing, while section 6 concludes the paper and suggests future research.

## 2 The Space of Parallel Plans

In a uniprocessor environment, a query execution plan (which we call a *sequential plan*) is a binary tree consisting of the basic relational operation nodes. In XPRS, the basic operations include **sequential scan, index scan, nestloop join, mergesort join** and **hashjoin**. At run time, the query executor processes each plan sequentially in a postorder (depth-first) sequence. Intermediate result generation is avoided by the use of pipelining, in which the result tuples of one relational operation are immediately processed as the input tuples of the next operation. Figure 1 gives an example of a sequential plan for a four-way join, $A \bowtie B \bowtie C \bowtie D$. The shaded boxes in Figure 1 represent plan fragments in a possible parallelization of the sequential plan, which we will discuss momentarily. Notice that the inner relation (the right subtree) of the hashjoin at the root of the plan tree is also a join. This kind of plans are called *bushy* tree plans. Most conventional query optimizers [12] only consider *left-deep* tree plans that do not allow inner relations to be a join in order to reduce the search space of possible plans.

We call query processing plans that specify a parallel execution *parallel plans*. Obviously, each parallel plan is a *parallelization* of some sequential plan and each sequential plan may have many different parallelizations. Parallelizations can be characterized in the following three ways.

- **Form of Parallelism**

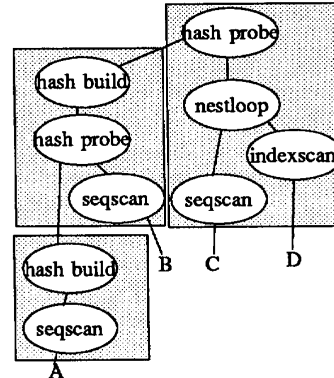


Figure 1: **Example Execution Plan**

There are two forms of parallelism that we can exploit in query processing: ***intra-operation*** parallelism and ***inter-operation*** parallelism. Intra-operation parallelism is achieved by partitioning data among multiple processors and having those processors execute the same operation in parallel. Inter-operation parallelism is achieved by partitioning the query plan and executing different operations in parallel. As we will discuss in Section 4.1, all of the basic relational operations can achieve intra-operation parallelism in a straightforward manner. Moreover, if we have a bushy-tree plan, we might have two operations that do not depend on each other's output. Therefore they can be executed concurrently with inter-operation parallelism. For example, the sequential plan in Figure 1 can be parallelized with intra-operation parallelism in each node and inter-operation parallelism between $A \bowtie B$ and $C \bowtie D$.

- **Unit of Parallelism**

  *Unit of parallelism* refers to the group of operations that is assigned to the same process for execution. We also call a unit of parallelism a *plan fragment* since it is a "fragment" of a complete plan tree. Each plan fragment will be executed in a pipelined fashion, and consequently should not contain any "blocking" between operations. Blocking happens for certain operations, such as hash and sort, when one operation has to wait for another operation to finish producing all the tuples before it can proceed. Plan fragments should be chosen so that such blockings only occur at plan fragment boundaries. For example, the shaded boxes in Figure 1 show that the sequential plan has three plan fragments with blocking at the boundaries.

- **Degree of Parallelism**

  *Degree of parallelism* is the number of processes that are used to execute a plan fragment. Presumably we want the degree of parallelism to be as high as possible; however, excessive parallelism may cause high

resource contention, resulting in a loss of performance as will be shown in Section 4.1. An optimal degree of parallelism must be decided according to run-time system resource availability to achieve maximum performance.

In this paper only intra-operation parallelism and left-deep tree sequential plans are considered for the following reasons.

- It is much easier to achieve load balancing in intra-operation parallelism because we can always partition the input data carefully into equal size portions. Load balancing becomes much harder with inter-operation parallelism because different operations may have very different complexity and different sizes of input data.
- Intra-operation parallelism requires less main memory. For example, if we want to execute two hashjoins with inter-operation parallelism, we need to allocate two hash tables, while intra-operation parallelism only requires one hash table at a time.
- Inter-operation parallelism can only help mix up CPU-bound and I/O-bound operations. A more detailed discussion on this subject is deferred to a subsequent paper.

The overall performance goal of a multiprocessor database machine is to obtain increased throughput as well as reduced response time in a multiuser environment. The objective function that XPRS uses for query optimization is a combination of resource consumption and response time as follows:

$$cost = resource_consumption + w \times response_time.$$

Here $w$ is a system-specific weighting factor.

Our optimization problem is to find the parallel plan with minimum cost among all possible parallelizations of all possible sequential plans of a query. Obviously, the search space of parallel plans is orders of magnitude larger than the search of space of sequential plans; therefore query optimization by exhaustive search[12] is impractical. The second difficulty is that many system parameters that affect query execution cost in a multi-user environment are unknown at compile time. In this paper, we specifically consider two of such parameters: available buffer size and number of free processors. Buffer size not only affects the buffer hit rate but also determines the number of batches in a hashjoin[11]. The number of free processors determines the possible speedup of query execution. The following two design hypotheses that we will justify in Section 3 and 4 dramatically simplify the optimization problem.

For ease of exposition, we assume that the buffer size and number of free processors are fixed during the entire query execution. As we will describe in Section 5, our operational prototype only fixes these parameters during the execution of individual plan fragments.

Let $BPP(Q, NBUFS, NPROCS)$ be the best parallel plan for query $Q$ given $NBUFS$ units of buffer space and $NPROCS$ free processors, $BP(Q, NBUFS)$ be the best sequential plan for $Q$ given $NBUFS$ units of buffer space, $Cost(P, NBUFS)$ be the cost of a sequential plan $P$ given $NBUFS$ units of buffer space, and $PARALLEL(P)$ be the set of possible parallelizations of a sequential plan $P$.

1. **The Buffer-Size-Independent Hypothesis**

   *The choice of the best sequential plan is insensitive to the amount of buffer space available as long as the buffer size is above the hashjoin threshold, i.e.,*

   $$Cost(BP(Q, NBUFS), NBUFS) \approx Cost(BP(Q, NBUFS'), NBUFS),$$

   *where* $NBUFS \neq NBUFS', NBUFS \geq T, NBUFS' \geq T,$ *and* $T$ *is the hashjoin threshold.*

   As we will discuss in details in the next section, certain exceptions to the above hypothesis do exist, however, they can be localized within specific operations and handled with a mechanism that dynamically chooses the implementation of an operation at run time according to real buffer sizes.

2. **The Two-Phase Hypothesis**

   *The best parallel plan is a parallelization of the best sequential plan, i.e.,*

   $$BPP(Q, NBUFS, NPROCS) \in PARALLEL(BP(Q, NBUFS)).$$

We will justify these two hypotheses with expermental results in the following two sections.

# 3 Effect of Buffer Size on Query Execution Cost

In this section, we study the effect of buffer size on execution cost of sequential plans and justify the buffer-size-independent hypothesis by running all possible plans for a large number of queries in XPRS and determining the cost of the best plan for each buffer size. In general, the cost of a sequential plan is measured by resource consumption, which is a linear combination of I/O cost and CPU cost as follows,

$$Cost = \#page_io + c \times \#tuples_processed.$$

Here $c$ is another system-specific weighting factor.

Because buffer sizes only affect the I/O cost of query execution, we only need to measure the I/O costs of real query executions in experiments to follow. Moreover, we use a LRU buffer replacement strategy in all the experiments and have all query executions start with an empty buffer. Before we present these experiments, we must deal with two exceptions by introducing *choose* nodes in XPRS plans.

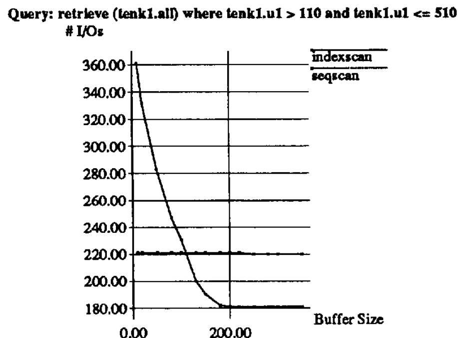


Figure 2: **Cost of SeqScan v.s. IndexScan**

## 3.1 Execution Cost of Single Operations and Choose Nodes

We have identified two situations in the execution of single operations that may cause problems with the buffer-size-independent hypothesis. One is choosing between an indexscan using an unclustered index and a sequential scan. The other is choosing between a nestloop with an indexscan over the inner relation and a hashjoin.

A sequential scan only needs one buffer page and additional buffer pages do not reduce query execution cost. However, the cost of an indexscan using an unclustered index is very sensitive to buffer size. If there is enough buffer space to hold all the pages that need to be fetched during the indexscan, then we only need to read each of those pages once. Therefore, with sufficient buffer space, if an indexscan touches less than all pages, it will have lower cost than a sequential scan. On the other hand, with few buffers, an indexscan may end up fetching the same page from disk many times and becomes more expensive than a sequential scan. Figure 2 gives an example of this situation by plotting the cost of each plan versus buffer size for a selection query on a 10,000-tuple relation from the Wisconsin benchmark. Obviously, when the two curves cross, we require a mechanism to switch from a sequential scan to an index scan.

In addition, a similar situation occur in choosing between a nestloop with an indexscan over the inner relation and a hashjoin. Assuming sufficient buffer space, a nestloop with indexscan will fetch ultimately all the pages in the outer relation and all the pages in the inner relation that match some tuple in the outer relation plus a small number of index pages. On the other hand, a hashjoin will need to fetch all the pages in both the outer and inner relations. If the join selectivity is small, the nestloop plan may not need to fetch all the pages in the inner relation and therefore will have a lower cost than the hashjoin plan. On the other hand, with few buffers, the indexscan in nestloop may have to fetch the same page many times and cause the nestloop to become more expensive than a hashjoin. Therefore, there may exist a "cross over" point between nestloop with an indexscan and hashjoin.

To solve the two "cross over" problems, we introduce two new *choose* nodes in the query tree, which can choose between the two join or two scan methods according to run-time buffer size. With these *choose* nodes, we now demonstrate experimental evidence that supports the validity of the buffer-size-independent hypothesis.

## 3.2 Experiments on the Buffer Size Independent Hypothesis

In the experiments, we first run a wide variety of benchmark queries using all the possible execution plans under different buffer sizes and measure the real execution costs and then calculate the the relative error that results from the buffer-size-independent hypothesis. We will show that such errors are extremely small.

We have used queries from two benchmarks. The first benchmark is the Wisconsin benchmark [13], a standard benchmark for measuring query processing power. Because the Wisconsin benchmark has no more than 3-way joins and has limited join selectivities, we also developed a random benchmark generator to generate additional varieties of complex queries.

The relations in our random benchmark have the following POSTQUEL definition:

```
create r (ua1=int4, a1=int4, ua20=int4,
          a20=int4, ua50=int4, a50=int4,
          ua100=int4, a100=int4,
          filling=text)
```

We generated 10 random relations with cardinalities ranging from 100 to 10,000. All the integer attribute values are randomly distributed between 0 and 9,999. All the "ua" attributes are unclustered attributes and all the "a" attributes are clustered attributes. The number following "ua" or "a" indicates the number of times each value is repeated in the attribute. For example, each value in "ua20" is duplicated 20 times. We define indices on all the integer attributes so that the optimizer can always have the choice of generating an indexscan. The attribute "filling" is a variable length string, and is used to vary the tuple size of different relations. The generator can make the "filling" attribute 68 bytes or 968 bytes so that resulting tuple size is 100 bytes or 1000 bytes.

The queries in the random benchmark are generated in the following way. To generate a random join of $k$ relations, we first randomly choose $k$ relations. Then we start with the first relation in the chosen list and the rest in the unchosen list. We randomly pick a relation in the unchosen list, join it with a randomly picked relation in the chosen list on two randomly chosen attributes and move it from the unchosen list to the chosen list. We repeat this operation until the unchosen list becomes empty; and we have generated a random join on $k$ relations. Next we generate random selections on the relations. Each relation has a 50% probability of having a restriction of either an equal-

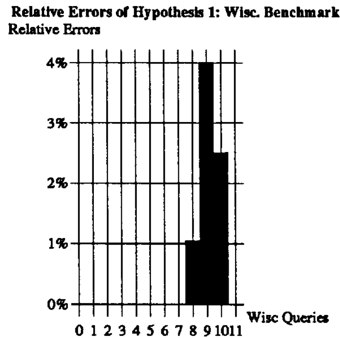


Figure 3: **Relative Errors of the Buffer Size Independent Hypothesis on the Wisconsin Benchmark**

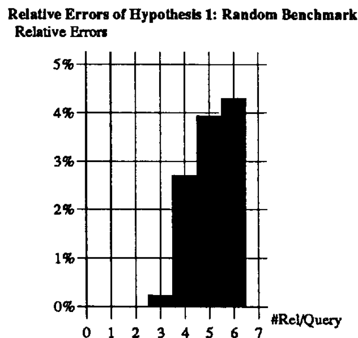


Figure 4: **Relative Errors of the Buffer Size Independent Hypothesis on the Random Benchmark**

ity condition or inequality conditions with a lower bound and a upper bound. The target list is also randomly selected from all the attributes of all the relations with each attribute having 50% probability of being chosen.

In the experiments, we first have the XPRS query optimizer generate all possible sequential plans for each query. If a plan is known to be dominated by another plan, it is discarded. For example, since we know that hashjoin is the best join plan without the use indices (assuming that our buffer size is above the hashjoin threshold)[11], we can always remove nestloop and mergejoin plans which do not use indices from the test plans, which cuts down our experiment running time substantially. Another example is that mergejoin plans with inner relation and outer relation exchanged always have the same cost and therefore only one of them need to be executed. After selecting interesting execution plans, we run each plan with different buffer sizes varying from the minimum amount of buffer space to make all hashjoins possible to $MAXBUFS$, the buffer size that can hold all the relations in main memory. For each execution we measure the actual execution cost. To avoid the problem of inaccurate estimate of intermediate result sizes to hashjoin, we always execute the plans that do not contain hashjoins first and then set the sizes of intermediate results to the real sizes. The effect of inaccurate size estimates on our results is left as a future research topic.

From the statistics collected in the experiments, we know for each query, $Q$, the real execution cost of each query plan under different buffer sizes, i.e., given any plan $P$ and buffer size $NBUFS$, we know $Cost(P, NBUFS)$ and $BP(Q, NBUFS)$. Let $BCP(Q, NBUFS)$ be the plan obtained by adding appropriate *choose* nodes to plan $BP(Q, NBUFS)$. Since we know the cost of all the plans that are only different from $BP(Q, NBUFS)$ in some join or scan methods, we can also calculate the cost of $BCP(Q, NBUFS)$ under any buffer sizes. The relative error of the buffer-size-independent hypothesis is computed with the following formula:

$FixBufCost = Cost(BCP(Q, MAXBUFS), NBUFS)$,
$MinCost = Cost(BP(Q, NBUFS), NBUFS)$,

$$Error(Q, NBUFS) = \frac{FixBufCost - MinCost}{MinCost}$$

Figure 3 shows the graph of average relative errors of the Wisconsin benchmark queries. For each query $Q$, we first compute $Error(Q, NBUFS)$ for each buffer size, then we compute the average relative error over the buffer sizes. As we can see from the graph, the first few queries have 0 error. This is because those are the selection queries and two-way join queries for which the *choose* nodes can always choose the optimal plan. Errors occur in the three-way join queries, but they are never above 4%.

Figure 4 shows the graph of average relative error on the random benchmark. 120 queries of up to 6-way joins are executed in the experiment. The relative error is averaged over queries that have the same number of relations (20 of them each). As we can see from the graph, the average relative errors remain below 5%.

Thus, two different benchmarks support the buffer-size-independent hypothesis. With choose nodes we have encapsulated enough of the plan switches required when buffer size changes so that average relative error is never above 5%.

# 4 Parallel Execution Cost of Sequential Plans

This section examines the costs of parallel executions. Section 4.1 discusses the implemenations of intra-operation parallelism in XPRS and shows that intra-operation parallelism can achieve near-linear speedup within the limit of system resources. Then Section 4.2 presents experiment results that support the two-phase hypothesis.

## 4.1 Implementations of Intra-operation Parallelism and their Performances

In XPRS, intra-operation parallelism is implemented in two ways: *page partitioning* and *range partitioning.* In

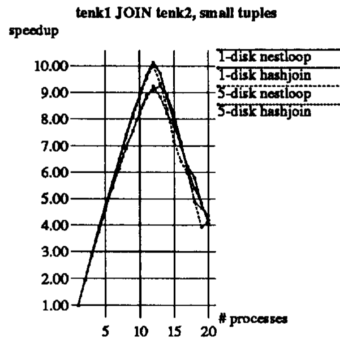


Figure 5: **Speedup of Parallel Join: small tuples**

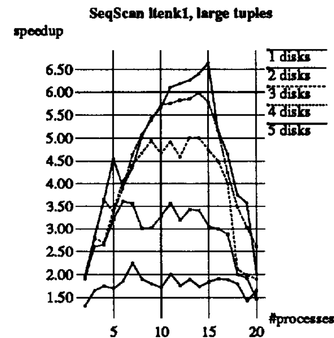


Figure 6: **Speedup of Parallel Seq. Scan: large tup.**

page partitioning we partition relations across disk page boundaries and assign a subset of disk pages to each participating process to work on. In range partitioning we partition relations according the value of a certain attribute. Page partitioning is used for sequential scans while range partitioning is used for index scans and sorting. We can use the data distribution information in the system catalog to obtain a well-balanced range partition. For an index scan, range partitioning can also be facilitated by using the keys in the B-tree root node. A join operation can be parallelized by appropriately parallelizing its outer and inner paths. We can parallelize both the outer and inner paths for hashjoin and mergejoin, while we only parallelize the outer path for nestloop join. Also note that parallel hashjoin is the only operation with a critical section. It requires parallel access to a shared hash table because multiple processes may insert tuples into the same hash bucket simultaneously. To minimize the probability of conflict, the number of hash buckets should be large compared to the degree of parallelism.

[4] has shown that intra-operation parallelism can achieve near-linear speedup in response time in a shared-nothing environments. We briefly show the same near-linear speedup in XPRS in a shared-memory environment. In addition, we show performance varying the number of disks and number of processes independently and also the degradation resulting from excessive parallelism.

Currently XPRS is operational on a Sequent Symmetry running the Dynix operating system with 12 CPU's connected via an 80MB/s bus and 5 disks controlled by 3 disk controllers .

In the experiments, we created the two 10,000-tuple relations from the Wisconsin Benchmark, *tenk1* and *tenk2*. Because the tuple sizes in the Wisconsin Benchmark relations are relatively small, we also created another 10,000-tuple relation, *ltenk1*, which has the same fields as tenk1 except that each tuple is filled to 1,000 bytes by an extra string field. Appropriate indices were created according to the benchmark specification. All the relations are striped across a set of disks using a simple *mod* function, i.e., block x, is stored on disk (x *mod* #disks). For example, if we only use two disks, then all the odd number blocks will be on one disk and all the even number blocks will be on the other. The relations are also partitioned among the parallel scan processes with a simple *mod* function, i.e., process i scans block x such that x *mod* #processes = i. Before each execution, the file system cache is always cleared so that no blocks of the test relations are left in memory. All the processes are pre-forked so that process startup overhead is negligible.

We have measured the speedup of parallel scans and joins on the above relations varying the number of processes and number of disks. Due to space limit, we can not present all the experiment results. We only present some sample results of join *tenk*1 $\bowtie$ *tenk*2 in Figure 5 and scan on *ltenk*1 in Figure 6. We have found that a sequential scan is CPU-bound when the tuples are small and becomes I/O-bound when the tuples get large. Moreover, index scans are I/O-bound because they do not need to examine every tuple in a page. Our experiment results show that parallel scans and joins can achieve close-to-linear speedup until they run out of processors if they are CPU-bound such as in Figure 5, or disk bandwidth if they are I/O-bound such as in Figure 6. Figure 5 also shows that the synchronization overhead caused by the shared hash table in hasjoins is negligible. In Figure 6, we see a drop in the speedup when the number of processes exceeds the number of disks. This is caused by the operating system readahead. When we have fewer processes than disks, the access pattern on each disk is sequential. Consequently normal file system readahead will prefetch the next disk block to be processed. When there are more processes than disks, the access pattern to each disk becomes random and the file system readahead is ineffective. Observe that there is always a performance drop when the degree of parallelism exceeds the number of processors. It results from the extra context switches and virtual memory overhead generated when the number of processes exceeds the number of processors. In addition, a process holding the shared buffer pool lock might be descheduled

and the convoy problem will occur. In a multi-user environment there will be multiple commands concurrently being processed, and the effective parallelism is the sum of the degree of parallelism of the individual command. It is important to ensure that this sum does not exceed the number of processors.

## 4.2 Experiments on the Two-Phase Hypothesis

The experiments that we run to justify the two-phase hypothesis are similar to those described in Section 3.2. We have used the queries from the Wisconsin benchmark and the random benchmark. For each test query, we first generate all the possible sequential plans as described in Section 3.2. We ran each plan with varying degrees of intra-operation parallelism and buffer sizes. The degree of parallelism is varied from 1 to 12 (the number of processors in our system) and the buffer size is varied in the same way as in Section 3.2. For each execution of a plan, $P$, with $NBUFS$ buffers and $NPROCS$ processes, we measure the resource consumption and response time, and compute the cost of the execution, $Cost(P, NBUFS, NPROCS)$. Let $BP(Q, NBUFS)$ be the best sequential plan with buffer size $NBUFS$. The relative error of the two-phase hypothesis is calculated with the following formula:

$TwoPhaseCost =$

$$Cost(BP(Q, NBUFS), NBUFS, NPROCS),$$

$MinCost = min_{\{P\}}\ Cost(P, NBUFS, NPROCS),$
$Error(Q, NBUFS, NPROCS) =$

$$\frac{TwoPhaseCost - MinCost}{MinCost}$$

Because the cost of parallel plans depends on the system-specific weighting factor, $w$, we need to calculate errors for different values $w$.

Figure 7 shows the average relative error of the Wisconsin benchmark queries. For each query, $Q$, the relative error, $Error(Q, NBUFS, NPROCS)$, is averaged over all combinations of $NBUFS$ and $NPROCS$. As we can see, for small values of $w$, the relative errors are near 0. For large values of $w$, the relative error never exceeds 8%.

Figure 8 shows the average relative error of the random benchmark. Due to the enormous of computing resources demands of this experiment, we have only run queries of up to 3-way joins from the random benchmark. As we can see, the average relative error never exceeds 6%.

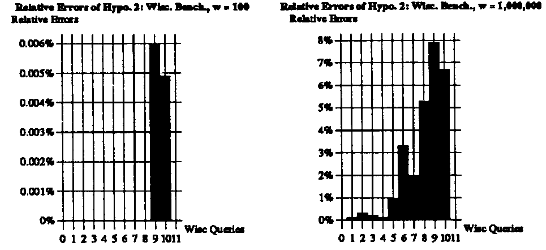


**Figure 7: Relative Errors of the Two-phase Hypothesis on the Wisconsin Benchmark**

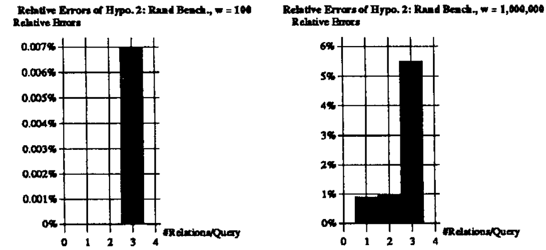


**Figure 8: Relative Errors of the Two-phase Hypothesis on the Random Benchmark**

# 5 XPRS Query Processing

Based on our two hypotheses, we can deal with compile-time unknown parameters and the enormous search space of parallel plan optimization by the following two-phase algorithm.

- **Phase 1.** Find the optimal sequential plan assuming the entire buffer pool is available, i.e., find $BP(Q, ALLBUFS)$ where $ALLBUFS$ is the size of the whole buffer pool. Add appropriate *choose* nodes to $BP(Q, ALLBUFS)$ to get a modified plan $BCP(Q, ALLBUFS)$.
- **Phase 2.** Find the optimal parallelization of the optimal sequential plan, i.e., find $min\{cost(PP, NBUFS, NPROCS) \mid PP \in PARALLEL(BCP(Q, ALLBUFS))\}$ where $NBUFS$ and $NPROCS$ are the run-time available buffer size and the number of free processors.

Because we have a fixed buffer size $ALLBUFS$ in Phase 1, it can be performed at compile time. Phase 2 still has to be performed at run time, because it takes the run-time parameters $NBUFS$ and $NPROCS$ into account and tries to dynamically determine the best parallelization of the sequential plan chosen in Phase 1.

Figure 9 gives an overall architecture of XPRS query processing. The XPRS optimizer is a modified version of the POSTGRES optimizer with a postprocessor that uses cost functions similar to those in [11] and [14] to determine if two join or scan methods (specifically sequential scan versus unclustered index scan and nestloop with index versus hashjoin) may have "cross over" points in their cost curves against buffer size. If such points exist, the postprocessor will modify the optimal plan generated by the optimizer by adding appropriate *choose* nodes.

The parallelizer takes a sequential plan and decomposes it into a set of plan fragments, decides the degrees of parallelism for each fragment and a processing schedule for all the fragments, then passes a collection of parallel versions of the first plan fragment to the parallel executor, which distributes the versions to a number of POSTGRES backend processes according to the degree of parallelism determined by the parallelizer. The multiple POSTGRES backend processes execute the versions in parallel and send an acknowledgements back to the parallel executor after they finish executing. The parallel executor will notify the parallelizer that another plan fragment is needed.

The XPRS parallelizer is the key mechanism for exploiting parallelism. It initially decomposes a sequential plan into plan fragments across blocking boundaries. Obviously larger plan fragments will cause less temporary relation overhead. However, the size of plan fragments is also constrained by the run-time available buffer space. For example, a plan fragment can contain as many as two hashjoins, i.e., one hash probe node followed by a hash build node. If the XPRS parallelizer cannot obtain the minimum required memory for both hashjoins, it will decompose the plan fragment into two smaller fragments between the hash probe node and the hash build node, save the intermediate results of the hash probe into a temporary relation and the following hash build node will read from the temporary relation after the previous hash probe is finished. Subsequent to the decomposition, we only need enough memory for one hashjoin at a time. The parallelizer also decides the degree of parallelism for each plan fragment which is constrained by the disk bandwidth and number of free processors. The performance curves in Section 4.1 have shown the consequences of excessive parallelism. The parallelizer first examines the system load average (the average number of ready processes in ready queue) and make sure not to create more processes than the available processors. Then it estimates I/O request rate of the plan fragment based on tuple sizes and operation types, and determines the maximum parallelism for the plan fragment without over saturating disk bandwidth.

## 6 Conclusion

In this paper, we have described our approach to the optimization of parallel query execution plans in a shared-memory multiprocessor environment. We have presented experimental results that justify our two-phase optimization approach, which reduces the complexity of parallel query optimization without significantly compromising optimality of the resulting parallel plan. The first phase of of our two-phase optimization is a conventional query optimization with a fixed buffer size that finds the optimal sequential plan augmented with appropriate *choose* nodes that are encapsulated in individual operations. The second phase finds the best parallelization of the sequential plan obtained from the first phase according to run-time eironment. Our approach achieves run-time flexibility while still doing most of the optimization at compile time.

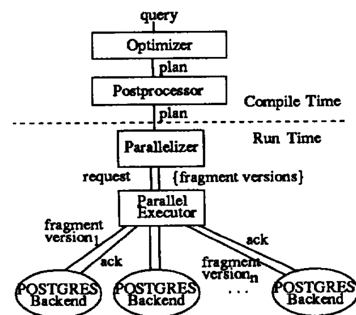


Figure 9: **Architecture of XPRS Query Processing**

As future work, we will study the tradeoffs between intra-operation parallelism and inter-operation parallelism more closely and look into the scheduling and memory allocation issues in processing multiple queries submitted by different users in parallel. In the experiments presented this paper, we have assumed perfect estimates of intermediate result sizes and uniform data distribution. The effects of inaccurate size estimates and skewed data distributions also need to be studied more carefully.

## References

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

[2] Patterson, D., et. al., "RAID: Redundant Arrays of Inexpensive Disks," Proc. 1988 ACM-SIGMOD.

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

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

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

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

[7] Murphy, M. and Rotem, D., "Processor Scheduling for Multiprocessor Joins," Proc. 1989 IEEE Data Engineering.

[8] Murphy, M. and Shan, M., "Execution Plan Balancing," Proc. 1991 IEEE Data Engineering.

[9] Graefe, G. and Ward, K., "Dynamic Query Evaluation Plans," Proc. 1989 ACM-SIGMOD.

[10] Cornell, D. and Yu, P., "Integration of Buffer Management and Query Optimization In Relation Database Environment," Proc. 1989 VLDB.

[11] Shapiro, L., "Join Processing in Database Systems with Large Main Memories," ACM-TODS, Sept. 1986.

[12] Selinger, P., et. al., "Access Path Selection in a Relational Data Base System," Proc. 1979 ACM-SIGMOD.

[13] Bitton, D., et. al., "Benchmarking Database Systems: A Systematic Approach," Proc. VLDB, 1983.

[14] Mackert, L., et. al., "Index Scans Using a Finite LRU Buffer: A Validated I/O Model," ACM-TODS, Sept. 1989.